

3.1

BORLAND® C++

PROGRAMMER'S GUIDE

- LANGUAGE STRUCTURE
- CLASS LIBRARIES
- ADVANCED PROGRAMMING TECHNIQUES
- ANSI C IMPLEMENTATION

B O R L A N D

Borland[®] C++

Version 3.1

Programmer's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

Copyright © 1991, 1992 by Borland International. All rights reserved.
All Borland products are trademarks or registered trademarks of
Borland International, Inc. Other brand and product names are
trademarks or registered trademarks of their respective holders.
Windows, as used in this manual, refers to Microsoft's
implementation of a windows system.

PRINTED IN THE USA.
10 9 8 7 6 5 4

C O N T E N T S

Introduction	1	Constants and internal representation	19
What's in this book	1	Constant expressions	20
An introduction to the formal definitions	2	Punctuators	21
Syntax and terminology	3	Brackets	21
Chapter 1 Lexical elements	5	Parentheses	21
Whitespace	6	Braces	21
Line splicing with \	6	Comma	22
Comments	7	Semicolon	22
C comments	7	Colon	23
Nested comments	7	Ellipsis	23
C++ comments	8	Asterisk (pointer declaration)	23
Comment delimiters and whitespace	8	Equal sign (initializer)	23
Tokens	8	Pound sign (preprocessor directive)	24
Keywords	9	Chapter 2 Language structure	25
Identifiers	10	Declarations	25
Naming and length restrictions	10	Objects	25
Identifiers and case sensitivity	10	Lvalues	26
Uniqueness and scope	11	Rvalues	27
Constants	11	Types and storage classes	27
Integer constants	11	Scope	27
Decimal constants	11	Block scope	28
Octal constants	12	Function scope	28
Hexadecimal constants	13	Function prototype scope	28
Long and unsigned suffixes	13	File scope	28
Character constants	14	Class scope (C++)	28
Escape sequences	14	Scope and name spaces	28
Borland C++ special two-character constants	15	Visibility	29
Signed and unsigned char	15	Duration	29
Wide character constants	16	Static duration	29
Floating-point constants	16	Local duration	30
Floating-point constants—data types	16	Dynamic duration	30
Enumeration constants	17	Translation units	31
String literals	18	Linkage	31
		Name mangling	32

Declaration syntax	33	Declarations and definitions	60
Tentative definitions	33	Declarations and prototypes	61
Possible declarations	34	Definitions	63
External declarations and definitions .	36	Formal parameter declarations	64
Type specifiers	38	Function calls and argument	
Type taxonomy	38	conversions	64
Type void	39	Structures	65
The fundamental types	39	Untagged structures and typedefs	66
Integral types	40	Structure member declarations	66
Floating-point types	41	Structures and functions	67
Standard conversions	41	Structure member access	67
Special char, int, and enum		Structure word alignment	69
conversions	42	Structure name spaces	69
Initialization	42	Incomplete declarations	70
Arrays, structures, and unions	43	Bit fields	70
Simple declarations	44	Unions	71
Storage class specifiers	45	Anonymous unions (C++ only)	72
Use of storage class specifier auto ..	45	Union declarations	73
Use of storage class specifier extern .	45	Enumerations	73
Use of storage class specifier		Expressions	75
register	45	Expressions and C++	78
Use of storage class specifier static ..	46	Evaluation order	78
Use of storage class specifier		Errors and overflows	79
typedef	46	Operator semantics	79
Modifiers	47	Operator descriptions	79
The const modifier	47	Unary operators	81
The interrupt function modifier	49	Binary operators	81
The volatile modifier	49	Additive operators	81
The cdecl and pascal modifiers	50	Multiplicative operators	81
pascal	50	Shift operators	81
cdecl	50	Bitwise operators	81
The pointer modifiers	51	Logical operators	81
Function type modifiers	52	Assignment operators	81
Complex declarations and declarators .	53	Relational operators	82
Pointers	54	Equality operators	82
Pointers to objects	55	Component selection operators	82
Pointers to functions	55	Class-member operators	82
Pointer declarations	56	Conditional operator	82
Pointers and constants	57	Comma operator	82
Pointer arithmetic	58	Postfix and prefix operators	82
Pointer conversions	59	Array subscript operator []	82
C++ reference declarations	59	Function call operators ()	83
Arrays	59	Structure/union member operator	
Functions	60	(dot)	83

Structure/union pointer		
operator <code>-></code>	83	
Postfix increment operator <code>++</code>	84	
Postfix decrement operator <code>--</code>	84	
Increment and decrement operators ..	84	
Prefix increment operator	84	
Prefix decrement operator	84	
Unary operators	85	
Address operator <code>&</code>	85	
Indirection operator <code>*</code>	86	
Unary plus operator <code>+</code>	86	
Unary minus operator <code>-</code>	86	
Bitwise complement operator <code>~</code>	86	
Logical negation operator <code>!</code>	86	
The <code>sizeof</code> operator	87	
Multiplicative operators	87	
Additive operators	88	
The addition operator <code>+</code>	88	
The subtraction operator <code>-</code>	89	
Bitwise shift operators	89	
Bitwise shift operators (<code><<</code> and <code>>></code>) ..	89	
Relational operators	90	
The less-than operator <code><</code>	90	
The greater-than operator <code>></code>	91	
The less-than or equal-to operator		
<code><=</code>	91	
The greater-than or equal-to		
operator <code>>=</code>	91	
Equality operators	91	
The equal-to operator <code>==</code>	91	
The inequality operator <code>!=</code>	92	
Bitwise AND operator <code>&</code>	92	
Bitwise exclusive OR operator <code>^</code>	93	
Bitwise inclusive OR operator <code> </code>	93	
Logical AND operator <code>&&</code>	93	
Logical OR operator <code> </code>	94	
Conditional operator <code>?:</code>	94	
Assignment operators	95	
The simple assignment operator <code>=</code> ..	95	
The compound assignment		
operators	96	
Comma operator	96	
C++ operators	97	
Statements	97	
Blocks	98	
Labeled statements	98	
Expression statements	99	
Selection statements	99	
if statements	99	
switch statements	100	
Iteration statements	101	
while statements	101	
do while statements	101	
for statements	102	
Jump statements	103	
break statements	103	
continue statements	103	
goto statements	103	
return statements	104	
Chapter 3 C++ specifics	105	
Referencing	105	
Simple references	106	
Reference arguments	106	
Scope access operator	108	
The new and delete operators	108	
Handling errors	109	
The operator new with arrays	109	
The operator delete with arrays	109	
The <code>::</code> operator new	110	
Initializers with the new operator ...	110	
Classes	111	
Class names	111	
Class types	111	
Class name scope	112	
Class objects	113	
Class member list	113	
Member functions	113	
The keyword <code>this</code>	113	
Inline functions	114	
Static members	115	
Member scope	116	
Nested types	117	
Member access control	118	
Base and derived class access	120	
Virtual base classes	122	
Friends of classes	122	
Constructors and destructors	124	

Constructors	125	Chapter 4 The preprocessor	157
Constructor defaults	126	Null directive #	159
The copy constructor	127	The #define and #undef directives	159
Overloading constructors	127	Simple #define macros	159
Order of calling constructors	128	The #undef directive	160
Class initialization	129	The -D and -U options	161
Destructors	132	The Define option	161
When destructors are invoked	132	Keywords and protected words	162
atexit, #pragma exit, and destructors ..	133	Macros with parameters	162
exit and destructors	133	File inclusion with #include	165
abort and destructors	133	Header file search with	
Virtual destructors	134	<header_name>	166
Overloaded operators	135	Header file search with	
Operator functions	136	"header_name"	166
Overloaded operators and		Conditional compilation	166
inheritance	136	The #if, #elif, #else, and #endif conditional	
Overloading new and delete	137	directives	167
Overloading unary operators	138	The operator defined	167
Overloading binary operators	139	The #ifdef and #ifndef conditional	
Overloading the assignment		directives	168
operator =	139	The #line line control directive	169
Overloading the function call		The #error directive	170
operator ()	140	The #pragma directive	171
Overloading the subscript operator ..	140	#pragma argsused	171
Overloading the class member access		#pragma exit and #pragma startup ..	171
operator	140	#pragma hdrfile	172
Virtual functions	140	#pragma hdrstop	173
Abstract classes	142	#pragma inline	173
C++ scope	143	#pragma intrinsic	173
Class scope	144	#pragma option	173
Hiding	144	#pragma saveregs	175
C++ scoping rules summary	144	#pragma warn	175
Templates	145	Predefined macros	175
Function templates	146	__BCPLUSPLUS__	175
Overriding a template function ..	148	__BORLANDC__	176
Implicit and explicit template		__CDECL__	176
functions	148	__cplusplus	176
Class templates	149	__DATE__	177
Arguments	150	__DLL__	177
Angle brackets	150	__FILE__	177
Type-safe generic lists	151	__LINE__	177
Eliminating pointers	152	__MSDOS__	177
Template compiler switches	152	__OVERLAY__	178
Using template switches	153	__PASCAL__	178

__STDC__	178	Member functions	204
__TCPLUSPLUS__	178	istream	204
__TEMPLATES__	178	ofstream	205
__TIME__	178	Member functions	205
__TURBOC__	179	ostream	205
_Windows	179	Member functions	206
Chapter 5 Using C++ streams	181	ostream_withassign	206
What is a stream?	181	Member functions	206
The istream library	182	ostrstream	206
The streambuf class	182	Member functions	207
The ios class	182	streambuf	207
Output	183	Member functions	207
Fundamental types	184	strstreambase	210
Output formatting	184	Member functions	210
Manipulators	185	strstreambuf	210
Filling and padding	187	Member functions	211
Input	187	strstream	211
I/O of user-defined types	188	Member function	212
Simple file I/O	189	Chapter 6 The container class	
String stream processing	190	libraries	213
Screen output streams	192	What's new since version 2.0?	214
Stream class reference	193	Why two sets of libraries?	215
conbuf	194	Container basics	216
Member functions	194	Object-based and other classes	218
constream	195	Class categories	218
Member functions	195	Non-container classes	218
filebuf	196	Error class	218
Member functions	196	Sortable class	219
fstream	197	Association class	219
Member functions	197	Container classes	219
fstreambase	197	Containers and ownership	220
Member functions	198	Container iterators	222
ifstream	198	Sequence classes	223
Member functions	199	Collections	223
ios	199	Unordered collections	224
Data members	199	Ordered collections	224
Member functions	200	The BIDS template library	224
iostream	202	Templates, classes, and containers	225
iostream_withassign	202	Container implementation	225
Member functions	202	The template solution	226
istream	202	ADTs and FDSs	226
Member functions	203	Class templates	227
istream_withassign	204	Container class compatibility	229

Header files	230	Example	265
Tuning an application	231	Member functions	266
FDS implementation	231	Dictionary	267
ADT implementation	235	Member functions	268
The class library directory	238	DoubleList	268
The INCLUDE directory	238	Member functions	268
The SOURCE directory	239	Friends	270
The LIB directory	239	DoubleListIterator	270
The EXAMPLES directory	240	Member functions	270
Preconditions and checks	240	Error	271
Container class reference	241	Member functions	271
AbstractArray	242	HashTable	272
Data members	242	Member functions	273
Member functions	243	Friends	274
Friends	245	HashTableIterator	274
Array	245	Member functions	274
Example	245	List	275
Member functions	246	Member functions	275
ArrayIterator	247	Friends	276
Member functions	247	ListIterator	276
Association	248	Member functions	276
Member functions	248	MemBlocks	277
Example	249	MemStack	278
Bag	250	Object	279
Member functions	250	Data member	279
BaseDate	252	Member functions	279
Member functions	252	Friends	281
BaseTime	253	Related functions	282
Member functions	253	PriorityQueue	282
Btree	255	Member functions	283
Member functions	255	Queue	284
Friends	257	Example	284
BtreeIterator	257	Member functions	285
Member functions	257	Set	285
Collection	258	Member functions	286
Member functions	259	Sortable	286
Container	259	Member functions	288
Member functions	261	Related functions	288
Friends	263	SortedArray	289
ContainerIterator	263	Stack	289
Member functions	263	Example	290
Date	264	Member functions	291
Member functions	264	String	292
Deque	265	Member functions	292

Example	293
Time	294
Member functions	294
Timer	295
Member functions	295
TShouldDelete	296
Member functions	296

Chapter 7 Converting from Microsoft

C	299
Environment and tools	299
Paths for .h and .LIB files	300
MAKE	301
Command-line compiler	302
Command-line options and libraries ..	306
Linker	306
Source-level compatibility	308
__MSC macro	308
Header files	308
Memory models	309
Keywords	310
Floating-point return values	310
Structures returned by value	310
Conversion hints	311

Chapter 8 Building a Windows application

Compiling and linking within the IDE ..	314
Understanding resource files	315
Understanding module definition files ..	315
Compiling and linking WHELLO ...	315
Using the project manager	316
Setting compile and link options ..	317
WinMain	318
Compiling and linking from the command line ..	318
Compiling from the command line ..	319
Linking from the command line	320
Using a makefile	321
Another makefile for Windows ...	322
Prologs and epilogs	322
Windows All Functions Exportable (-W) ..	323

Windows Explicit Functions Exported (-WE) ..	323
Windows Smart Callbacks (-WS)	323
Windows DLL All Functions Exportable (-WD) ..	324
Windows DLL Explicit Functions Exported (-WDE) ..	324
The _export keyword	324
Prologs, epilogs, and exports: a summary ..	325
Memory models	326
Module definition files	326
A quick example	327
Linking for Windows	328
Linking in the IDE	329
Linking with TLINK	329
Linker options	329
Linking .OBJ and .LIB files	330
Linking .OBJ and .LIB files for DLLs ..	331
Dynamic link libraries	332
Compiling and linking a DLL within the IDE ..	332
Compiling and linking a DLL from the command line ..	332
Module definition files	333
Import libraries	333
Creating DLLs	333
LibMain and WEP	334
Pointers and memory	335
Static data in DLLs	336
C++ classes and pointers	336

Chapter 9 DOS memory management

Running out of memory	339
Memory models	339
The 8086 registers	340
General-purpose registers	340
Segment registers	341
Special-purpose registers	341
The flags register	341
Memory segmentation	342
Address calculation	343

Pointers	344	Using complex math	371
Near pointers	344	Using BCD math	372
Far pointers	344	Converting BCD numbers	373
Huge pointers	345	Number of decimal digits	373
The six memory models	346	Chapter 11 Video functions	375
Mixed-model programming: Addressing		Some words about video modes	375
modifiers	350	Some words about windows and	
Segment pointers	351	viewports	376
Declaring far objects	352	What is a window?	376
Declaring functions to be near or far	352	What is a viewport?	377
Declaring pointers to be near, far, or		Coordinates	377
huge	353	Programming in text mode	377
Pointing to a given segment:offset		The console I/O functions	377
address	355	Text output and manipulation	377
Using library files	355	Window and mode control	379
Linking mixed modules	355	Attribute control	379
Overlays (VROOMM) for DOS	357	State query	380
How overlays work	357	Cursor shape	380
Getting the best out of Borland C++		Text windows	380
overlays	359	An example	381
Requirements	359	The <i>text_modes</i> type	381
Using overlays	360	Text colors	382
Overlay example	360	High-performance output	383
Overlaying in the IDE	361	Programming in graphics mode	384
Overlaid programs	361	The graphics library functions	385
The far call requirement	361	Graphics system control	385
Buffer size	362	A more detailed discussion	387
What not to overlay	362	Drawing and filling	387
Debugging overlays	362	Manipulating the screen and	
External routines in overlays	363	viewport	389
Swapping	364	Text output in graphics mode	390
Expanded memory	364	Color control	392
Extended memory	364	Pixels and palettes	392
Chapter 10 Math	367	Background and drawing color	393
Floating-point options	367	Color control on a CGA	393
Emulating the 80x87 chip	368	CGA low resolution	393
Using 80x87 code	368	CGA high resolution	394
No floating-point code	368	CGA palette routines	395
Fast floating-point option	368	Color control on the EGA and	
The 87 environment variable	369	VGA	395
Registers and the 80x87	370	Error handling in graphics mode ..	395
Disabling floating-point exceptions ..	370	State query	396

Chapter 12 BASM and inline assembly	399	Inline assembly and register variables	404
Inline assembly language	399	Inline assembly, offsets, and size overrides	404
BASM	400	Using C structure members	405
Inline syntax	400	Using jump instructions and labels	406
Opcodes	402	Interrupt functions	406
String instructions	403	Using low-level practices	408
Prefixes	403	Appendix A ANSI implementation-specific standards	411
Jump instructions	403	Index	423
Assembly directives	404		
Inline assembly references to data and functions	404		

T A B L E S

1.1: All Borland C++ keywords	9	2.13: Borland C++ statements	98
1.2: Borland C++ extensions to C	9	4.1: Borland C++ preprocessing directives	
1.3: Keywords specific to C++	9	syntax	158
1.4: Borland C++ register		5.1: Stream manipulators	186
pseudovariables	10	5.2: File modes	190
1.5: Constants—formal definitions	12	5.3: Console stream manipulators	192
1.6: Borland C++ integer constants without		6.1: ADTs as fundamental data	
L or U	13	structures	226
1.7: Borland C++ escape sequences	15	6.2: FDS class templates	227
1.8: Borland C++ floating constant sizes and		6.3: Abbreviations in CLASSLIB names	228
ranges	17	6.4: ADT class templates	228
1.9: Data types, sizes, and ranges	19	6.5: Object-based FDS classes	229
2.1: Borland C++ declaration syntax	35	6.6: Class debugging modes	241
2.2: Borland C++ declarator syntax	36	7.1: CL and BCC options compared	302
2.3: Borland C++ class declarations (C++		7.2: LINK and TLINK options	
only)	37	compared	307
2.4: Declaring types	39	8.1: Compiler options and the <code>_export</code>	
2.5: Integral types	40	keyword	325
2.6: Methods used in standard arithmetic		8.2: Startup and library files for DLLs	331
conversions	42	9.1: Memory models	349
2.7: Borland C++ modifiers	47	9.2: Pointer results	351
2.8: Complex declarations	54	11.1: Graphics mode state query	
2.9: External function definitions	63	functions	397
2.10: Associativity and precedence of		12.1: Opcode mnemonics	402
Borland C++ operators	76	12.2: String instructions	403
2.11: Borland C++ expressions	77	12.3: Jump instructions	404
2.12: Bitwise operators truth table	93	A.1: Identifying diagnostics in C++	411

F I G U R E S

1.1: Internal representations of data types .20	9.3: Tiny model memory segmentation . .347
5.1: Class streambuf and its derived classes182	9.4: Small model memory segmentation .347
5.2: Class ios and its derived classes183	9.5: Medium model memory segmentation348
6.1: Class hierarchies in CLASSLIB217	9.6: Compact model memory segmentation348
6.2: TShouldDelete hierarchy236	9.7: Large model memory segmentation .348
6.3: Class hierarchies in CLASSLIB296	9.8: Huge model memory segmentation .349
8.1: Compiling and linking a Windows program314	9.9: Memory maps for overlays359
9.1: 8086 registers340	11.1: A window in 80×25 text mode381
9.2: Flags register of the 8086342	

I N T R O D U C T I O N

To get an overview of the Borland C++ documentation set, start with the User's Guide. Read the introduction and Chapter 1 in that book for information on how to most effectively use the Borland C++ manuals.

This manual contains materials for the advanced programmer. If you already know how to program well (whether in C, C++, or another language), this manual is for you. It provides a language reference, and programming information on C++ streams, object container classes, converting from Microsoft C, Windows applications, memory models, floating point, overlays, video functions, BASM, inline assembly, and ANSI implementation.



Code examples have a **main** function. EasyWin makes all these examples work in Windows so you don't need **WinMain** and its complicated parameters.

Typefaces and icons used in these books are described in the *User's Guide*.

What's in this book

Chapters 1 through 4: Lexical elements, Language structure, C++ specifics, and The preprocessor, describe the Borland C++ language. Any extensions to the ANSI C standard are noted in these chapters. These chapters provide a formal language definition, reference, and syntax for both the C and C++ aspects of Borland C++. Some overall information for Chapters 1 through 4 is included in the next section of this introduction.

Chapter 5: Using C++ streams tells you how to use the C++ version 2.1 stream library.

Chapter 6: The container class library tells you how to use the Borland C++ object container classes (including templates) in your programs.

Chapter 7: Converting from Microsoft C provides some guidelines on converting your Microsoft C programs to Borland C++.

Chapter 8: Building a Windows application gets you started in Windows programming.

Chapter 9: DOS memory management covers memory models, overlays, and mixed-model programming.

Chapter 10: Math covers floating point and BCD math.

Chapter 11: Video functions is devoted to handling text and graphics in Borland C++.

Chapter 12: BASM and inline assembly tells how to write assembly language programs so they work well when called from Borland C++ programs. It includes information on the built-in assembler in the IDE.

Appendix A: ANSI implementation-specific standards describes those aspects of the ANSI C standard that have been left loosely defined or undefined by ANSI. This appendix tells how Borland C++ operates in respect to each of these aspects.

An introduction to the formal definitions

Chapters 1 through 4 constitute a formal description of the C and C++ languages as implemented in Borland C++. Together, these chapters describe the Borland C++ language; they provide a formal language definition, reference, and syntax for both the C++ and C aspects of Borland C++. These chapters do not provide a language tutorial. We've used a modified Backus-Naur form notation to indicate syntax, supplemented where necessary by brief explanations and program examples. They are organized in this manner:

- Chapter 1, "Lexical elements," shows how the lexical tokens for Borland C++ are categorized. Lexical elements is concerned with the different categories of word-like units, known as *tokens*, recognized by a language.
- Chapter 2, "Language structure," explains how to use the elements of Borland C++. Language structure details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

- Chapter 3, “C++ specifics,” covers those aspects specific to C++.
- Chapter 4, “The preprocessor,” covers the preprocessor, including macros, includes, and pragmas, as well as many other easy yet useful items.

Borland C++ is a full implementation of AT&T’s C++ version 2.1, the object-oriented superset of C developed by Bjarne Stroustrup of AT&T Bell Laboratories. This manual refers to AT&T’s previous version as C++ 2.0. In addition to offering many new features and capabilities, C++ often veers from C by small or large amounts. We’ve made note of these differences throughout these chapters. All the Borland C++ language features derived from C++ are discussed in greater detail in Chapter 3.

Borland C++ also fully implements the ANSI C standard, with several extensions as indicated in the text. You can set options in the compiler to warn you if any such extensions are encountered. You can also set the compiler to treat the Borland C++ extension keywords as normal identifiers (see Chapter 5, “The command-line compiler,” in the *User’s Guide*).

There are also “conforming” extensions provided via the **#pragma** directives offered by ANSI C for handling nonstandard, implementation-dependent features.

Syntax and terminology

Syntactic definitions consist of the name of the nonterminal token or symbol being defined, followed by a colon (:). Alternatives usually follow on separate lines, but a single line of alternatives can be used if prefixed by the phrase “one of.” For example,

external-definition:
function-definition
declaration

octal-digit: one of
 0 1 2 3 4 5 6 7

Optional elements in a construct are printed within angle brackets:

integer-suffix:
unsigned-suffix <*long-suffix*>

Throughout these chapters, the word “argument” is used to mean the actual value passed in a call to a function. “Parameter” is used

to mean the variable defined in the function header to hold the value.

Lexical elements

This chapter provides a formal definition of the Borland C++ lexical elements. It is concerned with the different categories of word-like units, known as *tokens*, recognized by a language. By contrast, language structure (covered in Chapter 2) details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

The tokens in Borland C++ are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor.

A Borland C++ program starts life as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the Borland C++ editor). The basic program unit in Borland C++ is the file. This usually corresponds to a named DOS file located in RAM or on disk and having the extension .C or .CPP.

The preprocessor first scans the program text for special preprocessor *directives* (see page 157). For example, the directive **#include** *<inc_file>* adds (or includes) the contents of the file *inc_file* to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

Whitespace

In the tokenizing phase of compilation, the source code file is *parsed* (that is, broken down) into tokens and *whitespace*. *Whitespace* is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int i; float f;
```

and

```
int i ;  
float f ;
```

are lexically equivalent and parse identically to give the six tokens:

1. **int**
2. **i**
3. **;**
4. **float**
5. **f**
6. **;**

The ASCII characters representing whitespace can occur within *literal strings*, in which case they are protected from the normal parsing process; in other words, they remain as part of the string:

```
char name[] = "Borland International";
```

parses to seven tokens, including the single literal-string token "Borland International".

Line splicing with \

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"Borland \  
International"
```

is parsed as "Borland International" (see page 18, "String literals," for more information).

Comments

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the C method and the C++ method. Both are supported by Borland C++, with an additional, optional extension permitting nested comments. You can mix and match either kind of comment in both C and C++ programs.

C comments

A C comment is any sequence of characters placed after the symbol pair `/*`. The comment terminates at the first occurrence of the pair `*/` following the initial `/*`. The entire sequence, including the four comment delimiter symbols, is replaced by one space *after* macro expansion. Note that some C implementations remove comments without space replacements.

See page 163 for a description of token pasting.

Borland C++ does not support the nonportable *token pasting* strategy using `/**/`. Token pasting in Borland C++ is performed with the ANSI-specified pair `##`, as follows:

```
#define VAR(i,j) (i/**/j) /* won't work */
#define VAR(i,j) (i##j) /* OK in Borland C++ */
#define VAR(i,j) (i ## j) /* Also OK */
```

In Borland C++,

```
int /* declaration */ i /* counter */;
```

parses as

```
int i ;
```

to give the three tokens: **int i ;**

Nested comments

ANSI C doesn't allow nested comments. Attempting to comment out the preceding line with

```
/* int /* declaration */ i /* counter */; */
```

fails, since the scope of the first `/*` ends at the first `*/`. This gives

```
i ; */
```

which would generate a syntax error.

By default, Borland C++ won't allow nested comments, but you can override this with compiler options. You can enable nested comments via the Source Options dialog box (O|C|Source) in the IDE or with the `-C` option (for the command-line compiler).

C++ comments

You can also use // to create comments in C code. This is specific to Borland C++.

C++ allows a single-line comment using two adjacent slashes (`//`). The comment can start in any position, and extends until the next new line:

```
class X { // this is a comment
... };
```

Comment delimiters and whitespace

In rare cases, some whitespace before `/*` and `//`, and after `*/`, although not syntactically mandatory, can avoid portability problems. For example, this C++ code

```
int i = j///* divide by k*/k;
+m;
```

parses as `int i = j +m;` not as

```
int i = j/k;
+m;
```

as expected under the C convention. The more legible

```
int i = j/ /* divide by k*/ k;
+m;
```

avoids this problem.

Tokens

Borland C++ recognizes six classes of tokens. The formal definition of a token is as follows:

token:
keyword
identifier
constant
string-literal
operator
punctuator

Punctuators are also known as separators.

As the source code is parsed, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, **external** would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier *al*.

Keywords

Keywords are words reserved for special purposes and must not be used as normal identifier names. The following two tables list the Borland C++ keywords. You can use options in the IDE (or command-line compiler options) to select ANSI keywords only, UNIX keywords, and so on; see Chapter 2, "IDE Basics" and Chapter 5, "The command-line compiler," in the *User's Guide*, for information on these options.

Table 1.1
All Borland C++ keywords

_asm	_ds	int	_seg
asm	else	_interrupt	short
auto	enum	interrupt	signed
break	_es	_loadds	sizeof
case	_export	long	_ss
_cdecl	extern	_near	static
cdecl	_far	near	struct
char	far	new	switch
class	_fastcall	operator	template
const	float	_pascal	this
continue	for	pascal	typedef
_cs	friend	private	union
default	goto	protected	unsigned
delete	_huge	public	virtual
do	huge	register	void
double	if	return	volatile
	inline	_saveregs	while

Table 1.2
Borland C++ extensions to C

_cdecl	_es	huge	_pascal
cdecl	_export	interrupt	pascal
_cs	_far	_loadds	_saveregs
_ds	far	_near	_seg
	_fastcall	near	_ss

Table 1.3
Keywords specific to C++

asm	operator
class	private
delete	protected
friend	public
inline	template
new	this
	virtual

Table 1.4
Borland C++ register
pseudovariables

_AH	_BP	_CX	_DX
_AL	_BX	_DH	_ES
_AX	_CH	_DI	_FLAGS
_BH	_CL	_DL	_SI
_BL	_CS	_DS	_SP
			_SS

Identifiers

The formal definition of an identifier is as follows:

identifier:

nondigit

identifier nondigit

identifier digit

nondigit: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z _

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

Naming and length restrictions

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. Identifiers can contain the letters *A* to *Z* and *a* to *z*, the underscore character (`_`), and the digits 0 to 9. There are only two restrictions:

1. The first character must be a letter or an underscore.
2. By default, Borland C++ recognizes only the first 32 characters as significant. The number of significant characters can be *reduced* by menu and command-line options, but not increased. Use the `-in` command-line option (where $1 \leq n \leq 32$) or Identifier Length in the Source Options dialog box (O|C|Source).

Identifiers in C++ programs are significant to any length.

Identifiers and case sensitivity

Borland C++ identifiers are case sensitive, so that *Sum*, *sum*, and *suM* are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, Borland C++ offers the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. By checking Case-sensitive Link in the Linker dialog box

(Options | Linker | Settings), or using the `/c` command-line switch with TLINK, you can ensure that global identifiers are *case insensitive*. Under this regime, the globals `Sum` and `sum` are considered identical, resulting in a possible “Duplicate symbol” warning during linking.

An exception to these rules is that identifiers of type **pascal** are always converted to all uppercase for linking purposes.

Uniqueness and scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same *scope* and sharing the same *name space*. Duplicate names are always legal for *different* name spaces regardless of scope. The rules are covered in the discussion on scope starting on page 27.

Constants

Constants are tokens representing fixed numeric or character values. Borland C++ supports four classes of constants: floating point, integer, enumeration, and character.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in Table 1.5.

Integer constants

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Table 1.6. Note that the rules vary between decimal and nondecimal constants.

Decimal constants

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit will be truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10; /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0; /*decimal 0 = octal 0 */
```

Table 1.5: Constants—formal definitions

<i>constant:</i>	0 X hexadecimal-digit hexadecimal-constant hexadecimal-digit
<i>floating-constant</i>	
<i>integer-constant</i>	
<i>enumeration-constant</i>	<i>nonzero-digit:</i> one of 1 2 3 4 5 6 7 8 9
<i>character-constant</i>	
<i>floating-constant:</i>	<i>octal-digit:</i> one of 0 1 2 3 4 5 6 7
<i>fractional-constant</i> <exponent-part> <floating-suffix>	<i>hexadecimal-digit:</i> one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
<i>digit-sequence</i> <i>exponent-part</i> <floating-suffix>	<i>integer-suffix:</i> <i>unsigned-suffix</i> <long-suffix> <i>long-suffix</i> <unsigned-suffix>
<i>fractional-constant:</i>	<i>unsigned-suffix:</i> one of u U
<digit-sequence> . <i>digit-sequence</i>	<i>long-suffix:</i> one of l L
<i>digit-sequence</i> .	<i>enumeration-constant:</i> <i>identifier</i>
<i>exponent-part:</i>	<i>character-constant:</i> <i>c-char-sequence</i>
e <sign> <i>digit-sequence</i>	<i>c-char-sequence:</i> <i>c-char</i> <i>c-char-sequence</i> <i>c-char</i>
E <sign> <i>digit-sequence</i>	<i>c-char:</i> Any character in the source character set except the single-quote ('), backslash (\), or newline character <i>escape-sequence</i> .
<i>sign:</i> one of	<i>escape-sequence:</i> one of
+ -	\ <i>"</i> \ <i>'</i> \ <i>?</i> \ <i>\</i>
<i>digit-sequence:</i>	\ <i>a</i> \ <i>b</i> \ <i>f</i> \ <i>n</i>
<i>digit</i>	\ <i>o</i> \ <i>oo</i> \ <i>ooo</i> \ <i>r</i>
<i>digit-sequence</i> <i>digit</i>	\ <i>t</i> \ <i>v</i> \ <i>Xh...</i> \ <i>xh...</i>
<i>floating-suffix:</i> one of	
f l F L	
<i>integer-constant:</i>	
<i>decimal-constant</i> < <i>integer-suffix</i> >	
<i>octal-constant</i> < <i>integer-suffix</i> >	
<i>hexadecimal-constant</i> < <i>integer-suffix</i> >	
<i>decimal-constant:</i>	
<i>nonzero-digit</i>	
<i>decimal-constant</i> <i>digit</i>	
<i>octal-constant:</i>	
0	
<i>octal-constant</i> <i>octal-digit</i>	
<i>hexadecimal-constant:</i>	
0 x <i>hexadecimal-digit</i>	

Octal constants

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 will be truncated.

Hexadecimal constants

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF will be truncated.

Long and unsigned suffixes

The suffix *L* (or *l*) attached to any constant forces it to be represented as a **long**. Similarly, the suffix *U* (or *u*) forces the constant to be **unsigned**. It is **unsigned long** if the value of the number itself is greater than decimal 65,535, regardless of which base is used. You can use both *L* and *U* suffixes on the same constant in any order or case: *ul*, *lu*, *UL*, and so on.

Table 1.6
Borland C++ integer
constants without L or U

Decimal constants	
0 to 32,767	int
32,768 to 2,147,483,647	long
2,147,483,648 to 4,294,967,295	unsigned long
> 4294967295	truncated
Octal constants	
00 to 077777	int
0100000 to 0177777	unsigned int
02000000 to 01777777777	long
020000000000 to 03777777777	unsigned long
> 03777777777	truncated
Hexadecimal constants	
0x0000 to 0x7FFF	int
0x8000 to 0xFFFF	unsigned int
0x10000 to 0x7FFFFFFF	long
0x80000000 to 0xFFFFFFFF	unsigned long
> 0xFFFFFFFF	truncated

The data type of a constant in the absence of any suffix (*U*, *u*, *L*, or *l*) is the first of the following types that can accommodate its value:

decimal	int, long int, unsigned long int
octal	int, unsigned int, long int, unsigned long int
hexadecimal	int, unsigned int, long int, unsigned long int

If the constant has a *U* or *u* suffix, its data type will be the first of **unsigned int**, **unsigned long int** that can accommodate its value.

If the constant has an *L* or *l* suffix, its data type will be the first of **long int**, **unsigned long int** that can accommodate its value.

If the constant has both *u* and *l* suffixes (*ul*, *lu*, *Ul*, *lU*, *uL*, *Lu*, *LU*, or *UL*), its data type will be **unsigned long int**.

Table 1.6 summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding *L* or *U* suffix has been used.

Character constants A *character constant* is one or more characters enclosed in single quotes, such as 'A', '=', '\n'. In C, single character constants have data type **int**; they are represented internally with 16 bits, with the upper byte zero or sign-extended. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

Escape sequences

The backslash character (\) is used to introduce an *escape sequence*, allowing the visual representation of certain nongraphic characters. For example, the constant `\n` is used for the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, `'\03'` for *Ctrl-C* or `'\x3F'` for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff for Borland C++). Larger numbers generate the compiler error, "Numeric constant too large." For example, the octal number `\777` is larger than the maximum value allowed, `\377`, and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Originally, Turbo C allowed only three digits in a hexadecimal escape sequence. The ANSI C rules adopted in Borland C++ might cause problems with old code that assumes only the first three characters are converted. For example, using Turbo C 1.x to define a string with a bell (ASCII 7) followed by numeric characters, a programmer might write:

```
printf("\x007.1A Simple Operating System");
```

This is intended to be interpreted as `\x007` and “2.1A Simple Operating System”. However, Borland C++ compiles it as the hexadecimal number `\x0072` and the literal string “.1A Simple Operating System”.

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities may also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant `\258` would be interpreted as a two-character constant made up of the characters `\25` and `8`.

The next table shows the available escape sequences.

Table 1.7
Borland C++ escape
sequences

*The \\ must be used to
represent a real ASCII
backslash, as used in DOS
paths.*

Sequence	Value	Char	What it does
<code>\a</code>	0x07	BEL	Audible bell
<code>\b</code>	0x08	BS	Backspace
<code>\f</code>	0x0C	FF	Formfeed
<code>\n</code>	0x0A	LF	Newline (linefeed)
<code>\r</code>	0x0D	CR	Carriage return
<code>\t</code>	0x09	HT	Tab (horizontal)
<code>\v</code>	0x0B	VT	Vertical tab
<code>\\</code>	0x5c	<code>\</code>	Backslash
<code>\'</code>	0x27	<code>'</code>	Single quote (apostrophe)
<code>\"</code>	0x22	<code>"</code>	Double quote
<code>\?</code>	0x3F	<code>?</code>	Question mark
<code>\O</code>		any	O = a string of up to three octal digits
<code>\xH</code>		any	H = a string of hex digits
<code>\XH</code>		any	H = a string of hex digits

Borland C++ special two-character constants

Borland C++ also supports two-character constants (for example, `'An'`, `'\n\t'`, and `'\007\007'`). These constants are represented as 16-bit `int` values, with the first character in the low-order byte and the second character in the high-order byte. These constants are not portable to other C compilers.

Signed and unsigned char

In C, one-character constants, such as `'A'`, `'\t'`, and `'\007'`, are also represented as 16-bit `int` values. In this case, the low-order byte is *sign extended* into the high byte; that is, if the value is greater than 127 (base 10), the upper byte is set to `-1` (`=0xFF`). This

can be disabled by declaring that the default **char** type is **unsigned** (use the `-K` command-line compiler option or choose Unsigned Characters in the Options | Compiler | Code Generation dialog box), which forces the high byte to be zero regardless of the value of the low byte.

Wide character constants

A character constant preceded by an *L* is a wide-character constant of data type **wchar_t** (an integral type defined in `stddef.h`). For example,

```
x = L 'A';
```

Floating-point constants

A floating constant consists of:

- decimal integer
- decimal point
- decimal fraction
- *e* or *E* and a signed integer exponent (optional)
- type suffix: *f* or *F* or *l* or *L* (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter *e* (or *E*) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Examples:

Constant	Value
23.45e6	23.45×10^6
.0	0
0.	0
1.	$1.0 \times 10^0 = 1.0$
-1.23	-1.23
2e-5	2.0×10^{-5}
3E+10	3.0×10^{10}
.09E34	0.09×10^{34}

Floating-point constants—data types

In the absence of any suffixes, floating-point constants are of type **double**. However, you can coerce a floating constant to be of type

float by adding an *f* or *F* suffix to the constant. Similarly, the suffix *l* or *L* forces the constant to be data type **long double**. The next table shows the ranges available for **float**, **double**, and **long double**.

Table 1.8
Borland C++ floating
constant sizes and ranges

Type	Size (bits)	Range
float	32	3.4×10^{-38} to 3.4×10^{38}
double	64	1.7×10^{-308} to 1.7×10^{308}
long double	80	3.4×10^{-4932} to 1.1×10^{4932}

Enumeration constants

Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration. Negative initializers are allowed.

See page 73 for a detailed look at **enum** declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional *initializers*. In this example,

```
enum team { giants, cubs, dodgers };
```

giants, **cubs**, and **dodgers** are enumeration constants of type **team** that can be assigned to any variables of type **team** or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

String literals String literals, also known as string constants, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type **array of char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written "".

The characters inside the double quotes can include escape sequences (see page 14). This code, for example,

```
"\t\t\"Name\"\\tAddress\n\n"
```

prints out like this:

```
        "Name" \        Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The \ provides interior double quotes.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single '\0' character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

```
#include <stdio.h>

int main()
{
    char    *p;

    p = "This is an example of how Borland C++"
        " will automatically\ndo the concatenation for"
        " you on very long strings,\nresulting in nicer"
        " looking programs.";
    printf(p);
    return(0);
}
```

The output of the program is

```
This is an example of how Borland C++ will automatically
do the concatenation for you on very long strings,
resulting in nicer looking programs.
```

You can also use the backslash (\) as a continuation character in order to extend a string constant across line boundaries:

```
puts("This is really \
a one-line string");
```

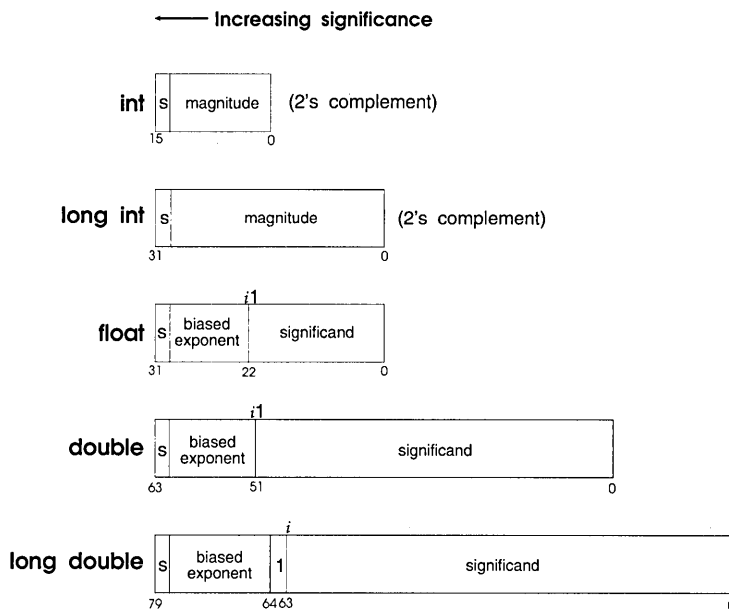
Constants and internal representation

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation specific and usually derive from the architecture of the host computer. For Borland C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80x86 microprocessors governs the choices of inner representations for the various data types. The next table lists the sizes and resulting ranges of the data types for Borland C++; see page 39 for more information on these data types. Figure 1.1 shows how these types are represented internally.

Table 1.9: Data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
enum	16	-32,768 to 32,767	Ordered sets of values
unsigned int	16	0 to 65,535	Larger numbers and loops
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4×10^{-38} to 3.4×10^{38}	Scientific (7-digit precision)
double	64	1.7×10^{-308} to 1.7×10^{308}	Scientific (15-digit precision)
long double	80	3.4×10^{-4932} to 1.1×10^{4932}	Financial (19-digit precision)
near pointer	16	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

Figure 1.1
Internal representations of
data types



s = Sign bit (0 = positive, 1 = negative)

i = Position of implicit binary point

1 = Integer bit of significand:

Stored in **long double**

Implicit (always 1) in **float**, **double**

Exponent bias (normalized values):

float : 127 (7FH)

double : 1023 (3FFH)

long double : 16,383 (3FFFH)

Constant expressions

A constant expression is an expression that always evaluates to a constant (and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is

constant-expression:

Conditional-expression

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a **sizeof** operator:

- assignment
- comma
- decrement
- function call
- increment

Punctuators

The punctuators (also known as separators) in Borland C++ are defined as follows:

punctuator: one of

[] () { } , ; : ... * = #

Brackets [] (open and close brackets) indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];          /* 3 x 4 matrix */
ch = str[3];           /* 4th element */
...
```

Parentheses () (open and close parentheses) group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);      /* override normal precedence */
if (d == z) ++x;     /* essential with conditional statement */

func();              /* function call, no args */
int (*fptr)();      /* function pointer declaration */
fptr = func;        /* no () means func pointer */

void func2(int n);  /* function declaration with args */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered on page 79.

Braces `{ }` (open and close braces) indicate the start and end of a compound statement:

```
if (d == z)
{
    ++x;
    func();
}
```

The closing brace serves as a terminator for the compound statement, so a `;` (semicolon) is not required after the `}`, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
    {};           /*illegal semicolon*/
else
```

Comma The comma `(,)` separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in *comma expressions*. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them:

```
func(i, j);           /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5)); /* also calls func
                                         with two args! */
```

Semicolon The semicolon `(;)` is a statement terminator. Any legal C or C++ expression (including the empty expression) followed by `;` is interpreted as a statement, known as an *expression statement*. The expression is evaluated and its value is discarded. If the expression statement has no side effects, Borland C++ may ignore it.

```
a + b;    /* maybe evaluate a + b, but discard value */
++a;     /* side effect on a, but discard value of ++a */
;        /* empty expression = null statement */
```

Semicolons are often used to create an *empty statement*:

```
for (i = 0; i < n; i++)
{
    ;
}
```

Colon Use the colon (:) to indicate a labeled statement:

```
start:   x=0;
...
goto start;
...
switch (a) {
    case 1: puts("One");
           break;
    case 2: puts("Two");
           break;
    ...
default:  puts("None of the above!");
           break;
}
```

Labels are covered on page 98.

Ellipsis Ellipsis (...) are three successive periods with no whitespace intervening. Ellipsis are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that **func** will be defined in such a way that calls must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.



In C++, you can omit the comma preceding the ellipsis.

Asterisk (pointer declaration)

The * (asterisk) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr; /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;           /* a pointer to a pointer to an int */
double ***double_ptr;   /* a pointer to a pointer to a pointer
                        to doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;
a = b * 3.14;
```

Equal sign (initializer) The = (equal sign) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };  
int x = 5;
```

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C function, no code can precede any variable declarations.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... } /* parameter i has default value of  
zero */
```

The equal sign is also used as the assignment operator in expressions:

```
a = b + c;  
ptr = farmalloc(sizeof(float)*100);
```

Pound sign (preprocessor directive) The # (pound sign) indicates a preprocessor directive when it occurs as the first nonwhitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See page 157 for more on the preprocessor directives.

and ## (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase.

Language structure

This chapter provides a formal definition of Borland C++'s language structure. It details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units. By contrast, lexical elements (described in Chapter 1) are concerned with the different categories of word-like units, known as tokens, recognized by a language.

Declarations

Scope is discussed starting on page 27; visibility on page 29; duration on page 29; and linkage on page 31.

This section briefly reviews concepts related to declarations: objects, types, storage classes, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.

Objects

An *object* is an identifiable region of memory that can hold a fixed or variable value (or set of values). (This use of the word *object* is not to be confused with the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a *data type*). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely "points" to the object. The type is used

- to determine the correct memory allocation required initially
- to interpret the bit patterns found in the object during subsequent accesses
- in many type-checking situations, to ensure that illegal assignments are trapped

Borland C++ supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in various memory models.

The Borland C++ standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that Borland C++ can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as *defining declarations*, also establish the creation (where and when) of the object, that is, the allocation of physical memory and its possible initialization. Other declarations, known as *referencing declarations*, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its *declaration point* in the source code. Legal exceptions to this rule, known as *forward references*, are labels, calls to undeclared functions, and class, struct, or union tags.

Lvalues

An *lvalue* is an object locator: An expression that designates an object. An example of an lvalue expression is **P*, where *P* is any expression evaluating to a nonnull pointer. A *modifiable lvalue* is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A **const** pointer to a constant, for example, is *not* a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the *l* stood for “left,” meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if *a* and *b* are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as $a = 1$; and $b = a + b$ are legal.

Rvalues The expression $a + b$ is not an lvalue: $a + b = a$ is illegal because the expression on the left is not related to an object. Such expressions are often called *rvalues* (short for right values).

Types and storage classes

Associating identifiers with objects requires that each identifier has at least two attributes: *storage class* and *type* (sometimes referred to as data type). The Borland C++ compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location (data segment, register, heap, or stack) of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type, as explained earlier, determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object’s storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The special compile-time operator, **sizeof**, lets you determine the size in bytes of any standard or user-defined type; see page 87 for more on this operator.

Scope

The *scope* of an identifier is that part of the program in which the identifier can be used to access its object. There are five categories of scope: *block* (or *local*), *function*, *function prototype*, *file*, and *class* (C++ only). These depend on how and where identifiers are declared.

Block scope The *scope* of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the *enclosing* block). Parameter declarations with a function definition also have block scope, limited to the scope of the block that defines the function.

Function scope The only identifiers having function scope are statement labels. Label names can be used with **goto** statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing *label_name*: followed by a statement. Label names must be unique within a function.

Function prototype scope Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.

File scope File scope identifiers, also known as *globals*, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.

Class scope (C++) For now, think of a class as a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class. Classes and their objects have many special access and scoping rules; see pages 111 to 124.

Scope and name spaces *Name space* is the scope within which an identifier must be unique. There are four distinct classes of identifiers in C:

1. **goto** label names. These must be unique within the function in which they are declared.
2. Structure, union, and enumeration tags. These must be unique within the block in which they are defined. Tags declared outside of any function must be unique within all tags defined externally.
3. Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.

Structures, classes, and enumerations are in the same name space in C++.

4. Variables, **typedefs**, functions, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Visibility

The *visibility* of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily *hidden* by the appearance of a duplicate identifier: The object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Visibility cannot exceed scope, but scope can exceed visibility.

```
...
{
    int i; char ch; // auto by default
    i = 3;         // int i and char ch in scope and visible
...
{
    double i;
    i = 3.0e3;    // double i in scope and visible
                 // int i=3 in scope but hidden
    ch = 'A';    // char ch in scope and visible
}
// double i out of scope
i += 1;        // int i visible and = 4
...           // char ch still in scope & visible = 'A'
}
...           // int i and char ch out of scope
```



Again, special rules apply to hidden class names and class member names: Special C++ operators allow hidden identifiers to be accessed under certain conditions (see page 112).

Duration

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedefs** and types, have real memory allocated during run time. There are three kinds of duration: *static*, *local*, and *dynamic*.

Static duration Objects with *static* duration are allocated memory as soon as execution is underway; this storage allocation lasts until the program terminates. Static duration objects usually reside in fixed data segments allocated according to the memory model in force. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, constructor.

Static duration must not be confused with file or global scope. An object can have static duration and local scope.

Local duration *Local* duration objects, also known as *automatic* objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects always must have local or function scope. The storage class specifier **auto** may be used when declaring local duration variables, but is usually redundant, since **auto** is the default for variables declared within a block.

An object with local duration also has local scope, since it does not exist outside of its enclosing block. The converse is not true: A local scope object can have static duration.

When declaring variables (for example, **int**, **char**, **float**), the storage class specifier **register** also implies **auto**; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. Borland C++ can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an **auto**, local object with no warning or error.

Dynamic duration *Dynamic* duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the *heap*, using either standard library functions such as **malloc**, or by using the C++ operator **new**. The corresponding deallocations are made using **free** or **delete**.

Translation units

The term *translation unit* refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

```
translation-unit:  
    external-declaration  
    translation-unit external-declaration  
  
external-declaration  
    function-definition  
    declaration
```

For more details, see
"External declarations and
definitions" on page 36.

The word *external* has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the following section, "Linkage.") Any declaration that also reserves storage for an object or function is called a definition (or defining declaration).

Linkage

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier **static** or **extern**.

Each instance of a particular identifier with *external linkage* represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with *internal linkage* represents the same object or function only within one file. Identifiers with *no linkage* represent unique entities.

External and internal linkage rules are as follows:

1. Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**.
For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.
2. If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.
3. If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.
4. If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

The following identifiers have no linkage attribute:

1. any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
2. function parameters
3. block scope identifiers for objects declared without the storage class specifier **extern**

Name mangling

When a C++ module is compiled, the compiler generates function names that include an encoding of the function's argument types. This is known as name mangling. It makes overloaded functions possible, and helps the linker catch errors in calls to functions in other modules. However, there are times when you won't want name mangling. When compiling a C++ module to be linked with a module that does not have mangled names, the C++ compiler has to be told not to mangle the names of the functions from the other module. This situation typically arises when linking with libraries or .OBJ files compiled with a C compiler.

To tell the C++ compiler not to mangle the name of a function, simply declare the function as `extern "C"`, like this:

```
extern "C" void Cfunc( int );
```

This declaration tells the compiler that references to the function **Cfunc** should not be mangled.

You can also apply the `extern "C"` declaration to a block of names:

```
extern "C" {  
    void Cfunc1( int );  
    void Cfunc2( int );  
    void Cfunc3( int );  
};
```

As with the declaration for a single function, this declaration tells the compiler that references to the functions **Cfunc1**, **Cfunc2**, and **Cfunc3** should not be mangled. You can also use this form of block declaration when the block of function names is contained in a header file:

```
extern "C" {  
    #include "locallib.h"  
};
```

Declaration syntax

All six interrelated attributes (storage class, type, scope, visibility, duration, and linkage) are determined in diverse ways by *declarations*.

Declarations can be *defining declarations* (also known simply as *definitions*) or *referencing declarations* (sometimes known as *nondefining declarations*). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration simply introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Tentative definitions

The ANSI C standard introduces a new concept: that of the *tentative definition*. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;  
int x;          /*legal, one copy of x is reserved */  
  
int y;  
int y = 4;     /* legal, y is initialized to 4 */  
  
int z = 5;  
int z = 6;     /* not legal, both are initialized definitions */
```



Unlike ANSI C, C++ doesn't have the concept of a tentative declaration; an external data declaration without a storage class specifier is always a definition.

Possible declarations

The range of objects that can be declared includes

- variables
- functions
- classes and class members (C++)
- types
- structure, union, and enumeration tags
- structure members
- union members
- arrays of other types
- enumeration constants
- statement labels
- preprocessor macros

The full syntax for declarations is shown in the following tables. The recursive nature of the declarator syntax allows complex declarators. We encourage the use of **typedefs** to improve legibility.

Table 2.1
Borland C++ declaration
syntax

<i>declaration:</i> <decl-specifiers> <declarator-list>; <i>asm-declaration</i> <i>function-declaration</i> <i>linkage-specification</i>	int long signed unsigned float double void
<i>decl-specifier:</i> <i>storage-class-specifier</i> <i>type-specifier</i> <i>fct-specifier</i> friend (C++ specific) typedef	<i>elaborated-type-specifier:</i> <i>class-key</i> <i>identifier</i> <i>class-key</i> <i>class-name</i> enum <i>enum-name</i>
<i>decl-specifiers:</i> <decl-specifiers> <i>decl-specifier</i>	<i>class-key:</i> (C++ specific) class struct union
<i>storage-class-specifier:</i> auto register static extern	<i>enum-specifier:</i> enum <identifier> { <enum-list> }
<i>fct-specifier:</i> (C++ specific) inline virtual	<i>enum-list:</i> <i>enumerator</i> <i>enumerator-list</i> , <i>enumerator</i>
<i>type-specifier:</i> <i>simple-type-name</i> <i>class-specifier</i> <i>enum-specifier</i> <i>elaborated-type-specifier</i> const volatile	<i>enumerator:</i> <i>identifier</i> <i>identifier</i> = <i>constant-expression</i>
<i>simple-type-name:</i> <i>class-name</i> typedef-name char short	<i>constant-expression:</i> <i>conditional-expression</i> <i>linkage-specification:</i> (C++ specific) extern <i>string</i> { <declaration-list> } extern <i>string</i> <i>declaration</i>
	<i>declaration-list:</i> <i>declaration</i> <i>declaration-list</i> ; <i>declaration</i>

For the following table, note that there are restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail starting on page 47.

Table 2.2: Borland C++ declarator syntax

<i>declarator-list:</i> init-declarator declarator-list , init-declarator	class-name (C++ specific) ~ class-name (C++ specific) typedef -name
init-declarator: declarator <initializer>	type-name: type-specifier <abstract-declarator>
declarator: dname modifier-list ptr-operator declarator declarator (parameter-declaration-list) <cv-qualifier-list> (The <cv-qualifier-list> is for C++ only.) declarator [<constant-expression>] (declarator)	abstract-declarator: ptr-operator <abstract-declarator> <abstract-declarator> (argument-declaration-list) <cv-qualifier-list> <abstract-declarator> [<constant-expression>] (abstract-declarator)
modifier-list: modifier modifier-list modifier	argument-declaration-list: <arg-declaration-list> arg-declaration-list , ... <arg-declaration-list> ... (C++ specific)
modifier: cdecl pascal interrupt near far huge	arg-declaration-list: argument-declaration arg-declaration-list , argument-declaration
ptr-operator: * <cv-qualifier-list> & <cv-qualifier-list> (C++ specific) class-name :: * <cv-qualifier-list> (C++ specific)	argument-declaration: decl-specifiers declarator decl-specifiers declarator = expression (C++ specific) decl-specifiers <abstract-declarator> decl-specifiers <abstract-declarator> = expression (C++ specific)
cv-qualifier-list: cv-qualifier <cv-qualifier-list>	fcn-definition: <decl-specifiers> declarator <ctor-initializer> fcn-body
cv-qualifier const volatile	fcn-body: compound-statement
dname: name	initializer: = expression = { initializer-list } (expression-list) (C++ specific)
	initializer-list: expression initializer-list , expression { initializer-list <, > }

External declarations and definitions

The storage class specifiers **auto** and **register** cannot appear in an external declaration (see "Translation units," page 31). For each identifier in a translation unit declared with internal linkage, there can be no more than one external definition.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of **sizeof**), there must be exactly one external definition of that identifier somewhere in the entire program.

Borland C++ allows later re-declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. For example,

```
int a[];           // no size
struct mystruct; // tag only, no member declarators
...
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
    int i, j;
};                // add member declarators
```

The following table covers class declaration syntax. Page 105 covers C++ reference types (closely related to pointer types) in detail.

Table 2.3: Borland C++ class declarations (C++ only)

<i>class-specifier:</i> class-head { <member-list> }	<i>access-specifier</i> <virtual> class-name
<i>class-head:</i> class-key <identifier> <base-spec> class-key class-name <base-spec>	<i>access-specifier:</i> private protected public
<i>member-list:</i> member-declaration <member-list> access-specifier : <member-list>	<i>conversion-function-name:</i> operator conversion-type-name
<i>member-declaration:</i> <decl-specifiers> <member-declarator-list> ; function-definition <;> qualified-name ;	<i>conversion-type-name:</i> type-specifiers <ptr-operator>
<i>member-declarator-list:</i> member-declarator member-declarator-list, member-declarator	<i>ctor-initializer:</i> : mem-initializer-list
<i>member-declarator:</i> declarator <pure-specifier> <identifier> : constant-expression	<i>mem-initializer-list:</i> mem-initializer mem-initializer , mem-initializer-list
<i>pure-specifier:</i> = 0	<i>mem-initializer:</i> class name (<argument-list>) identifier (<argument-list>)
<i>base-spec:</i> : base-list	<i>operator-function-name:</i> operator operator
<i>base-list:</i> base-specifier base-list , base-specifier	<i>operator:</i> one of new delete sizeof
<i>base-specifier:</i> class-name virtual <access-specifier> class-name	+ - * / % ^ & ~ ! = <> += -= *= /= %= ^= &= = << >> >>= <<<= == != <= >= && ++ -- , ->* -> () [] .*

Type specifiers

The *type specifier* with one or more optional *modifiers* is used to specify the type of the declared identifier:

```
int i; // declare i as a signed integer
unsigned char ch1, ch2; // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type **signed int** (or equivalently, **int**) is the assumed default. However, in C++ there are some situations where a missing type specifier leads to syntactic ambiguity, so C++ practice uses the explicit entry of all **int** type specifiers.

Type taxonomy

There are four basic type categories: *void*, *scalar*, *function*, and *aggregate*. The scalar and aggregate types can be further divided as follows:

- ▣ Scalar: arithmetic, enumeration, pointer, and reference types (C++)
- ▣ Aggregate: array, structure, union, and class types (C++)

Types can also be divided into *fundamental* and *derived* types. The fundamental types are **void**, **char**, **int**, **float**, and **double**, together with **short**, **long**, **signed**, and **unsigned** variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.



A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes.

Given any nonvoid type **type** (with some provisos), you can declare derived types as follows:

Table 2.4
Declaring types

Note that `type&`, `var`, `type &var`, and `type & var` are all equivalent.

<code>type t;</code>	An object of type <code>type</code>
<code>type array[10];</code>	Ten <code>types</code> : <code>array[0] – array[9]</code>
<code>type *ptr;</code>	<code>ptr</code> is a pointer to <code>type</code>
<code>type &ref = t;</code>	<code>ref</code> is a reference to <code>type</code> (C++)
<code>type func(void);</code>	<code>func</code> returns value of type <code>type</code>
<code>void func1(type t);</code>	<code>func1</code> takes a type <code>type</code> parameter
<code>struct st {type t1; type t2};</code>	structure <code>st</code> holds two <code>types</code>

And here's how you could declare derived types in a class:

```
class ct { // class ct holds ptr to type plus a function
        // taking a type parameter (C++)
    type *ptr;
    public:
    void func(type*);
}
```

Type void

void is a special type specifier indicating the absence of any values. It is used in the following situations:

*C++ handles **func()** in a special manner. See "Declarations and prototypes" on page 61 and code examples on page 62.*

- An empty parameter list in a function declaration:

```
int func(void); // func takes no arguments
```

- When the declared function does not return a value:

```
void func(int n); // return value
```

- As a generic pointer: A pointer to **void** is a generic pointer to anything:

```
void *ptr; // ptr can later be set to point to any object
```

- In *typecasting* expressions:

```
extern int errfunc(); // returns an error code
```

```
...
```

```
(void) errfunc(); // discard return value
```

The fundamental types

signed and unsigned are modifiers that can be applied to the integral types.

The fundamental type specifiers are built from the following keywords:

char	int	signed
double	long	unsigned
float	short	

From these keywords, you can build the integral and floating-point types, which are together known as the *arithmetic* types. The include file `limits.h` contains definitions of the value ranges for all the fundamental types.

Integral types **char**, **short**, **int**, and **long**, together with their unsigned variants, are all considered *integral* data types. The integral type specifiers are as follows, with synonyms listed on the same line:

Table 2.5
Integral types

char, signed char unsigned char	Synonyms if default char set to signed
char, unsigned char signed char	Synonyms if default char set to unsigned
int, signed int unsigned, unsigned int	
short, short int, signed short int unsigned short, unsigned short int	
long, long int, signed long int unsigned long, unsigned long int	

At most, one of **signed** and **unsigned** can be used with **char**, **short**, **int**, or **long**. If you use the keywords **signed** and **unsigned** on their own, they mean **signed int** and **unsigned int**, respectively.

In the absence of **unsigned**, **signed** is usually assumed. An exception arises with **char**. Borland C++ lets you set the default for **char** to be **signed** or **unsigned**. (The default, if you don't set it yourself, is **signed**.) If the default is set to **unsigned**, then the declaration `char ch` declares `ch` as **unsigned**. You would need to use `signed char ch` to override the default. Similarly, with a **signed** default for **char**, you would need an explicit `unsigned char ch` to declare an **unsigned char**.

At most, one of **long** and **short** can be used with **int**. The keywords **long** and **short** used on their own mean **long int** and **short int**.

ANSI C does not dictate the sizes or internal representations of these types, except to insist that **short**, **int**, and **long** form a non-decreasing sequence with "**short** <= **int** <= **long**." All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

In Borland C++, the types **int** and **short** are equivalent, both being 16 bits. **long** is a 32-bit object. The signed varieties are all stored in 2's complement format using the most significant bit (MSB) as a

sign bit: 0 for positive, 1 for negative (which explains the ranges shown in Table 1.9 on page 19). In the unsigned versions, all bits are used to give a range of $0 - (2^n - 1)$, where n is 8, 16, or 32.

Floating-point types The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. Borland C++ uses the IEEE floating-point formats. (Appendix A, “ANSI implementation-specific standards,” tells more about implementation-specific items.)

float and **double** are 32- and 64-bit floating-point data types, respectively. **long** can be used with **double** to declare an 80-bit precision floating-point identifier: **long double test_case**, for example.

Table 1.9 on page 19 indicates the storage allocations for the floating-point types.

Standard conversions When you use an arithmetic expression, such as $a + b$, where a and b are different arithmetic types, Borland C++ performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of “lower” types to “higher” types in the interests of accuracy and consistency.

Here are the steps Borland C++ uses to convert the operands in an arithmetic expression:

1. Any small integral types are converted as shown in Table 2.6. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers, **double**, **float**, or **long double**).
2. If either operand is of type **long double**, the other operand is converted to **long double**.
3. Otherwise, if either operand is of type **double**, the other operand is converted to **double**.
4. Otherwise, if either operand is of type **float**, the other operand is converted to **float**.
5. Otherwise, if either operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
7. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
8. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Table 2.6
Methods used in standard
arithmetic conversions

Type	Converts to	Method
char	int	Zero or sign-extended (depends on default char type)
unsigned char	int	Zero-filled high byte (always)
signed char	int	Sign-extended (always)
short	int	Same value
unsigned short	unsigned int	Same value
enum	int	Same value

Special char, int, and enum conversions

*The conversions discussed in
this section are specific to
Borland C++.*

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int**.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged.

Converting a shorter integral type to a longer type either sign extends or zero fills the extra bits of the new value, depending on whether the shorter type is **signed** or **unsigned**, respectively.

Initialization

Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

*If it has automatic storage
duration, its value is
indeterminate.*

- to zero if it is of an arithmetic type
- to null if it is a pointer type

The syntax for initializers is as follows:

```
initializer
= expression
= {initializer-list} <,>
                               (expression list)
```



```
initializer-list
expression
initializer-list, expression
{initializer-list} <,>
```

Rules governing initializers are

1. The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.
2. The item to be initialized must be an object type or an array of unknown size.
3. For C (not required for C++), all expressions must be constants if they appear in one of these places:
 - a. in an initializer for an object that has static duration
 - b. in an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed)
4. If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
5. If there are fewer initializers in a brace-enclosed list than there are members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- ▣ an initializer list as described in the following section
- ▣ a single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array *days*, intended to count how many times each day of the week appears in a month (and assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

Use these rules to initialize character arrays and wide character arrays:

1. You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

```
char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for `name[0]`), 'n' (for `name[1]`), and so on (and including a null terminator).

2. You can initialize a wide character array (one that is compatible with `wchar_t`) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

Here is an example of a structure initialization:

```
struct mystruct {
    int i;
    char str[21];
    double d;
} s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces. You can eliminate the braces, but you must follow certain rules, and it isn't recommended practice.

Simple declarations

Simple declarations of variable identifiers have the following pattern:

```
data-type var1 <=init1>, var2 <=init2>, ...;
```

where *var1*, *var2*, ... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type *data-type*. For example,

```
int x = 1, y = 2;
```

creates two integer variables called *x* and *y* (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.



In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions.

Storage class specifiers

A *storage class specifier*, or a type specifier, must be present in a declaration. The storage class specifiers can be one of the following:

auto	register	typedef
extern	static	

Use of storage class specifier **auto**

The storage class specifier **auto** is used only with local scope variable declarations. It conveys local (automatic) duration, but since this is the default for all local scope variable declarations, its use is rare.

Use of storage class specifier **extern**

The storage class specifier **extern** can be used with function and variable file scope and local scope declarations to indicate external linkage. With file scope variables, the default storage class specifier is **extern**. When used with variables, **extern** indicates that the variable has static duration. (Remember that functions always have static duration.) See page 32 for information on using **extern** to prevent name mangling when combining C and C++ code.

Use of storage class specifier **register**

The storage class specifier **register** is allowed only for local variable and function parameter declarations. It is equivalent to **auto**, but it makes a request to the compiler that the variable should be allocated to a register if possible. The allocation of a register can significantly reduce the size and improve the performance of programs in many situations. However, since Borland C++ does a good job of placing variables in registers, it is rarely necessary to use the **register** keyword.

Borland C++ lets you select register variable options from the Options | Compiler | Optimizations Options dialog box. If you check Automatic, Borland C++ will try to allocate registers even if you have not used the **register** storage class specifiers.

Use of storage class specifier **static**

The storage class specifier **static** can be used with function and variable file scope and local scope declarations to indicate internal linkage. **static** also indicates that the variable has static duration. In the absence of constructors or explicit initializers, static variables are initialized with 0 or null.



In C++, a static data member of a class has the same value for all instances of a class. A static member function of a class can be invoked independently of any class instance.

Use of storage class specifier **typedef**

The keyword **typedef** indicates that you are defining a new data type specifier rather than declaring an object. **typedef** is included as a storage class specifier because of syntactical rather than functional similarities.

```
static long int biggy;  
typedef long int BIGGY;
```

The first declaration creates a 32-bit, **long int**, static-duration object called *biggy*. The second declaration establishes the identifier *BIGGY* as a new type specifier, but does not create any run-time object. *BIGGY* can be used in any subsequent declaration where a type specifier would be legal. For example,

```
extern BIGGY salary;
```

has the same effect as

```
extern long int salary;
```

Although this simple example can be achieved by `#define BIGGY long int`, more complex **typedef** applications achieve more than is possible with textual substitutions.

Important!

typedef does not create new data types; it merely creates useful mnemonic synonyms or aliases for existing types. It is especially valuable in simplifying complex declarations:

```
typedef double (*PFD)();  
PFD array_pfd[10];  
/* array_pfd is an array of 10 pointers to functions  
returning double */
```

You can't use **typedef** identifiers with other data-type specifiers:

```
unsigned BIGGY pay;      /* ILLEGAL */
```

Modifiers

In addition to the storage class specifier keywords, a declaration can use certain *modifiers* to alter some aspect of the identifier/object mapping. The modifiers available with Borland C++ are summarized in Table 2.7.

The **const** modifier

The **const** modifier prevents any assignments to the object or any other side effects, such as increment or decrement. A **const** pointer cannot be modified, though the object to which it points can be. Consider the following examples:

The modifier **const** used by itself is equivalent to **const int**.

```
const float pi = 3.1415926;
const maxint = 32767;
char *const str = "Hello, world"; // A constant pointer
char const *str2 = "Hello, world"; /* A pointer to a constant
char */
```

Given these, the following statements are illegal:

```
pi = 3.0;          /* Assigns a value to a const */
i = maxint++;     /* Increments a const */
str = "Hi, there!"; /* Points str to something else */
```

Note, however, that the function call `strcpy(str, "Hi, there!")` is legal, since it does a character-by-character copy from the string literal "Hi, there!" into the memory locations pointed to by *str*.



In C++, **const** also hides the **const** object and prevents external linkage. You need to use **extern const**. A pointer to a **const** can't be assigned to a pointer to a non-**const** (otherwise, the **const** value could be assigned to using the non-**const** pointer). For example,

```
char *str3 = str2 /* disallowed */
```

Only **const** member functions can be called for a **const** object.

Table 2.7
Borland C++ modifiers

C++ extends **const** and **volatile** to include classes and member functions.

Modifier	Use with	Use
const	Variables only	Prevents changes to object.
volatile	Variables only	Prevents register allocation and some optimization. Warns compiler that

Table 2.7: Borland C++ modifiers (continued)

		object may be subject to outside change during evaluation.
Borland C++ extensions		
<code>cdecl</code>	Functions	Forces C argument-passing convention. Affects Linker and link-time names.
<code>cdecl</code>	Variables	Forces global identifier case-sensitivity and leading underscores.
<code>pascal</code>	Functions	Forces Pascal argument-passing convention. Affects Linker and link-time names.
<code>pascal</code>	Variables	Forces global identifier case-insensitivity with no leading underscores.
<code>interrupt</code>	Functions	Function compiles with the additional register-housekeeping code needed when writing interrupt handlers.
<code>near,</code> <code>far,</code> <code>huge</code>	Pointer types	Overrides the default pointer type specified by the current memory model.
<code>_cs,</code> <code>_ds,</code> <code>_es,</code> <code>_seg,</code> <code>_ss</code>	Pointer types	Segment pointers. See page 350.
<code>near,</code> <code>far,</code> <code>huge</code>	Functions	Overrides the default function type specified by the current memory
model.		
<code>near,</code> <code>far</code>	Variables	Directs the placement of the object in memory.
<code>_export</code>	Functions/classes	Tells the compiler which functions or classes to export.
<code>_loadds</code>	Functions	Sets DS to point to the current data segment.
<code>_saveregs</code>	Functions	Preserves all register values (except for return values) during execution of the function.
<code>_fastcall</code>	Functions	Forces register parameter passing convention. Affects the linker and link-time names.

The interrupt function modifier

The **interrupt** modifier is specific to Borland C++. **interrupt** functions are designed to be used with the 8086/8088 interrupt vectors. Borland C++ will compile an **interrupt** function with extra function entry and exit code so that registers AX, BX, CX, DX, SI, DI, ES, and DS are preserved. The other registers (BP, SP, SS, CS, and IP) are preserved as part of the C-calling sequence or as part of the interrupt handling itself. The function will use an **iret** instruction to return, so that the function can be used to service hardware or software interrupts. Here is an example of a typical **interrupt** definition:

```
void interrupt myhandler()
{
    ...
}
```

You should declare interrupt functions to be of type **void**. Interrupt functions can be declared in any memory model. For all memory models except huge, DS is set to the program data segment. For the huge model, DS is set to the module's data segment.

The volatile modifier

*In C++, **volatile** has a special meaning for class member functions. If you've declared a volatile object, you can only use its volatile member functions.*

The **volatile** modifier indicates that the object may be modified; not only by you, but also by something outside of your program, such as an interrupt routine or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions containing it, since the value could (in theory) change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile int ticks;
interrupt timer()
{
    ticks++;
}

wait(int interval)
{
    ticks = 0;
    while (ticks < interval);    // Do nothing
}
```

These routines (assuming **timer** has been properly associated with a hardware clock interrupt) implement a timed wait of ticks

specified by the argument *interval*. A highly optimizing compiler might not load the value of *ticks* inside the test of the **while** loop, since the loop doesn't change the value of *ticks*.

The cdecl and pascal modifiers

*Page 31 tells how to use **extern**, which allows C names to be referenced from a C++ program.*

Borland C++ allows your programs to easily call routines written in other languages, and vice versa. When you mix languages like this, you have to deal with two important issues: identifiers and parameter passing.

In Borland C++, all global identifiers are saved in their original case (lower, upper, or mixed) with an underscore (**_**) prepended to the front of the identifier, unless you have selected the **-u** option or unchecked the Generate Underbars box in the Options | Compiler | Advanced Code Generation dialog box.

pascal

In Pascal, global identifiers are not saved in their original case, nor are underscores prepended to them. Borland C++ lets you declare any identifier to be of type **pascal**; the identifier is converted to uppercase, and no underscore is prepended. (If the identifier is a function, this also affects the parameter-passing sequence used; see "Function type modifiers," page 52, for more details.)

*The **-p** compiler option or Calling Convention Pascal in the Options | Compiler | Entry | Exit Code dialog box causes all functions (and pointers to those functions) to be treated as if they were of type **pascal**.*

The **pascal** modifier is specific to Borland C++; it is intended for functions (and pointers to functions) that use the Pascal parameter-passing sequence. Also, functions declared to be of type **pascal** can still be called from C routines, so long as the C routine sees that the function is of type **pascal**.

```
pascal putnums(int i, int j, int k)
{
    printf("And the answers are: %d, %d, and %d\n",i,j,k);
}
```

Functions of type **pascal** cannot take a variable number of arguments, unlike functions such as **printf**. For this reason, you cannot use an ellipsis (...) in a **pascal** function definition.



Most of the Windows API functions are **pascal** functions.

cdecl

Once you have compiled with Pascal calling convention turned on (using the **-p** option or IDE Options | Compiler | Entry/Exit

Code), you may want to ensure that certain identifiers have their case preserved and keep the underscore on the front, especially if they're C identifiers from another file. You can do so by declaring those identifiers to be **cdecl**. (This also has an effect on parameter passing for functions).

*main must be declared as **cdecl**; this is because the C start-up code always tries to call **main** with the C calling convention.*

Like **pascal**, the **cdecl** modifier is specific to Borland C++. It is used with functions and pointers to functions. It overrides the **-p** option or IDE Options | Compiler | Entry/Exit Code compiler directive and allows a function to be called as a regular C function. For example, if you were to compile the previous program with the Pascal calling option set but wanted to use **printf**, you might do something like this:

```
extern cdecl printf();
void putnums(int i, int j, int k);

cdecl main()
{
    putnums(1,4,9);
}

void putnums(int i, int j, int k)
{
    printf("And the answers are: %d, %d, and %d\n",i,j,k);
}
```

If you compile a program with the **-p** option or IDE Options | Compiler | Entry/Exit Code, all functions used from the run-time library will need to have **cdecl** declarations. If you look at the header files (such as `stdio.h`), you'll see that every function is explicitly defined as **cdecl** in anticipation of this.

The pointer modifiers

Borland C++ has eight modifiers that affect the pointer declarator (*); that is, they modify pointers to data. These are **near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_seg**, and **_ss**.

C lets you compile using one of several memory models. The model you use determines (among other things) the internal format of pointers. For example, if you use a small data model (tiny, small, medium), all data pointers contain a 16-bit offset from the data segment (DS) register. If you use a large data model (compact, large, huge), all pointers to data are 32 bits long and give both a segment address and an offset.

Sometimes, when using one size of data model, you want to declare a pointer to be of a different size or format than the current default. You do so using the pointer modifiers.

See the discussion starting on page 344 in Chapter 9 for an in-depth explanation of **near**, **far**, and **huge** pointers, and page 345 for a description of normalized pointers. Also see the discussion starting on page 350 for more on **_cs**, **_ds**, **_es**, **_seg**, and **_ss**.

Function type modifiers

The **near**, **far**, and **huge** modifiers can also be used as function type modifiers; that is, they can modify functions and function pointers as well as data pointers. In addition, you can use the **_export**, **_loadds**, and **_saveregs** modifiers to modify functions.

The **near**, **far**, and **huge** function modifiers can be combined with **cdecl** or **pascal**, but not with **interrupt**.

Functions of type **huge** are useful when interfacing with code in assembly language that doesn't use the same memory allocation as Borland C++.

A non-**interrupt** function can be declared to be **near**, **far**, or **huge** in order to override the default settings for the current memory model.

A **near** function uses **near** calls; a **far** or **huge** function uses **far** call instructions.

In the tiny, small, and compact memory models, an unqualified function defaults to type **near**. In the medium and large models, an unqualified function defaults to type **far**. In the huge memory model, it defaults to type **huge**.

A **huge** function is the same as a **far** function, except that the DS register is set to the data segment address of the source module when a **huge** function is entered, but left unset for a **far** function.



The **_export** modifier makes the function exportable from Windows. It's used in an executable (if you don't use smart callbacks) or in a DLL; see page 323 of Chapter 8 for details. The **_export** modifier has no significance for DOS programs.

The **_loadds** modifier indicates that a function should set the DS register, just as a huge function does, but does not imply **near** or **far** calls. Thus, **_loadds far** is equivalent to **huge**.

The **_saveregs** modifier causes the function to preserve all register values and restore them before returning (except for explicit return values passed in registers such as AX or DX).

The **_loadds** and **_saveregs** modifiers are useful for writing low-level interface routines, such as mouse support routines.

The `_fastcall` modifier is documented in Appendix A, “The Optimizer” in the *User’s Guide*.

Complex declarations and declarators

See Table 2.1 on page 35 for the declarator syntax. The definition covers both identifier and function declarators.

Simple declarations have a list of comma-delimited identifiers following the optional storage class specifiers, type specifiers, and other modifiers.

A complex declaration uses a comma-delimited list of declarators following the various specifiers and modifiers. Within each declarator, there exists just one identifier, namely the identifier being declared. Each of the declarators in the list is associated with the leading storage class and type specifier.

The format of the declarator indicates how the declared *dname* is to be interpreted when used in an expression. If *type* is any type, and *storage class specifier* is any storage class specifier, and if *D1* and *D2* are any two declarators, then the declaration

storage-class-specifier type D1, D2;

indicates that each occurrence of *D1* or *D2* in an expression will be treated as an object of type *type* and storage class *storage class specifier*. The type of the *dname* embedded in the declarator will be some phrase containing *type*, such as “*type*,” “pointer to *type*,” “array of *type*,” “function returning *type*,” or “pointer to function returning *type*,” and so on.

For example, in the declarations

```
int n, nao[], naf[3], *pn, *apn[], (*pan)[], &nr=n;
int f(void), *fnp(void), (*pfn)(void);
```

each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single `int` object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Table 2.8: Complex declarations

Declarator syntax	Implied type of name	Example
type name;	type	int count;
type name[];	(open) array of type	int count[];
type name[3];	Fixed array of three elements, all of type (<i>name</i> [0], <i>name</i> [1], and <i>name</i> [2])	int count[3];
type *name;	Pointer to type	int *count;
type *name[];	(open) array of pointers to type	int *count[];
type *(name[]);	Same as above	int *(count[]);
type (*name) [];	Pointer to an (open) array of type	int (*count) [];
type &name;	Reference to type (C++ only)	int &count;
type name();	Function returning type	int count();
type *name();	Function returning pointer to type	int *count();
type *(name());	Same as above	int *(count());
type (*name)();	Pointer to function returning type	int (*count)();

Note the need for parentheses in *(*name)[]* and *(*name)()*, since the precedence of both the array declarator `[]` and the function declarator `()` is higher than the pointer declarator `*`. The parentheses in *(*name[])* are optional.

Pointers

See page 85 for a discussion of referencing and dereferencing.

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer classes have distinct properties, purposes, and rules for manipulation, although they do share certain Borland C++ operations. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Pointers to objects

A pointer of type “pointer to object of **type**” holds the address of (that is, points to) an object of **type**. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

The size of pointers to objects is dependent on the memory model and the size and disposition of your data segments, possibly influenced by the optional pointer modifiers (discussed starting on page 51).

Pointers to functions

A pointer to a function is best thought of as an address, usually in a code segment, where that function’s executable code is stored; that is, the address to which control is transferred when that function is called. The size and disposition of your code segments is determined by the memory model in force, which in turn dictates the size of the function pointers needed to call your functions.

A pointer to a function has a type called “pointer to function returning **type**,” where **type** is the function’s return type.



Under C++, which has stronger type checking, a pointer to a function has type “pointer to function taking argument types **type** and returning **type**.” In fact, under C, a function defined with argument types will also have this narrower type. For example,

```
void (*func)();
```

In C, this is a pointer to a function returning nothing. In C++, it’s a pointer to a function taking no arguments and returning nothing. In this example,

```
void (*func)(int);
```

func* is a pointer to a function taking an **int argument and returning nothing.

Pointer

declarations

See page 39 for details on **void**.

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. Borland C++ lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to **void**. And in C, but not C++, you can assign a **void*** pointer to a non-**void*** pointer.

If **type** is any predefined or user-defined type, including **void**, the declaration

```
type *ptr; /* Danger--uninitialized pointer */
```

declares *ptr* to be of type "pointer to **type**." All the scoping, duration, and visibility rules apply to the *ptr* object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic **NULL** (defined in the standard library header files, such as `stdio.h`) can be used for legibility. All pointers can be successfully tested for equality or inequality to **NULL**.

The pointer type "pointer to void" must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that *vptr* is a generic pointer capable of being assigned to by any "pointer to **type**" value, including null, without complaint. Assignments without proper casting between a "pointer to **type1**" and a "pointer to **type2**," where **type1** and **type2** are different types, can invoke a compiler warning or error. If **type1** is a function and **type2** isn't (or vice versa), pointer assignments are illegal. If **type1** is a pointer to **void**, no cast is needed. Under C, if **type2** is a pointer to **void**, no cast is needed.

Assignment restrictions also apply to pointers of different sizes (**near**, **far**, and **huge**). You can assign a smaller pointer to a larger one without error, but you can't assign a larger pointer to a smaller one unless you are using an explicit cast. For example,

Warning! You need to initialize pointers before using them.

```

char near *ncp;
char far *fcp;
char huge *hcp;
fcp = ncp;           // legal
hcp = fcp;          // legal
fcp = hcp;          // not legal
ncp = fcp;          // not legal
ncp = (char near*)fcp; // now legal

```

Pointers and constants

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be assigned to. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```

int i;                // i is an int
int * pi;             // pi is a pointer to int
(uninitialized)
int * const cp = &i;  // cp is a constant pointer to int.
const int ci = 7;     // ci is a constant int
const int * pci;      // pci is a pointer to constant int
const int * const cpc = &ci; // cpc is a constant pointer to a
// constant int

```

The following assignments are legal:

```

i = ci;               // Assign const-int to int
*cp = ci;             // Assign const-int to
// object-pointed-at-by-a-const-pointer
++pci;               // Increment a pointer-to-const
pci = cpc;           // Assign a const-pointer-to-a-const to a
// pointer-to-const

```

The following assignments are illegal:

```

ci = 0;              // NO--cannot assign to a const-int
ci--;                // NO--cannot change a const-int
*pci = 3;           // NO--cannot assign to an object
// pointed at by pointer-to-const
cp = &ci;           // NO--cannot assign to a const-pointer,
// even if value would be unchanged
cpc++;              // NO--cannot change const-pointer

```

```

pi = pci; // NO--if this assignment were allowed,
          // you would be able to assign to *pci
          // (a const value) by assigning to *pi.

```

Similar rules apply to the **volatile** modifier. Note that **const** and **volatile** can both appear as modifiers to the same identifier.

Pointer arithmetic

The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers.

The difference between two pointers only has meaning if both pointers point into the same array.

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type “pointer to **type**” automatically take into account the size of **type**; that is, the number of bytes needed to store a **type** object.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to **type**, adding an integral value to the pointer advances the pointer by that number of objects of **type**. If **type** has size 10 bytes, then adding an integer 5 to a pointer to **type** advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if *ptr1* points to the third element of an array, and *ptr2* points to the tenth element, then the result of *ptr2* - *ptr1* would be 7.

When an integral value is added to or subtracted from a “pointer to **type**,” the result is also of type “pointer to **type**.”

There is no such element as “one past the last element”, of course, but a pointer is allowed to assume such a value. If *P* points to the last array element, *P* + 1 is legal, but *P* + 2 is undefined. If *P* points to one past the last array element, *P* - 1 is legal, giving a pointer to the last element. However, applying the indirection operator * to a “pointer to one past the last element” leads to undefined behavior.

Informally, you can think of *P* + *n* as advancing the pointer by (*n* * **sizeof(type)**) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type *ptrdiff_t* defined in `stddef.h` (**signed long** for huge and far pointers; **signed int** for all others). This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of *ptrdiff_t*. In the expression *P1* - *P2*, where *P1* and *P2* are of type pointer to **type** (or pointer to qualified **type**), *P1* and *P2* must point to existing elements or to one past the last element. If *P1* points to the *i*-th

element, and $P2$ points to the j -th element, $P1 - P2$ has the value $(i - j)$.

Pointer conversions

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (**type***) will convert a pointer to type "pointer to **type**."

C++ reference declarations

C++ reference types are closely related to pointer types. *Reference types* create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by *value*. In C++ you can pass arguments by value or by reference. See page 105, "Referencing," for complete details.

Arrays

The declaration

```
type declarator [<constant-expression>]
```

declares an array composed of elements of **type**. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. Thus, a two-dimensional array of five rows and seven columns called *alpha* is declared as

```
type alpha [5] [7];
```

In certain contexts, the first array declarator of a series may have no expression inside the brackets. Such an array is of indeter-

minate size. The contexts where this is legitimate are ones in which the size of the array is not needed to reserve space.

For example, an **extern** declaration of an array object does not need the exact dimension of the array, nor does an array function parameter. As a special extension to ANSI C, Borland C++ also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a **sizeof** or **&** operator, an array type expression is converted to a pointer to the first element of the array.

Functions

Functions are central to C and C++ programming. Languages such as Pascal distinguish between procedure and function. Borland C++ functions play both roles.

Declarations and definitions

Each program must have a single external function named **main** marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. They can be restricted by using the **static** storage class specifier (see page 31).

Functions are defined in your source files or made available by linking precompiled libraries.

In C++ you must always use function prototypes. We recommend that you also use them in C.

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide Borland C++ with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a

definition and a declaration is that the definition has a function body.)

Declarations and prototypes

In C++, this declaration means `<type> func(void)`

In the original Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows:

```
<type> func()
```

where **type** is the optional return type defaulting to **int**. A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

You can enable a warning within the IDE or with the command-line compiler: "Function called without a prototype."

This problem was eased by the introduction of function prototypes with the following declaration syntax:

```
<type> func(parameter-declarator-list);
```

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */

foo()
{
    int limit = 32;
    char ch = 'A';

    long mval;

    mval = lmax(limit, ch); /* function call */
}
```

Since it has the function prototype for **lmax**, this program converts *limit* and *ch* to **long**, using the standard rules of assignment, before it places them on the stack for the call to **lmax**. Without the function prototype, *limit* and *ch* would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to **lmax** would not match in size or content what **lmax** was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function **strcpy** takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is only used for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
func(void);
```



In C++, **func()** also declares a function taking no arguments.

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as **printf**), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

stdarg.h contains macros that you can use in user-defined functions with variable numbers of parameters.

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Here are some more examples of function declarators and prototypes:

```
int f(); /* In C, a function returning an int with no
        information about parameters. This is the K&R
        "classic style." */

int f(); /* In C++, a function taking no arguments */

int f(void); /* A function returning an int that takes no
            parameters. */

int p(int, long); /* A function returning an int that accepts two
                parameters: the first, an int; the second, a
                long. */

int pascal q(void); /* A pascal function returning an int that takes
                    no parameters at all. */

char far *s(char *source, int kind); /* A function returning a far
        pointer to a char and accepting two parameters: the
        first, a pointer to a char; the second, an int. */
```

```

int printf(char *format,...); /* A function returning an int and
                               accepting a pointer to a char fixed parameter and
                               any number of additional parameters of unknown
                               type. */

int (*fp)(int); /* A pointer to a function returning an int and
                 accepting a single int parameter. */

```

Definitions

The general syntax for external function definitions is given in the following table:

Table 2.9
External function definitions

```

file
  external-definition
  file external-definition

external-definition:
  function-definition
  declaration
  asm-statement

function-definition:
  <declaration-specifiers> declarator <declaration-list>
  compound-statement

```

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):


You can mix elements from 1 and 2.

1. Optional storage class specifiers: **extern** or **static**. The default is **extern**.
2. A return type, possibly **void**. The default is **int**.
3. Optional modifiers: **pascal**, **cdecl**, **interrupt**, **near**, **far**, **huge**, **_export**, **_loadds**, **_saveregs**. The defaults depend on the memory model and compiler option settings.
4. The name of the function.
5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is **func(void)**. The old style of **func()** is legal in C but antiquated and possibly unsafe.
6. A function body representing the code to be executed when the function is called.

Formal parameter declarations

The formal parameter declaration list follows a similar syntax to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) { // no args
int func(T1 t1, T2 t2, T3 t3=1) { // three simple parameters, one
// with default argument
int func(T1* ptr1, T2& tref) { // a pointer and a reference arg
int func(register int i) { // request register for arg
int func(char *str,...) { /* one string arg with a variable
number of other args, or with a fixed number of args with
varying types */
```

 In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, enumerations; pointers or references to structures and unions; or pointers to functions or classes.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all enjoy automatic scope and duration for the duration of the function. The only legal storage class specifier is **register**.

The **const** and **volatile** modifiers can be used with formal argument declarators.

Function calls and argument conversions

A function is called with actual arguments placed in the same sequence as their matching formal arguments. The actual arguments are converted as if by initialization to the declared types of the formal arguments.

Here is a summary of the rules governing how Borland C++ deals with language modifiers and formal parameters in function calls, both with and without prototypes:

1. The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
2. A function may modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, Borland C++ converts integral arguments to a function call according to the integral widening (expansion) rules described in the section "Standard conversions," starting on page 41. When a function prototype is in scope, Borland C++ converts the given argument to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), Borland C++ converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need only be compatible to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Important! If your function prototype does not match the actual function definition, Borland C++ will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught. C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

Structures

Structure initialization is discussed on page 42.

A *structure* is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure

member can be a bit field type not allowed elsewhere. The Borland C++ structure type lets you handle complex data structures almost as easily as single variables.



In C++, a structure type is treated as a class type (with certain differences: Default access is public, and the default for the base class is also public). This allows more sophisticated control over access to structure members by using the C++ access specifiers: **public** (the default), **private**, and **protected**. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword **struct**. For example,

```
struct mystruct { ... }; // mystruct is the structure tag
...
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */
```

Untagged structures and typedefs

Untagged structure and union members are ignored during initialization.

If you omit the structure tag, you can get an *untagged* structure. You can use untagged structures to declare the identifiers in the comma-delimited *struct-id-list* to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere:

```
struct { ... } s, *ps, arrs[10]; // untagged structure
```

It is possible to create a **typedef** while declaring a structure, with or without a tag:

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10]; // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT; // no tag
YRSTRUCT y, *yp, arry[20];
```

You don't usually need both a tag and a **typedef**: Either can be used in structure declarations.

Structure member declarations

The *member-decl-list* within the braces declares the types and names of the structure members using the declarator syntax shown in Table 2.2 on page 36.

A structure member can be of any type, with two exceptions:

1. The member type cannot be the same as the **struct** type being currently declared:

You can omit the **struct** keyword in C++.

```
struct mystruct { mystruct s } s1, s2; // illegal
```

A member can be a pointer to the structure being declared, as in the following example:

```
struct mystruct { mystruct *ps } s1, s2; // OK
```

Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.

2. Except in C++, a member cannot have the type “function returning...,” but the type “pointer to function returning...” is allowed. In C++, a **struct** can have member functions.

Structures and functions

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void); // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s); // directly
void func2(mystruct *sptr); // via a pointer
void func3(mystruct &sref); // as a reference (C++ only)
```

Structure member access

Structure and union members are accessed using the selection operators `.` and `->`. Suppose that the object `s` is of struct type `S`, and `sptr` is a pointer to `S`. Then if `m` is a member identifier of type `M` declared in `S`, the expressions `s.m` and `sptr->m` are of type `M`, and both represent the member object `m` in `s`. The expression `sptr->m` is a convenient synonym for `(*sptr).m`.

The operator `.` is called the direct member selector; the operator `->` is called the indirect (or pointer) member selector; for example,

```
struct mystruct
{
    int i;
    char str[21];
    double d;
}
```

```

} s, *sptr=&s;
...
s.i = 3;           // assign to the i member of mystruct s
sptr->d = 1.23;    // assign to the d member of mystruct s

```

The expression $s.m$ is an lvalue, provided that s is not an lvalue and m is not an array type. The expression $sptr->m$ is an lvalue unless m is an array type.

If structure B contains a field whose type is structure A , the members of A can be accessed by two applications of the member selectors:

```

struct A {
    int j;
    double x;
};

struct B {
    int i;
    struct A a;
    double d;
} s, *sptr;
...
s.i = 3;           // assign to the i member of B
s.a.j = 2;        // assign to the j member of A
sptr->d = 1.23;    // assign to the d member of B
(sptr->a).x = 3.14 // assign to x member of A

```

Each structure declaration introduces a unique structure type, so that in

```

struct A {
    int i,j;
    double d;
} a, a1;

struct B {
    int i,j;
    double d;
} b;

```

the objects a and $a1$ are both of type struct A , but the objects a and b are of different structure types. Structures can be assigned only if the source and destination have the same type:

```

a = a1; // OK: same type, so member by member assignment
a = b;  // ILLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign
                                member-by-member */

```

Structure word alignment

Memory is allocated to a structure member-by-member from left to right, from low to high memory address. In this example,

```
struct mystruct {
    int i;
    char str[21];

    double d;
} s;
```

the object *s* occupies sufficient memory to hold a 2-byte integer, a 21-byte string, and an 8-byte double. The format of this object in memory is determined by the Borland C++ word alignment option. With this option off (the default), *s* will be allocated 31 contiguous bytes.

If you turn on word alignment with the Options | Compiler | Code Generation dialog box or with the `-a` compiler option, Borland C++ pads the structure with bytes to ensure the structure is aligned as follows:

1. The structure will start on a word boundary (even address).
2. Any non-**char** member will have an even byte offset from the start of the structure.
3. A final byte is added (if necessary) at the end to ensure that the whole structure contains an even number of bytes.

With word alignment on, the structure would therefore have a byte added before the **double**, making a 32-byte object.

Structure name spaces

Structure tag names share the same name space with union tags and enumeration tags (but **enums** within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example,

```
goto s;
...
s:
struct s { // OK: tag and label name spaces different
    int s; // OK: label, tag and member name spaces different
    float s; // ILLEGAL: member name duplicated
} s; // OK: var name space different. In C++, this can only
// be done if s does not have a constructor.

union s { // ILLEGAL: tag space duplicate
    int s; // OK: new member space
    float f;
} f; // OK: var name space

struct t {
    int s; // OK: different member space
    ...
} s; // ILLEGAL: var name duplicate
```

Incomplete declarations

A pointer to a structure type *A* can legally appear in the declaration of another structure *B* before *A* has been declared:

```
struct A; // incomplete
struct B { struct A *pa };
struct A { struct B *pb };
```

The first appearance of *A* is called *incomplete* because there is no definition for it at that point. An incomplete declaration is allowed here, since the definition of *B* doesn't need the size of *A*.

Bit fields

A structure can contain any mixture of bit field and non-bit field types.

You can declare **signed** or **unsigned** integer members as bit fields from 1 to 16 bits wide. You specify the bit field width and optional identifier as follows:

```
type-specifier <bitfield-id> : width;
```

where *type-specifier* is **char**, **unsigned char**, **int**, or **unsigned int**. Bit fields are allocated from low-order to high-order bits within a word. The expression *width* must be present and must evaluate to a constant integer in the range 1 to 16.

If the bit field identifier is omitted, the number of bits specified in *width* is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused. For example,

```
struct mystruct {
    int    i : 2;
    unsigned j : 5;
    int    : 4;
    int    k : 1;
    unsigned m : 4;
} a, b, c;
```

produces the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
←-----→				←→	←-----→					←-----→		←-----→			
m				k	(unused)					j		i			

Integer fields are stored in 2's-complement form, with the leftmost bit being the MSB (most significant bit). With **int** (for example, **signed**) bit fields, the MSB is interpreted as a sign bit. A bit field of width 2 holding binary 11, therefore, would be interpreted as 3 if **unsigned**, but as -1 if **int**. In the previous example, the legal assignment `a.i = 6` would leave binary 10 = -2 in `a.i` with no warning. The signed **int** field `k` of width 1 can hold only the values -1 and 0, since the bit pattern 1 is interpreted as -1.



Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (`.` and `->`) used for non-bit field members. Also, bit fields pose several problems when writing portable code, since the organization of bits-within-bytes and bytes-within-words is machine dependent.

The expression `&mystruct.x` is illegal if `x` is a bit field identifier, since there is no guarantee that `mystruct.x` lies at a byte address.

Unions

Unions correspond to the variant record types of Pascal and Modula-2.

Union types are derived types sharing many of the syntactical and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The

value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {      /* union tag = myunion */
    int i;
    double d;
    char ch;
} mu, *muptr=&mu;
```

the identifier *mu*, of type **union myunion**, can be used to hold a 2-byte **int**, an 8-byte **double**, or a single-byte **char**, but only one of these at the same time.

sizeof(union myunion) and **sizeof(mu)** both return 8, but 6 bytes are unused (padded) when *mu* holds an **int** object, and 7 bytes are unused when *mu* holds a **char**. You access union members with the structure member selectors (**.** and **->**), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d);    // OK: displays mu.d = 4.016
printf("mu.i = %d\n",mu.i);    // peculiar result
mu.ch = 'A';
printf("mu.ch = %c\n",mu.ch);  // OK: displays mu.ch = A
printf("mu.d = %f\n",mu.d);    // peculiar result
muptr->i = 3;
printf("mu.i = %d\n",mu.i);    // OK: displays mu.i = 3
```

The second **printf** is legal, since *mu.i* is an integer type. However, the bit pattern in *mu.i* corresponds to parts of the **double** previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

Anonymous unions (C++ only)

A union that doesn't have a tag and is not used to declare a named object (or other type) is called an *anonymous union*. It has the following form:



```
union { member-list };
```

Its members can be accessed directly in the scope where this union is declared, without using the **x.y** or **p->y** syntax.

Anonymous unions can't have member functions and at file level must be declared static. In other words, an anonymous union may not have external linkage.

Union declarations

The general declaration syntax for unions is pretty much the same as that for structures. Differences are

1. Unions can contain bit fields, but only one can be active. They all start at the beginning of the union (and remember that, because bit fields are machine dependent, they pose several problems when writing portable code).
2. Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public.
3. Unions can be initialized only through their first declared member:

```
union local87 {
    int i;
    double d;
} a = { 20 };
```

4. A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union *can* have a constructor.

Enumerations

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration,

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, **enum days**, a variable *anyday* of this type, and a set of enumerators (*sun, mon,...*) with constant integer values.

Borland C++ is free to store enumerators in a single byte when Treat enums as ints is unchecked (O|C|Code Generation) or the **-b** flag. The default is on (meaning **enums** are always **ints**) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an

enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**, depending on the values of the enumerators. If all values can be represented in a **signed** or **unsigned char**, that is the type of each enumerator.



In C, a variable of an enumerated type can be assigned any value of type **int**—no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is,

```
anyday = mon;           // OK
anyday = 1;             // illegal, even though mon == 1
```

The identifier *days* is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type **enum days**:

```
enum days payday, holiday; // declare two variables
```



In C++, you can omit the **enum** keyword if *days* is not the name of anything else in the same scope.

As with **struct** and **union** declarations, you can omit the tag if no further variables of this **enum** type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

See page 17 for more on enumeration constants.

The enumerators listed inside the braces are also known as *enumeration constants*. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (*sun*) is set to zero, and each succeeding enumerator is set to one more than its predecessor (*mon* = 1, *tues* = 2, and so on).

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration,

```
/* initializer expression can include previously declared
   enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
            quarter = nickel * nickel } smallchange;
```

tuppence would acquire the value 2, *nickel* the value 5, and *quarter* the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

enum types can appear wherever **int** types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;           // OK
*daysptr = anyday;    // OK
mon = tues;             // ILLEGAL: mon is a constant
```

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```
int mon = 11;
{
    enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
    /* enumerator mon hides outer declaration of int mon */
    struct days { int i, j;}; // ILLEGAL: days duplicate tag
    double sat;             // ILLEGAL: redefinition of sat
}
mon = 12;                   // back in int mon scope
```



In C++, enumerators declared within a class are in the scope of that class.

Expressions

Table 2.11 shows how identifiers and operators are combined to form grammatically legal "phrases."

An *expression* is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in Table 2.11, indicates that expressions are defined recursively: Subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

The standard conversions are detailed in Table 2.6 on page 42.

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules which depend on the operators used, the presence of parentheses, and the data types of the operands. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by Borland C++ (see "Evaluation order" on page 78).

Expressions can produce an lvalue, an rvalue, or no value. Expressions may cause side effects whether they produce a value or not.

We've summarized the precedence and associativity of the operators in Table 2.10. The grammar in Table 2.11 on page 77 completely defines the precedence and associativity of the operators.

There are sixteen precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other. Where there are duplicates of operators in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

Table 2.10
 Associativity and
 precedence of Borland C++
 operators
*Precedence of each
 category is indicated by
 order in this table. The first
 category (the first line) has
 the highest precedence.*

Operators	Associativity
() [] -> :: .	Left to right
! ~ + - ++ -- & * (typecast) sizeof new delete	Right to left
* ->*	Left to right
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?: (conditional expression)	Right to left
= *= /= %= += -= &= ^= = <<= >>=	Right to left
,	Left to right

Table 2.11: Borland C++ expressions

<p>primary-expression: literal this (C++ specific) :: <i>identifier</i> (C++ specific) :: <i>operator-function-name</i> (C++ specific) :: <i>qualified-name</i> (C++ specific) (<i>expression</i>) <i>name</i> literal: <i>integer-constant</i> <i>character-constant</i> <i>floating-constant</i> <i>string-literal</i> name: <i>identifier</i> <i>operator-function-name</i> (C++ specific) <i>conversion-function-name</i> (C++ specific) ~ <i>class-name</i> <i>qualified-name</i> (C++ specific) qualified-name: (C++ specific) <i>qualified-class-name</i> :: <i>name</i> postfix-expression: <i>primary-expression</i> <i>postfix-expression</i> [<i>expression</i>] <i>postfix-expression</i> (<<i>expression-list</i>>) <i>simple-type-name</i> (<<i>expression-list</i>>) (C++ specific) <i>postfix-expression</i> . <i>name</i> <i>postfix-expression</i> -> <i>name</i> <i>postfix-expression</i> ++ <i>postfix-expression</i> -- expression-list: <i>assignment-expression</i> <i>expression-list</i> , <i>assignment-expression</i> unary-expression: <i>postfix-expression</i> ++ <i>unary-expression</i> -- <i>unary-expression</i> <i>unary-operator</i> <i>cast-expression</i> sizeof <i>unary-expression</i> sizeof (<i>type-name</i>) <i>allocation-expression</i> (C++ specific) <i>deallocation-expression</i> (C++ specific) unary-operator: one of & * + - - ! allocation-expression: (C++ specific) <<::> new <<i>placement</i>> <i>new-type-name</i> <<i>initializer</i>> <<::> new <<i>placement</i>> (<i>type-name</i>) <<i>initializer</i>> placement: (C++ specific) (<i>expression-list</i>) new-type-name: (C++ specific) <i>type-specifiers</i> <<i>new-declarator</i>> new-declarator: (C++ specific) <i>ptr-operator</i> <<i>new-declarator</i>> <i>new-declarator</i> [<<i>expression</i>>] deallocation-expression: (C++ specific) <<::> delete <i>cast-expression</i> <<::> delete [] <i>cast-expression</i> cast-expression: <i>unary-expression</i></p>	<p>(<i>type-name</i>) <i>cast-expression</i> pm-expression: <i>cast-expression</i> <i>pm-expression</i> .* <i>cast-expression</i> (C++ specific) <i>pm-expression</i> ->* <i>cast-expression</i> (C++ specific) multiplicative-expression: <i>pm-expression</i> <i>multiplicative-expression</i> * <i>pm-expression</i> <i>multiplicative-expression</i> / <i>pm-expression</i> <i>multiplicative-expression</i> % <i>pm-expression</i> additive-expression: <i>multiplicative-expression</i> <i>additive-expression</i> + <i>multiplicative-expression</i> <i>additive-expression</i> - <i>multiplicative-expression</i> shift-expression: <i>additive-expression</i> <i>shift-expression</i> << <i>additive-expression</i> <i>shift-expression</i> >> <i>additive-expression</i> relational-expression: <i>shift-expression</i> <i>relational-expression</i> < <i>shift-expression</i> <i>relational-expression</i> > <i>shift-expression</i> <i>relational-expression</i> <= <i>shift-expression</i> <i>relational-expression</i> >= <i>shift-expression</i> equality-expression: <i>relational-expression</i> <i>equality-expression</i> == <i>relational-expression</i> <i>equality-expression</i> != <i>relational-expression</i> AND-expression: <i>equality-expression</i> <i>AND-expression</i> & <i>equality-expression</i> exclusive-OR-expression: <i>AND-expression</i> <i>exclusive-OR-expression</i> ^ <i>AND-expression</i> inclusive-OR-expression: <i>exclusive-OR-expression</i> <i>inclusive-OR-expression</i> <i>exclusive-OR-expression</i> logical-AND-expression: <i>inclusive-OR-expression</i> <i>logical-AND-expression</i> && <i>inclusive-OR-expression</i> logical-OR-expression: <i>logical-AND-expression</i> <i>logical-OR-expression</i> <i>logical-AND-expression</i> conditional-expression: <i>logical-OR-expression</i> <i>logical-OR-expression</i> ? <i>expression</i> : <i>conditional-expression</i> assignment-expression: <i>conditional-expression</i> <i>unary-expression</i> <i>assignment-operator</i> <i>assignment-expression</i> assignment-operator: one of = * = / = % = + = - = << = >> = & = ^ = =</p>
<p>expression: <i>assignment-expression</i> <i>expression</i>, <i>assignment-expression</i> constant-expression: <i>conditional-expression</i></p>	

Expressions and

C++

C++ allows the overloading of certain standard C operators, as explained starting on page 136. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the relational operator `==` might be defined in the class **complex** to test the equality of two complex numbers without changing its normal usage with non-class data types. An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the rules for operators and conversions discussed in this section may not apply to expressions in C++.

Evaluation order

The order in which Borland C++ evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. Consider the expression

```
i = v[i++]; // i is undefined
```

The value of *i* depends on whether *i* is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for *sum* and *total*. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value.

Borland C++ regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the rearrangement affect the value of the expression.

You can use parentheses to force the order of evaluation in expressions. For example, if you have the variables *a*, *b*, *c*, and *f*, then the expression $f = a + (b + c)$ forces $(b + c)$ to be evaluated before adding the result to *a*.

Errors and overflows

See **matherr** and **signal** in the *Library Reference*.

We've summarized the precedence and associativity of the operators in Table 2.10. During the evaluation of an expression, Borland C++ can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2^n arithmetic on *n*-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines.

Operator semantics

The Borland C++ operators described here are the standard ANSI C operators.

Unless the operators are overloaded, the following information is true in both C and C++. In C++ you can overload all of these operators with the exception of `.` (member operator) and `?:` (conditional operator) (and you also can't overload the C++ operators `::` and `.*`).

If an operator is overloaded, the discussion may not be true for it anymore. Table 2.11 on page 77 gives the syntax for all operators and operator expressions.

Operator descriptions

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. Borland C++ is especially rich in operators, offering not only the common arithmetical and logical operators, but also many for bit-level

manipulations, structure and union component access, and pointer operations (referencing and dereferencing).



Overloading is covered starting on page 135.

C++ extensions offer additional operators for accessing class members and their objects, together with a mechanism for overloading operators. *Overloading* lets you redefine the action of any standard operators when applied to the objects of a given class. In this section, we confine our discussion to the standard operators of Borland C++.

After defining the standard operators, we discuss data types and declarations, and explain how these affect the actions of each operator. From there we can proceed with the syntax for building expressions from operators, punctuators, and objects.

The operators in Borland C++ are defined as follows:

operator: one of

[]	()	.	->	++	--
&	*	+	-	~	!
sizeof	/	%	<<	>>	<
>	<=	>=	==	!=	^
	&&		?:	=	*=
/=	%=	+=	-=	<<=	>>=
&=	^=	=	,	#	##

The operators # and ## are used only by the preprocessor (see page 157).

And the following operators specific to C++:

:: .* ->*

Except for [], (), and ?:, which bracket expressions, the multicharacter operators are considered as single tokens. The same operator token can have more than one interpretation, depending on the context. For example,

A * B	Multiplication
*ptr	Dereference (indirection)
A & B	Bitwise AND
&A	Address operation
int &	Reference modifier (C++)
label:	Statement label
a ? x : y	Conditional statement
void func(int n);	Function declaration
a = (b+c)*d;	Parenthesized expression
a, b, c;	Comma expression

<code>func(a, b, c);</code>	Function call
<code>a = ~b;</code>	Bitwise negation (one's complement)
<code>~func() {delete a;}</code>	Destructor (C++)

Unary operators

<code>&</code>	Address operator
<code>*</code>	Indirection operator
<code>+</code>	Unary plus
<code>-</code>	Unary minus
<code>~</code>	Bitwise complement (1's complement)
<code>!</code>	Logical negation
<code>++</code>	Prefix: preincrement; Postfix: postincrement
<code>--</code>	Prefix: predecrement; Postfix: postdecrement

Binary operators

Additive operators

<code>+</code>	Binary plus (addition)
<code>-</code>	Binary minus (subtraction)

Multiplicative operators

<code>*</code>	Multiply
<code>/</code>	Divide
<code>%</code>	Remainder

Shift operators

<code><<</code>	Shift left
<code>>></code>	Shift right

Bitwise operators

<code>&</code>	Bitwise AND
<code>^</code>	Bitwise XOR (exclusive OR)
<code> </code>	Bitwise inclusive OR

Logical operators

<code>&&</code>	Logical AND
<code> </code>	Logical OR

Assignment operators

<code>=</code>	Assignment
<code>*=</code>	Assign product
<code>/=</code>	Assign quotient
<code>%=</code>	Assign remainder (modulus)
<code>+=</code>	Assign sum
<code>-=</code>	Assign difference

	<code><<=</code>	Assign left shift
	<code>>>=</code>	Assign right shift
	<code>&=</code>	Assign bitwise AND
	<code>^=</code>	Assign bitwise XOR
	<code> =</code>	Assign bitwise OR
Relational operators	<code><</code>	Less than
	<code>></code>	Greater than
	<code><=</code>	Less than or equal to
	<code>>=</code>	Greater than or equal to
Equality operators	<code>==</code>	Equal to
	<code>!=</code>	Not equal to
Component selection operators	<code>.</code>	Direct component selector
	<code>-></code>	Indirect component selector
Class-member operators	<code>::</code>	Scope access/resolution
	<code>.*</code>	Dereference pointer to class member
	<code>->*</code>	Dereference pointer to class member
Conditional operator	<code>a ? x : y</code>	"if <i>a</i> then <i>x</i> ; else <i>y</i> "
Comma operator	<code>,</code>	Evaluate; e.g., <i>a</i> , <i>b</i> , <i>c</i> ; from left to right

The operator functions, as well as their syntax, precedences, and associativities, are covered starting on page 75.

Postfix and prefix operators

The six postfix operators `[]` `()` `.` `->` `++` and `--` are used to build postfix expressions as shown in the expressions syntax table (Table 2.11). The increment and decrement operators (`++` and `--`) are also prefix and unary operators; they are discussed starting on page 84.

Array subscript operator `[]`

In the expression

postfix-expression [*expression*]

either *postfix-expression* or *expression* must be a pointer and the other an integral type.

In C, but not necessarily in C++, the expression *exp1[exp2]* is defined as

** ((exp1) + (exp2))*

where either *exp1* is a pointer and *exp2* is an integer, or *exp1* is an integer and *exp2* is a pointer. (The punctuators `[]`, `*`, and `+` can be individually overloaded in C++.)

Function call operators (`()`)

The expression

postfix-expression(<arg-expression-list>)

is a call to the function given by the postfix expression. The *arg-expression-list* is a comma-delimited list of expressions of any type representing the actual (or real) function arguments. The value of the function call expression, if any, is determined by the return statement in the function definition. See “Function calls and argument conversions,” page 64, for more on function calls.

Structure/union member operator `.` (`dot`)

In the expression

postfix-expression . identifier

the postfix expression must be of type structure or union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if and only if the postfix expression is an lvalue. Detailed examples of the use of `.` and `->` for structures are given on page 67.

lvalues are defined on page 26.

Structure/union pointer operator `->`

In the expression

postfix-expression -> identifier

the postfix expression must be of type pointer to structure or pointer to union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if and only if the postfix expression is an lvalue.

Postfix increment
operator ++

In the expression

postfix-expression ++

the postfix expression is the operand; it must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue (see page 26 for more on modifiable lvalues). The postfix ++ is also known as the *postincrement* operator. The value of the whole expression is the value of the postfix expression *before* the increment is applied. After the postfix expression is evaluated, the operand is incremented by 1. The increment value is appropriate to the type of the operand. Pointer types are subject to the rules for pointer arithmetic.

Postfix decrement
operator --

The postfix decrement, also known as the *postdecrement*, operator follows the same rules as the postfix increment, except that 1 is subtracted from the operand *after* the evaluation.

Increment and
decrement
operators

The first two unary operators are ++ and --. These are also postfix and prefix operators, so they are discussed here. The remaining six unary operators are covered following this discussion.

Prefix increment
operator ++

In the expression

++ unary-expression

the unary expression is the operand; it must be of scalar type and must be a modifiable lvalue. The prefix increment operator is also known as the *preincrement* operator. The operand is incremented by 1 *before* the expression is evaluated; the value of the whole expression is the incremented value of the operand. The 1 used to increment is the appropriate value for the type of the operand. Pointer types follow the rules of pointer arithmetic.

Prefix decrement
operator --

The prefix decrement, also known as the *predecrement*, operator has the following syntax:

-- unary-expression

It follows the same rules as the prefix increment operator, except that the operand is decremented by 1 before the whole expression is evaluated.

Unary operators

The six unary operators (aside from ++ and --) are & * + - ~ and !. The syntax is

unary-operator cast-expression

cast-expression:

unary-expression

(type-name) cast-expression

Address operator &

The symbol & is also used in C++ to specify reference types; see page 105.

The & operator and * operator (the * operator is described in the next section) work together as the *referencing* and *dereferencing* operators. In the expression

& cast-expression

the *cast-expression* operand must be either a function designator or an lvalue designating an object that is not a bit field and is not declared with the **register** storage class specifier. If the operand is of type **type**, the result is of type pointer to **type**.



Some non-lvalue identifiers, such as function names and array names, are automatically converted into "pointer to X" types when appearing in certain contexts. The & operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following extract:

```
type t1 = 1, t2 = 2;
type *ptr = &t1;    // initialized pointer
*ptr = t2;         // same effect as t1 = t2
```

type *ptr = &t1 is treated as

```
T *ptr;
ptr = &t1;
```

so it is *ptr*, not **ptr*, that gets assigned. Once *ptr* has been initialized with the address *&t1*, it can be safely dereferenced to give the lvalue **ptr*.

Indirection operator * In the expression

** cast-expression*

the *cast-expression* operand must have type "pointer to **type**," where **type** is any type. The result of the indirection is of type **type**. If the operand is of type "pointer to function," the result is a function designator; if the operand is a pointer to an object, the result is an lvalue designating that object. In the following situations, the result of indirection is undefined:

1. The *cast-expression* is a null pointer.
2. The *cast-expression* is the address of an automatic variable and execution of its block has terminated.

Unary plus operator + In the expression

+ cast-expression

the *cast-expression* operand must be of arithmetic type. The result is the value of the operand after any required integral promotions.

Unary minus operator - In the expression

- cast-expression

the *cast-expression* operand must be of arithmetic type. The result is the negative of the value of the operand after any required integral promotions.

Bitwise complement operator ~ In the expression

~ cast-expression

the *cast-expression* operand must be of integral type. The result is the bitwise complement of the operand after any required integral promotions. Each 0 bit in the operand is set to 1, and each 1 bit in the operand is set to 0.

Logical negation operator ! In the expression

! cast-expression

the *cast-expression* operand must be of scalar type. The result is of type **int** and is the logical negation of the operand: 0 if the op-

erand is nonzero; 1 if the operand is zero. The expression *!E* is equivalent to $(0 \neq E)$.

The sizeof operator

There are two distinct uses of the **sizeof** operator:

sizeof *unary-expression*
sizeof (*type-name*)

How much space is set aside for each type depends on the machine.

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is *not* converted to a pointer type). The number of elements in an array equals **sizeof array**/**sizeof array[0]**.

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

sizeof cannot be used with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is **size_t**, defined as **unsigned int** in `stddef.h`.

You can use **sizeof** in preprocessor directives; this is specific to Borland C++.



In C++, **sizeof(*classtype*)**, where *classtype* is derived from some base class, returns the size of the object (remember, this includes the size of the base class size).

Multiplicative operators

There are three multiplicative operators: *****, **/** and **%**. The syntax is

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*

multiplicative-expression / cast-expression
multiplicative-expression % cast-expression

The operands for `*` (multiplication) and `/` (division) must be of arithmetical type. The operands for `%` (modulus, or remainder) must be of integral type. The usual arithmetic conversions are made on the operands (see page 41).

The result of $(op1 * op2)$ is the product of the two operands. The results of $(op1 / op2)$ and $(op1 \% op2)$ are the quotient and remainder, respectively, when $op1$ is divided by $op2$, provided that $op2$ is nonzero. Use of `/` or `%` with a zero second operand results in an error.

When $op1$ and $op2$ are integers and the quotient is not an integer, the results are as follows:

Rounding is always toward zero.

1. If $op1$ and $op2$ have the same sign, $op1 / op2$ is the largest integer less than the true quotient, and $op1 \% op2$ has the sign of $op1$.
2. If $op1$ and $op2$ have opposite signs, $op1 / op2$ is the smallest integer greater than the true quotient, and $op1 \% op2$ has the sign of $op1$.

Additive operators

There are two additive operators: `+` and `-`. The syntax is

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

The addition operator `+`

The legal operand types for $op1 + op2$ are

1. Both $op1$ and $op2$ are of arithmetic type.
2. $op1$ is of integral type, and $op2$ is of pointer to object type.
3. $op1$ is of integral type, and $op1$ is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands. In cases 2 and 3, the rules of pointer arithmetic apply. (Pointer arithmetic is covered on page 58.)

The subtraction operator -



The legal operand types for $op1 - op2$ are

1. Both $op1$ and $op2$ are of arithmetic type.
2. Both $op1$ and $op2$ are pointers to compatible object types. The unqualified type **type** is considered to be compatible with the qualified types **const type**, **volatile type**, and **const volatile type**.
3. $op1$ is of pointer to object type, and $op2$ is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands. In cases 2 and 3, the rules of pointer arithmetic apply.

Bitwise shift operators

There are two bitwise shift operators: \ll and \gg . The syntax is

shift-expression:
additive-expression
shift-expression \ll *additive-expression*
shift-expression \gg *additive-expression*

Bitwise shift operators
(\ll and \gg)

In the expressions $E1 \ll E2$ and $E1 \gg E2$, the operands $E1$ and $E2$ must be of integral type. The normal integral promotions are performed on $E1$ and $E2$, and the type of the result is the type of the promoted $E1$. If $E2$ is negative or is greater than or equal to the width in bits of $E1$, the operation is undefined.

The constants `ULONG_MAX` and `UINT_MAX` are defined in `limits.h`.

The result of $E1 \ll E2$ is the value of $E1$ left-shifted by $E2$ bit positions, zero-filled from the right if necessary. Left shifts of an **unsigned long** $E1$ are equivalent to multiplying $E1$ by 2^{E2} , reduced modulo $ULONG_MAX + 1$; left shifts of **unsigned ints** are equivalent to multiplying by 2^{E2} reduced modulo $UINT_MAX + 1$. If $E1$ is a signed integer, the result must be interpreted with care, since the sign bit may change.

The result of $E1 \gg E2$ is the value of $E1$ right-shifted by $E2$ bit positions. If $E1$ is of **unsigned** type, zero-fill occurs from the left if necessary. If $E1$ is of **signed** type, the fill from the left uses the sign bit (0 for positive, 1 for negative $E1$). This sign-bit extension ensures that the sign of $E1 \gg E2$ is the same as the sign of $E1$. Except for signed types, the value of $E1 \gg E2$ is the integral part of the quotient $E1/2^{E2}$.

Relational operators

There are four relational operators: `<` `>` `<=` and `>=`. The syntax for these operators is:

```
relational-expression:  
  shift-expression  
  relational-expression < shift-expression  
  relational-expression > shift-expression  
  relational-expression <= shift-expression  
  relational-expression >= shift-expression
```

The less-than operator `<`

In the expression `E1 < E2`, the operands must conform to one of the following sets of conditions:

Qualified names are defined on page 117.

1. Both `E1` and `E2` are of arithmetic type.
2. Both `E1` and `E2` are pointers to qualified or unqualified versions of compatible object types.
3. Both `E1` and `E2` are pointers to qualified or unqualified versions of compatible incomplete types.

In case 1, the usual arithmetic conversions are performed. The result of `E1 < E2` is of type `int`. If the value of `E1` is less than the value of `E2`, the result is 1 (true); otherwise, the result is zero (false).

In cases 2 and 3, where `E1` and `E2` are pointers to compatible types, the result of `E1 < E2` depends on the relative locations (addresses) of the two objects being pointed at. When comparing structure members within the same structure, the “higher” pointer indicates a later declaration. Within arrays, the “higher” pointer indicates a larger subscript value. All pointers to members of the same union object compare as equal.

Normally, the comparison of pointers to different structure, array, or union objects, or the comparison of pointers outside the range of an array object give undefined results; however, an exception is made for the “pointer beyond the last element” situation as discussed under “Pointer arithmetic” on page 58. If `P` points to an element of an array object, and `Q` points to the last element, the expression `P < Q + 1` is allowed, evaluating to 1 (true), even though `Q + 1` does not point to an element of the array object.

The greater-than operator >	The expression $E1 > E2$ gives 1 (true) if the value of $E1$ is greater than the value of $E2$; otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.
The less-than or equal-to operator <=	Similarly, the expression $E1 <= E2$ gives 1 (true) if the value of $E1$ is less than or equal to the value of $E2$. Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.
The greater-than or equal-to operator >=	Finally, the expression $E1 >= E2$ gives 1 (true) if the value of $E1$ is greater than or equal to the value of $E2$. Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.

Equality operators

There are two equality operators: `==` and `!=`. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.

➔ However, `==` and `!=` have a lower precedence than the relational operators `<`, `>`, `<=`, and `>=`. Also, `==` and `!=` can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

The syntax is

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

The equal-to operator `==`

In the expression $E1 == E2$, the operands must conform to one of the following sets of conditions:

1. Both $E1$ and $E2$ are of arithmetic type.
2. Both $E1$ and $E2$ are pointers to qualified or unqualified versions of compatible types.

3. One of *E1* and *E2* is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of **void**.
4. One of *E1* or *E2* is a pointer and the other is a null pointer constant.

If *E1* and *E2* have types that are valid operand types for a relational operator, the same comparison rules just detailed for *E1* < *E2*, *E1* <= *E2*, and so on, apply.

In case 1, for example, the usual arithmetic conversions are performed, and the result of *E1* == *E2* is of type **int**. If the value of *E1* is equal to the value of *E2*, the result is 1 (true); otherwise, the result is zero (false).

In case 2, *E1* == *E2* gives 1 (true) if *E1* and *E2* point to the same object, or both point "one past the last element" of the same array object, or both are null pointers.

If *E1* and *E2* are pointers to function types, *E1* == *E2* gives 1 (true) if they are both null or if they both point to the same function. Conversely, if *E1* == *E2* gives 1 (true), then either *E1* and *E2* point to the same function, or they are both null.

In case 4, the pointer to an object or incomplete type is converted to the type of the other operand (pointer to a qualified or unqualified version of **void**).

The inequality operator `!=` The expression *E1* != *E2* follows the same rules as those for *E1* == *E2*, except that the result is 1 (true) if the operands are unequal, and 0 (false) if the operands are equal.

Bitwise AND operator &

The syntax is

AND-expression:
equality-expression
AND-expression & *equality-expression*

In the expression *E1* & *E2*, both operands must be of integral type. The usual arithmetical conversions are performed on *E1* and *E2*, and the result is the bitwise AND of *E1* and *E2*. Each bit in the result is determined as shown in Table 2.12.

Table 2.12
Bitwise operators truth table

Bit value in E1	Bit value in E2	E1 & E2	E1 ^ E2	E1 E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Bitwise exclusive OR operator ^

The syntax is

exclusive-OR-expression:
AND-expression
exclusive-OR-expression ^ *AND-expression*

In the expression $E1 \wedge E2$, both operands must be of integral type. The usual arithmetic conversions are performed on $E1$ and $E2$, and the result is the bitwise exclusive OR of $E1$ and $E2$. Each bit in the result is determined as shown in Table 2.12.

Bitwise inclusive OR operator |

The syntax is

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*

In the expression $E1 | E2$, both operands must be of integral type. The usual arithmetic conversions are performed on $E1$ and $E2$, and the result is the bitwise inclusive OR of $E1$ and $E2$. Each bit in the result is determined as shown in Table 2.12.

Logical AND operator &&

The syntax is

logical-AND-expression:
inclusive-OR-expression
logical-AND-expression && *inclusive-OR-expression*

In the expression $E1 \&\& E2$, both operands must be of scalar type. The result is of type `int`, the result is 1 (true) if the values of $E1$ and $E2$ are both nonzero; otherwise, the result is 0 (false).

Unlike the bitwise **&** operator, **&&** guarantees left-to-right evaluation. *E1* is evaluated first; if *E1* is zero, *E1 && E2* gives 0 (false), and *E2* is not evaluated.

Logical OR operator | |

The syntax is

logical-OR-expression
logical-AND-expression
logical-OR-expression | | *logical-AND-expression*

In the expression *E1* | | *E2*, both operands must be of scalar type. The result is of type **int**, and the result is 1 (true) if either of the values of *E1* and *E2* are nonzero. Otherwise, the result is 0 (false).

Unlike the bitwise | operator, | | guarantees left-to-right evaluation. *E1* is evaluated first; if *E1* is nonzero, *E1* | | *E2* gives 1 (true), and *E2* is not evaluated.

Conditional operator ? :

The syntax is

conditional-expression
logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expression*

In C++, the result is an lvalue.

In the expression *E1* ? *E2* : *E3*, the operand *E1* must be of scalar type. The operands *E2* and *E3* must obey one of the following sets of rules:

1. Both of arithmetic type
2. Both of compatible structure or union types
3. Both of void type
4. Both of type pointer to qualified or unqualified versions of compatible types
5. One operand of pointer type, the other a null pointer constant
6. One operand of type pointer to an object or incomplete type, the other of type pointer to a qualified or unqualified version of void

First, *E1* is evaluated; if its value is nonzero (true), then *E2* is evaluated and *E3* is ignored. If *E1* evaluates to zero (false), then *E3* is

evaluated and $E2$ is ignored. The result of $E1 ? E2 : E3$ will be the value of whichever of $E2$ and $E3$ is evaluated.

In case 1, both $E2$ and $E3$ are subject to the usual arithmetic conversions, and the type of the result is the common type resulting from these conversions.

In case 2, the type of the result is the structure or union type of $E2$ and $E3$.

In case 3, the result is of type **void**.

In cases 4 and 5, the type of the result is pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.

In case 6, the type of the result is that of the nonpointer-to-void operand.

Assignment operators

There are eleven assignment operators. The `=` operator is the simple assignment operator; the other ten are known as compound assignment operators.

The syntax is

assignment-expression:

conditional-expression

unary-expression assignment-operator assignment-expression

assignment-operator: one of

`=` `*=` `/=` `%=` `+=` `-=`
`<<=` `>>=` `&=` `^=` `|=`

The simple assignment operator `=`

In the expression $E1 = E2$, $E1$ must be a modifiable lvalue. The value of $E2$, after conversion to the type of $E1$, is stored in the object designated by $E1$ (replacing $E1$'s previous value). The value of the assignment expression is the value of $E1$ after the assignment. The assignment expression is not itself an lvalue.

In C++, the result is an lvalue.

The operands $E1$ and $E2$ must obey one of the following sets of rules:

1. $E1$ is of qualified or unqualified arithmetic type and $E2$ is of arithmetic type.

2. *E1* has a qualified or unqualified version of a structure or union type compatible with the type of *E2*.
3. *E1* and *E2* are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
4. One of *E1* or *E2* is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
5. *E1* is a pointer and *E2* is a null pointer constant.

The compound assignment operators

The compound assignments *op=*, where *op* can be any one of the ten operator symbols * / % + - << >> & ^ |, are interpreted as follows:

$$E1 \text{ op} = E2$$

has the same effect as

$$E1 = E1 \text{ op } E2$$

except that the lvalue *E1* is evaluated only once. For example, *E1 += E2* is the same as *E1 = E1 + E2*.

The rules for compound assignment are therefore covered in the previous section (on the simple assignment operator =).

Comma operator

The syntax is

expression:
assignment-expression
expression , *assignment-expression*

In C++, the result is an lvalue.

In the comma expression

$$E1, E2$$

the left operand *E1* is evaluated as a **void** expression, then *E2* is evaluated to give the result and type of the comma expression. By recursion, the expression

$$E1, E2, \dots, E_n$$

results in the left-to-right evaluation of each *E_i*, with the value and type of *E_n* giving the result of the whole expression. To avoid

ambiguity with the commas used in function argument and initializer lists, parentheses must be used. For example,

```
func(i, (j = 1, j + 4), k);
```

calls **func** with three arguments, not four. The arguments are *i*, 5, and *k*.

C++ operators

See page 108 for information on the scope access operator (::).

The operators specific to C++ are ::, *, and ->*. The syntax for the *, and ->* operators is as follows:

```
pm-expression
  cast-expression
  pm-expression .* cast-expression
  pm-expression ->* cast-expression
```

The .* operator dereferences pointers to class members. It binds the *cast-expression*, which must be of type "pointer to member of class *type*", to the *pm-expression*, which must be of class *type* or of a class publicly derived from class *type*. The result is an object or function of the type specified by the *cast-expression*.

The ->* operator dereferences pointers to pointers to class members (no, that isn't a typo; it does indeed dereference pointers to pointers). It binds the *cast-expression*, which must be of type "pointer to member of *type*," to the *pm-expression*, which must be of type pointer to *type* or of type "pointer to class publicly derived from *type*." The result is an object or function of the type specified by the *cast-expression*.

If the result of either of these operators is a function, you can only use that result as the operand for the function call operator (). For example,

```
(ptr2object->*ptr2memberfunc) (10);
```

calls the member function denoted by *ptr2memberfunc* for the object pointed to be *ptr2object*.

Statements

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. Table 2.13 on page 98 lays out the syntax for statements:

Blocks

A compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces (`{ }`). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth.

Table 2.13: Borland C++ statements

<i>statement:</i> <i>labeled-statement</i> <i>compound-statement</i> <i>expression-statement</i> <i>selection-statement</i> <i>iteration-statement</i> <i>jump-statement</i> <i>asm-statement</i> <i>declaration (C++ specific)</i>	<i>declaration-list declaration</i>
<i>asm-statement:</i> asm tokens newline asm tokens; asm { tokens; <tokens;>= <tokens;> }	<i>statement-list:</i> statement statement-list statement
<i>labeled-statement:</i> identifier : statement case constant-expression : statement default : statement	<i>expression-statement:</i> <expression> ;
<i>compound-statement:</i> { <declaration-list> <statement-list> }	<i>selection-statement:</i> if (expression) statement if (expression) statement else statement switch (expression) statement
<i>declaration-list:</i> declaration	<i>iteration-statement:</i> while (expression) statement do statement while (expression) ; for (for-init-statement <expression> ; <expression>) statement
	<i>for-init-statement</i> expression-statement declaration (C++ specific)
	<i>jump-statement:</i> goto identifier ; continue ; break ; return <expression> ;

Labeled statements

A statement can be labeled in the following ways:

1. *label-identifier* : statement

The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and enjoy function scope. In C++ you can label both declaration and non-declaration statements.



2. **case** constant-expression : statement
 default : statement

Case and default labeled statements are used only in conjunction with switch statements.

Expression statements

Any expression followed by a semicolon forms an *expression statement*:

<expression>;

Borland C++ executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls.

A special case is the *null statement*, consisting of a single semicolon (;). The null statement does nothing. It is nevertheless useful in situations where the Borland C++ syntax expects a statement but your program does not need one.

Selection statements

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.

if statements

The parentheses around cond-expression are essential.

The basic **if** statement has the following pattern:

if (*cond-expression*) *t-st* **<else** *f-st***>**

The *cond-expression* must be of scalar type. The expression is evaluated. If the value is zero (or null for pointer types), we say that the *cond-expression* is false; otherwise, it is true.

If there is no **else** clause and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored.

If the optional **else** *f-st* is present and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored and *f-st* is executed.



Unlike, say, Pascal, Borland C++ does not have a specific Boolean data type. Any expression of integer or pointer type can serve a Boolean role in conditional tests. The relational expression (*a > b*) (if legal) evaluates to **int** 1 (true) if (*a > b*), and to **int** 0 (false) if (*a <= b*). Pointer conversions are such that a pointer can always be correctly compared to a constant expression evaluating to 0. That is, the test for null pointers can be written **if (!ptr)...** or **if (ptr == 0)....**

The *f-st* and *t-st* statements can themselves be **if** statements, allowing for a series of conditional tests nested to any depth. Care is needed with nested **if...else** constructs to ensure that the correct statements are selected. There is no **endif** statement: Any “else” ambiguity is resolved by matching an **else** with the last encountered **if-without-an-else** at the same block level. For example,

```
if (x == 1)
    if (y == 1) puts("x=1 and y=1");
    else puts("x != 1");
```

draws the wrong conclusion! The **else** matches with the second **if**, despite the indentation. The correct conclusion is that $x = 1$ and $y \neq 1$. Note the effect of braces:

```
if (x == 1)
{
    if (y == 1) puts("x = 1 and y = 1");
}
else puts("x != 1"); // correct conclusion
```

switch statements The **switch** statement uses the following basic format:

switch (*sw-expression*) *case-st*

A **switch** statement lets you transfer control to one of several case-labeled statements, depending on the value of *sw-expression*. The latter must be of integral type (in C++, it can be of class type, provided that there is an unambiguous conversion to integral type available). Any statement in *case-st* (including empty statements) can be labeled with one or more case labels:

case *const-exp-i* : *case-st-i*

where each case constant, *const-exp-i*, is a constant expression with a unique integer value (converted to the type of the controlling expression) within its enclosing **switch** statement.

There can also be at most one **default** label:

default : *default-st*

After evaluating *sw-expression*, a match is sought with one of the *const-exp-i*. If a match is found, control passes to the statement *case-st-i* with the matching case label.

If no match is found and there is a **default** label, control passes to *default-st*. If no match is found and there is no **default** label, none

It is illegal to have duplicate case constants in the same switch statement.

of the statements in *case-st* is executed. Program execution is not affected when **case** and **default** labels are encountered. Control simply passes through the labels to the following statement or switch. To stop execution at the end of a group of statements for a particular case, use **break**.

Iteration statements

Iteration statements let you loop a set of statements. There are three forms of iteration in Borland C++: **while**, **do**, and **for** loops.

while statements

The parentheses are essential.

The general format for this statement is

```
while (cond-exp) t-st
```

The loop statement, *t-st*, will be executed repeatedly until the conditional expression, *cond-exp*, compares equal to zero (false).

The *cond-exp* is evaluated and tested first (as described on page 99). If this value is nonzero (true), *t-st* is executed; if no jump statements that exit from the loop are encountered, *cond-exp* is evaluated again. This cycle repeats until *cond-exp* is zero.

As with **if** statements, pointer type expressions can be compared with the null pointer, so that `while (ptr)...` is equivalent to

```
while (ptr != NULL)...
```

The **while** loop offers a concise method for scanning strings and other null-terminated data structures:

```
char str[10]="Borland";  
char *ptr=&str[0];  
int count=0;  
//...  
while (*ptr++) // loop until end of string  
    count++;
```

In the absence of jump statements, *t-st* must affect the value of *cond-exp* in some way, or *cond-exp* itself must change during evaluation in order to prevent unwanted endless loops.

do while statements

The general format is

```
do do-st while (cond-exp);
```

The *do-st* statement is executed repeatedly until *cond-exp* compares equal to zero (false). The key difference from the **while** statement is that *cond-exp* is tested *after*, rather than before, each

execution of the loop statement. At least one execution of *do-st* is assured. The same restrictions apply to the type of *cond-exp* (scalar).

for statements The **for** statement format in C is

For C++, *<init-exp>* can be an expression or a declaration.

for (*<init-exp>*; *<test-exp>*; *<increment-exp>*) *statement*

The sequence of events is as follows:

1. The initializing expression *init-exp*, if any, is executed. As the name implies, this usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity (including declarations in C++). Hence the claim that any C program can be written as a single **for** loop. (But don't try this at home. Such stunts are performed by trained professionals.)
2. The expression *test-exp* is evaluated following the rules of the **while** loop. If *test-exp* is nonzero (true), the loop statement is executed. An empty expression here is taken as **while** (1), that is, always true. If the value of *test-exp* is zero (false), the **for** loop terminates.
3. *increment-exp* advances one or more counters.
4. The expression *statement* (possibly empty) is evaluated and control returns to step 2.

If any of the optional elements are empty, appropriate semicolons are required:

```
for (;;) {           // same as for (; 1;)
    // loop forever
}
```



The C rules for **for** statements apply in C++. However, the *init-exp* in C++ can also be a declaration. The scope of a declared identifier extends through the enclosing loop. For example,

```
for (int i = 1; i < j; ++i)
{
    if (i ...) ...           // ok to refer to i here
}
if (i...)                   // illegal; i is now out of scope
```

Jump statements

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**.

break statements The syntax is

break;

A **break** statement can be used only inside an iteration (**while**, **do**, and **for** loops) or a **switch** statement. It terminates the iteration or **switch** statement. Since iteration and **switch** statements can be intermixed and nested to any depth, take care to ensure that your **break** exits from the correct loop or switch. The rule is that a **break** terminates the *nearest* enclosing iteration or **switch** statement.

continue statements The syntax is

continue;

A **continue** statement can be used only inside an iteration statement; it transfers control to the test condition for **while** and **do** loops, and to the increment expression in a **for** loop.

With nested iteration loops, a **continue** statement is taken as belonging to the *nearest* enclosing iteration.

goto statements The syntax is

goto label;

The **goto** statement transfers control to the statement labeled *label* (see "Labeled statements," page 98), which must be in the same function.



In C++, it is illegal to bypass a declaration having an explicit or implicit initializer unless that declaration is within an inner block that is also bypassed.

return statements Unless the function return type is **void**, a function body must contain at least one **return** statement with the following format:

```
return return-expression;
```

where *return-expression* must be of type **type** or of a type that is convertible to **type** by assignment. The value of the *return-expression* is the value returned by the function. An expression that calls the function, such as `func(actual-arg-list)`, is an rvalue of type **type**, not an lvalue:

```
t = func(arg);           // OK
func(arg) = t;          /* illegal in C; legal in C++ if return type of
                        func is a reference */
(func(arg))++;         /* illegal in C; legal in C++ if return type of
                        func is a reference */
```

The execution of a function call terminates if a **return** statement is encountered; if no **return** is met, execution “falls through,” ending at the final closing brace of the function body.

If the return type is **void**, the **return** statement can be written as

```
{
    ...
    return;
}
```

with no return expression; alternatively, the **return** statement can be omitted.

C++ specifics

C++ is basically a superset of C. This means that, generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs peculiar to C++. Some situations need special care. For example, the same function **func** declared twice in C with different argument types will invoke a duplicated name error. Under C++, however, **func** will be interpreted as an overloaded function—whether this is legal or not will depend on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. We first review these aspects of C++ that can be used independently of classes, then get into the specifics of classes and class mechanisms.

Referencing

Pointer referencing and dereferencing is discussed on page 85.

While in C you pass arguments only by value, in C++ you can pass arguments by value or by reference. C++ reference types, which are closely related to pointer types, create aliases for objects and let you pass arguments to functions by reference.

Simple references

The reference declarator can be used to declare references outside functions:

Note that type& var, type &var, and type & var are all equivalent.

```
int i = 0;
int &ir = i; // ir is an alias for i
ir = 2;     // same effect as i = 2
```

This creates the lvalue *ir* as an alias for *i*, provided that the initializer is the same type as the reference. Any operations on *ir* have precisely the same effect as operations on *i*. For example, *ir = 2* assigns 2 to *i*, and *&ir* returns the address of *i*.

Reference arguments

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir); // ir is type "reference to int"
...
int sum=3;
func1(sum);          // sum passed by value
func2(sum);          // sum passed by reference
```

The *sum* argument passed by reference can be changed directly by **func2**. On the other hand, **func1** gets a copy of the *sum* argument (passed by value), so *sum* itself cannot be altered by **func1**.

When an actual argument *x* is passed by value, the matching formal argument in the function receives a copy of *x*. Any changes to this copy within the function body are not reflected in the value of *x* itself. Of course, the function can *return* a value that could be used later to change *x*, but the function cannot directly alter a parameter passed by value.

The C method for changing *x* uses the actual argument *&x*, the address of *x*, rather than *x* itself. Although *&x* is passed by value, the function can access *x* through the copy of *&x* it receives. Even if the function does not need to change *x*, it is still useful (though subject to possibly dangerous side effects) to pass *&x*, especially if *x* is a large data structure. Passing *x* directly by value involves the wasteful copying of the data structure.

Compare the three implementations of the function **treble**:

```

Implementation 1    int treble_1(n)
                   {
                   return 3*n;
                   }
                   ...
                   int x, i = 4;
                   x = treble_1(i);           // x now = 12, i = 4
                   ...

Implementation 2    void treble_2(int* np)
                   {
                   *np = (*np)*3;
                   }
                   ...
                   treble_2(int &i);         // i now = 12

Implementation 3    void treble_3(int& n)     // n is a reference type
                   {
                   n = 3*n;
                   }
                   ...
                   treble_3(i);             // i now = 36

```

The formal argument declaration `type& t` (or equivalently, `type &t`) establishes `t` as type "reference to **type**." So, when **treble_3** is called with the real argument `i`, `i` is used to initialize the formal reference argument `n`. `n` therefore acts as an alias for `i`, so that `n = 3*n` also assigns `3 * i` to `i`.

If the initializer is a constant or an object of a different type than the reference type, Borland C++ creates a temporary object for which the reference acts as an alias:

```

int& ir = 6;        /* temporary int object created, aliased by ir, gets
                   value 6 */

float f;
int& ir2 = f;      /* creates temporary int object aliased by ir2; f
                   converted before assignment */

ir2 = 2.0          // ir2 now = 2, but f is unchanged

```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

Scope access operator

The scope access (or resolution) operator `::` (two semicolons) lets you access a global (or file duration) name even if it is hidden by a local redeclaration of that name (see page 27 for more on scope):

This code also works if the "global" i is a file-level static.

```
int i;                // global i
...
void func(void);
{
    int i=0;          // local i hides global i
    i = 3;           // this i is the local i
    ::i = 4;         // this i is the global i
    printf ("%d",i); // prints out 3
}
```

The `::` operator has other uses with class types, as discussed throughout this chapter.

The new and delete operators

The **new** and **delete** operators offer dynamic storage allocation and deallocation, similar but superior to the standard library functions in the **malloc** and **free** families (see the *Library Reference*).

A simplified syntax is

```
pointer-to-name = new name <name-initializer>;
delete pointer-to-name;
```

name can be of any type except "function returning..." (however, pointers to functions are allowed).

new tries to create an object of type *name* by allocating (if possible) **sizeof(name)** bytes in *free store* (also called the heap). The storage duration of the new object is from the point of creation until the operator **delete** kills it by deallocating its memory, or until the end of the program.

If successful, **new** returns a pointer to the new object. A null pointer indicates a failure (such as insufficient or fragmented heap memory). As with **malloc**, you need to test for null before trying to access the new object (unless you use a new-handler; see the following section for details). However, unlike **malloc**, **new** calculates the size of *name* without the need for an explicit **sizeof**

operator. Further, the pointer returned is of the correct type, "pointer to *name*," without the need for explicit casting.

new, being a keyword, doesn't need a prototype.

```
name *nameptr;      // name is any nonfunction type
...
if (!(nameptr = new name)) {
    errmsg("Insufficient memory for name");
    exit (1);
}
// use *nameptr to initialize new name object
...
delete nameptr;    // destroy name and deallocate sizeof(name) bytes
```

Handling errors

You can define a function that will be called if the **new** operator fails (returns 0). To tell the **new** operator about the new-handler function, call **set_new_handler** and supply a pointer to the new-handler. The prototype for **set_new_handler** is as follows (from `new.h`):

```
void (*set_new_handler( void (*)() ))();
```

set_new_handler returns the old new-handler, and changes the function **_new_handler** so that it, in turn, points to the new-handler that you define.

The operator `new` with arrays

If *name* is an array, the pointer returned by **new** points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied (although the left-most dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];    // OK
mat_ptr = new int[n][10][12];   // OK
mat_ptr = new int[3][][12];     // illegal
mat_ptr = new int[][10][12];    // illegal
```

Although the first array dimension may be a variable, all following dimensions must be constants.

The operator `delete` with arrays

You must use the syntax "delete [] *expr*" when deleting an array. In C++ 2.1, the array dimension should not be specified within the brackets:


```

char * p;

void func()
{
    p = new char[10];    // allocate 10 chars
    delete[] p;        // delete 10 chars
}

```

C++ 2.0 code required the array size. In order to allow 2.0 code to compile, Borland C++ issues a warning and simply ignores any size that is specified. For example, if the preceding example reads `delete[10] p` and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

With Borland C++, the `[]` is actually only required when the array element is a class with a destructor. But it is good programming practice to always tell the compiler that an array is being deleted.

The `::operator`

`new`

When used with non-class objects, **new** works by calling a standard library routine, the global `::operator new`. With class objects of type *name*, a specific operator called `name::operator new` can be defined. **new** applied to class *name* objects invokes the appropriate `name::operator new` if present; otherwise, the standard `::operator new` is used.

Initializers with the `new` operator

The optional initializer is another advantage **new** has over **malloc** (although **calloc** does clear its allocations to zero). In the absence of explicit initializers, the object created by **new** contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the *default constructor* (see page 126). The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

Classes

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that *inherit* the members of one or more *base* (or parent) classes.

In C++, structures and unions are considered as classes with certain access defaults.

A simplified, “first-look” syntax for class declarations is

```
class-key class-name <: base-list> { <member-list> }
```

class-key is one of **class**, **struct**, or **union**.

The optional *base-list* lists the base class or classes from which the class *class-name* will derive (or *inherit*) objects and methods. If any base classes are specified, the class *class-name* is called a derived class (see page 120, “Base and derived class access”). The *base-list* has default and optional overriding *access specifiers* that can modify the access rights of the derived class to members of the base classes (see page 118, “Member access control”).

The optional *member-list* declares the class members (data and functions) of *class-name* with default and optional overriding access specifiers that may affect which functions can access which members.

Class names

class-name is any identifier unique within its scope. With structures, classes, and unions, *class-name* can be omitted (see “Untagged structures and typedefs,” page 66.)

Class types

The declaration creates a unique type, class type *class-name*. This lets you declare further *class objects* (or *instances*) of this type, and objects derived from this type (such as pointers to, references to, arrays of *class-name*, and so on):

```
class X { ... };  
X x, &xr, *xpnr, xarray[10];  
/* four objects: type X, reference to X, pointer to X and array of  
X*/
```

```

struct Y { ... };
Y y, &yr, *yptr, yarray[10];
// C would have
// struct Y y, *yptr, yarray[10];

union Z { ... };
Z z, &zr, *zptr, zarray[10];
// C would have
// union Z z, *zptr, zarray[10];

```

Note the difference between C and C++ structure and union declarations: The keywords **struct** and **union** are essential in C, but in C++ they are needed only when the class names, **Y** and **Z**, are hidden (see the following section).

Class name scope

The scope of a class name is local, with some tricks peculiar to classes. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can only be referred to using the *elaborated type specifier*. This means that the class key, **class**, **struct**, or **union** must be used with the class name. For example,

```

struct S { ... };

int S(struct S *Sptr);

void func(void)
{
    S t;           // ILLEGAL declaration: no class key
                  // and function S in scope
    struct S s;   // OK: elaborated with class key
    S(&s);        // OK: this is a function call
}

```

C++ also allows an incomplete class declaration:

```

class X; // no members, yet!

```

Incomplete declarations permit certain references to class name **X** (usually references to pointers to class objects) before the class has been fully defined (see "Structure member declarations," page 66). Of course, you must make a complete class declaration with members before you can define and use class objects.

Class objects

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including member and friend functions, and the redefinition of standard functions and operators when used with objects of a certain class. Redefined functions and operators are said to be *overloaded*. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called *member functions* for that class. C++ offers a mechanism whereby the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

Class member list

The optional *member-list* is a sequence of data declarations (of any type, including enumerations, bit fields and other classes) and function declarations and definitions, all with optional storage class specifiers and access modifiers. The objects thus defined are called *class members*. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

Member functions

A function declared without the **friend** specifier is known as a *member function* of the class. Functions declared with the **friend** modifier are called *friend functions*.

The same name can be used to denote more than one function, provided that they differ in argument type or number of arguments.

The keyword this

Nonstatic member functions operate on the class type object with which they are called. For example, if x is an object of class **X** and **f** is a member function of **X**, the function call $x.f()$ operates on x . Similarly, if $xptr$ is a pointer to an **X** object, the function call $xptr->f()$ operates on $*xptr$. But how does **f** know which x it is operating on? C++ provides **f** with a pointer to x called **this**. **this** is

passed as a hidden argument in all calls to nonstatic member functions.

The keyword **this** is a local variable available in the body of any nonstatic member function. **this** does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. If **x.f(y)** is called, for example, where *y* is a member of **X**, **this** is set to *&x* and *y* is set to **this->y**, which is equivalent to *x.y*.

Inline functions

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an *inline* function.

Borland C++ can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an *inline expansion* of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The *inline* specifier is a request (or hint) to the compiler that you would welcome an inline expansion. As with the **register** storage class specifier, the compiler may or may not take the hint!

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the *operator functions* that implement overloaded operators. For example, the following class declaration:

```
int i;                                // global int

class X {
public:
    char* func(void) { return i; } // inline by default
    char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func is defined “outside” the class with an explicit inline specifier. The *i* returned by **func** is the **char*** *i* of class **X**—see the section on member scope starting on page 116.

Static members

The storage class specifier **static** can be used in class declarations of data and function members. Such members are called *static members* and have distinct properties from nonstatic members. With nonstatic members, a distinct copy “exists” for each object in its class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If x is a static member of class **X**, it can be referenced as **X::x** (even if objects of class **X** haven’t been created yet). It is still possible to access x using the normal member access operators. For example, $y.x$ and $yptr->x$, where y is an object of class **X** and $yptr$ is a pointer to an object of class **X**, although the expressions y and $yptr$ are *not* evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
    int member_int;
public:
    static void func(int i, X* ptr);
};

void g(void);
{
    X obj;
    func(1, &obj);      // error unless there is a global func()
                       // defined elsewhere

    X::func(1, &obj);  // calls the static func() in X
                       // OK for static functions only
    obj.func(1, &obj); // so does this (OK for static and
                       // nonstatic functions)
}
```

Since a static member function can be called with no particular object in mind, it has no **this** pointer. A consequence of this is that a static member function cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, **func** might be defined as follows:

```
void X::func(int i, X* ptr)
{
    member_int = i;      // which object does member_int
                       // refer to? Error
    ptr->member_int = i; // OK: now we know!
}
```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members obey the usual class member access rules, except they can be initialized.

```
class X {
    ...
    static int x;
    ...
};

int X::x = 1;
```

The main use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- reduce the number of visible global names
- make obvious which static objects logically belong to which class
- permit access control to their names

Member scope

The expression **X::func()** in the example on page 114 uses the class name **X** with the scope access modifier to signify that **func**, although defined “outside” the class, is indeed a member function of **X**, and it exists within the scope of **X**. The influence of **X::** extends into the body of the definition. This explains why the *i* returned by **func** refers to **X::i**, the *char** *i* of **X**, rather than the global **int** *i*. Without the **X::** modifier, the function **func** would represent an ordinary non-class function, returning the global **int** *i*.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class **X** can be referenced using the selection operators `.` and `->` (as with C structures). Member functions can also be called using the selection operators (see also “The keyword **this**,” page 113). For example,

```
class X {
public:
    int i;
    char name[20];
    X *ptr1;
    X *ptr2;
    void Xfunc(char*data, X* left, X* right); // define elsewhere
};
void f(void);
{
    X x1, x2, *xptr=&x1;
    x1.i = 0;
    x2.i = x1.i;
    xptr->i = 1;
    x1.Xfunc("stan", &x2, xptr);
}
```

If *m* is a member or base member of class **X**, the expression `X::m` is called a *qualified name*; it has the same type as *m*, and it is an lvalue only if *m* is an lvalue. A key point is that even if the class name **X** is hidden by a non-type name, the qualified name `X::m` will access the correct class member, *m*.

Class members cannot be added to a class by another section of your program. The class **X** cannot contain objects of class **X**, but can contain pointers or references to objects of class **X** (note the similarity with C’s structure and union types).

Nested types In C++ 2.1, even tag or typedef names declared inside a class lexically belong to the scope of that class. Such names can in general be accessed only using the `xxx::yyy` notation, except when in the scope of the appropriate class.

A class declared within another class is called a *nested class*. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa).



Classes can be nested in this way to an arbitrary level. For example:


```

struct outer
{
    typedef int t; // 'outer::t' is a typedef name
    struct inner // 'outer::inner' is a class
    {
        static int x;
    };
    static int x;
    int f();
};

int outer::x; //define static data member
int outer::f()
{
    t x; // 't' visible directly here
    return x;
}

int outer::inner::x; //define static data member
outer::t x; // have to use 'outer::t' here

```

With C++ 2.0, any tags or typedef names declared inside a class actually belong to the global (file) scope. For example:

```

struct foo
{
    enum bar { x }; // 2.0 rules: 'bar' belongs to file scope
                  // 2.1 rules: 'bar' belongs to 'foo' scope
};

bar x;

```

The preceding fragment compiles without errors. But, because the code is illegal under the 2.1 rules, a warning is issued as follows:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

Member access control

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

- public** The member can be used by any function.
- private** The member can be used only by member functions and friends of the class in which it is declared.

Friend function declarations are not affected by access specifiers (see "Friends of classes," page 122).

protected Same as for **private**, but additionally, the member can be used by member functions and friends of classes *derived* from the declared class, but only in objects of the derived type. (Derived classes are explained in the next section.)

Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
    int i;    // X::i is private by default
    char ch; // so is X::ch
public:
    int j;    // next two are public
    int k;
protected:
    int l;    // X::l is protected
};

struct Y {
    int i;    // Y::i is public by default
private:
    int j;    // Y::j is private
public:
    int k;    // Y::k is public
};

union Z {
    int i;    // public by default; no other choice
    double d;
};
```

The access specifiers can be listed and grouped in any convenient sequence. You can save a little typing effort by declaring all the private members together, and so on.

Base and derived class access

When you declare a derived class **D**, you list the base classes **B1**, **B2**, ... in a comma-delimited *base-list*:

```
class-key D : base-list { <member-list> }
```

Since a base class can itself be a derived class, the access attribute question is recursive: You backtrack until you reach the basest of the base classes, those that do not inherit.

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) **D** can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by **D**? **D** may want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the *base-list*.

When declaring **D**, you can use the access specifier **public**, **protected**, or **private** in front of the classes in the *base-list*:

```
class D : public B1, private B2, ... {  
    ...  
}
```

Unions cannot have base classes, and unions cannot be used as base classes.

These modifiers do not alter the access attributes of base members as viewed by the base class, though they *can* alter the access attributes of base members as viewed by the derived class.

The default is **private** if **D** is a **class** declaration, and **public** if **D** is a **struct** declaration.

The derived class inherits access attributes from a base class as follows:

public base class: **public** members of the base class are **public** members of the derived class. **Protected** members of the base class are **protected** members of the derived class. **Private** members of the base class remain **private** to the base class.

protected base class: Both **public** and **protected** members of the base class are **protected** members of the derived class. **Private** members of the base class remain **private** to the base class.

private base class: Both **public** and **protected** members of the base class are **private** members of the

derived class. **Private** members of the base class remain **private** to the base class.

In both cases, note carefully that **private** members of a base class are, and remain, inaccessible to member functions of the derived class *unless friend* declarations are explicitly declared in the base class granting access. For example,

```
class X : A {           // default for class is private A
...
}
/* class X is derived from class A */

class Y : B, public C { // override default for C
...
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */

struct S : D {         // default for struct is public D
...                   /* struct S is derived from D */
}

struct T : private D, E { // override default for D
                        // E is public by default
...
}
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
```

The effect of access specifiers in the base list can be adjusted by using a *qualified-name* in the public or protected declarations in the derived class. For example,

```
class B {
    int a;           // private by default
public:
    int b, c;
    int Bfunc(void);
};

class X : private B { // a, b, c, Bfunc are now private in X
    int d;           // private by default, NOTE: a is not
                    // accessible in X
public:
    B::c;           // c was private, now is public
    int e;
    int Xfunc(void);
};

int Efunc(X& x);    // external to B and X
```

The function **Efunc** can use only the public names *c*, *e*, and **Xfunc**.

The function **Xfunc** is in **X**, which is derived from **private B**, so it has access to

- The “adjusted-to-public” *c*
- The “private-to-**X**” members from **B**: *b* and **Bfunc**
- **X**’s own private and public members: *d*, *e*, and **Xfunc**

However, **Xfunc** cannot access the “private-to-**B**” member, *a*.

Virtual base classes

With multiple inheritance, a base class can’t be specified more than once in a derived class:

```
class B { ...};  
class D : B, B { ... }; // Illegal
```

However, a base class can be indirectly passed to the derived class more than once:

```
class X : public B { ... }  
class Y : public B { ... }  
  
class Z : public X, public Y { ... } // OK
```

In this case, each object of class **Z** will have two sub-objects of class **B**. If this causes problems, the keyword **virtual** can be added to a base class specifier. For example,

```
class X : virtual public B { ... }  
class Y : virtual public B { ... }  
class Z : public X, public Y { ... }
```

B is now a virtual base class, and class **Z** has only one sub-object of class **B**.

Friends of classes

A **friend F** of a class **X** is a function or class that, although not a member function of **X**, has full access rights to the private and protected members of **X**. In all other respects, **F** is a normal function with respect to scope, declarations, and definitions.

Since **F** is not a member of **X**, it is not in the scope of **X** and it cannot be called with the $x.F$ and $xptr->F$ selector operators (where x is an **X** object, and $xptr$ is a pointer to an **X** object).

If the specifier **friend** is used with a function declaration or definition within the class **X**, it becomes a friend of **X**.

Friend functions defined within a class obey the same inline rules as member functions (see "Inline functions," page 114). Friend functions are not affected by their position within the class or by any access specifiers. For example,

```
class X {
    int i;                                // private to X
    friend void friend_func(X*, int);
    /* friend_func is not private, even though it's declared in the
       private section */
public:
    void member_func(int);
};

/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }

X xobj;

/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class **Y** into friends of class **X** with a single declaration:

```
class Y;                                // incomplete declaration
class X {
    friend Y;
    int i;
    void member_funcX();
};

class Y; {                                // complete the declaration
    void friend_X1(X&);
    void friend_X2(X*);
    ...
};
```

The functions declared in **Y** are friends of **X**, although they have no **friend** specifiers. They can access the private members of **X**, such as i and **member_funcX**.

It is also possible for an individual member function of class **X** to be a friend of class **Y**:

```
class X {
    ...
    void member_funcX();
}

class Y {
    int i;
    friend void X::member_funcX();
    ...
};
```

Class friendship is not transitive: **X** friend of **Y** and **Y** friend of **Z** does not imply **X** friend of **Z**. However, friendship *is* inherited.

Constructors and destructors

There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features.

1. They do not have return value declarations (not even **void**).
2. They cannot be inherited, though a derived class can call the base class's constructors and destructors.
3. Constructors, like most C++ functions, can have default arguments or use member initialization lists.
4. Destructors can be **virtual**, but constructors cannot.
5. You can't take their addresses.

```
int main(void)
{
    ...
    void *ptr = base::base;    // illegal
    ...
}
```

6. Constructors and destructors can be generated by Borland C++ if they haven't been explicitly defined; they are also invoked on many occasions without explicit calls in your

program. Any constructor or destructor generated by the compiler will be **public**.

7. You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```
{
...
X *p;
...
p->X::~X();           // legal call of destructor
X::X();               // illegal call of constructor
...
}
```

8. The compiler automatically calls constructors and destructors when defining and destroying objects.
9. Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.
10. An object with a constructor or destructor cannot be used as a member of a union.

If a class **X** has one or more constructors, one of them is invoked each time you define an object *x* of class **X**. The constructor creates *x* and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

Constructors

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the main function is called. When the pragma startup directive is used to install a function prior to the main function, global variable constructors are called prior to the startup functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X
{
public:
    X();    // class X constructor
};
```

A class **X** constructor cannot take **X** as an argument:

```
class X {
...
public:
    X(X);           // illegal
}
```

The parameters to the constructor can be of any type except that of the class of which it is a member. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the *copy constructor*. A constructor that accepts no parameters is called the *default constructor*. We discuss the default constructor next; the description of the copy constructor starts on page 127.

Constructor defaults

The default constructor for class **X** is one that takes no arguments; it usually has the form `X::X()`. If no user-defined constructors exist for a class, Borland C++ generates a default constructor. On a declaration such as `X x`, the default constructor creates the object `x`.

Important! Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero `int`. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor `X::X()` takes *no* arguments and must not be confused with, say, `X::X(int = 0)`, which can be called with no arguments as a default constructor, or can take an argument.

Take care to avoid ambiguity in calling constructors. In the following case, the two default constructors could become ambiguous:

```
class X
{
public:
    X();
    X(int i = 0);
};

main()
{
    X one(10); // OK; uses X::X(int)
    X two;     // illegal; ambiguous whether to call X::X() or
              // X::X(int = 0)
    ...
    return 0;
}
```

The copy constructor

A copy constructor for class **X** is one that can be called with a single argument of type `X: X(const X&)` or `X: X(const X&, int = 0)`. Default arguments are also allowed in a copy constructor. Copy constructors are invoked when copying a class object, typically when you declare with initialization by another class object: `X x = y`. Borland C++ generates a copy constructor for class **X** if one is needed and none is defined in class **X**.

Overloading constructors

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X
{
    int   integer_part;
    double double_part;
public:
    X(int i)   { integer_part = i; }
    X(double d) { double_part = d; }
};

main()
{
    X one(10); // invokes X::X(int) and sets integer_part to 10
    X one(3.14); // invokes X::X(double) setting double_part
```

```
...
    return 0;
}
```

Order of calling constructors

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y(); // base class constructor
X(); // derived class constructor
```

For the case of multiple base classes:

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```
Y(); // base class constructors come first
Z();
X();
```

Constructors for virtual base classes are invoked before any non-virtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first so that the virtual base class may be properly constructed. The code

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z(); // virtual base class initialization
Y(); // non-virtual base class
X(); // derived class
```

Or for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base(); // virtual base class highest in hierarchy
// base is only constructed once
base2(); // non-virtual base of virtual base level2
// must be called to construct level2
level2(); // virtual base class
base2(); // non-virtual base of level1
level1(); // other non-virtual base
toplevel();
```

In the event that a class hierarchy contains multiple instances of a virtual base class, that base class is only constructed once. If, however, there exist both virtual and non-virtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each non-virtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

Class initialization

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be of the type of the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```

class X
{
    int i;
public:
    X();           // function bodies omitted for clarity
    X(int x);
    X(const X&);
};

main()
{
    X one;        // default constructor invoked
    X two(1);     // constructor X::X(int) is used
    X three = 1; // calls X::X(int)
    X four = one; // invokes X::X(const X&) for copy
    X five(two); // calls X::X(const X&)
}

```

The constructor can assign values to its members in two ways. It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```

class X
{
    int a, b;
public:
    X(int i, int j) { a = i; b = j }
};

```

Or it can use an initializer list prior to the function body:

```

class X
{
    int a, b;
public:
    X(int i, int j) : a(i), b(j) {}
};

```

In both cases, an initialization of `X x(1, 2)` assigns a value of 1 to `x::a` and 2 to `x::b`. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

*Base class constructors must be declared as either **public** or **protected** to be called from a derived class.*

```

class base1
{
    int x;
public:
    base1(int i) { x = i; }
};

class base2
{

```

```

    int x;
public:
    base2(int i) : x(i) {}
};

class top : public base1, public base2
{
    int a, b;
public:
    top(int i, int j) : base1(i*5), base2(j+1), a(i) { b = j;}
};

```

With this class hierarchy, a declaration of `top one(1, 2)` would result in the initialization of **base1** with the value 5 and **base2** with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```

class X
{
    int a, b;
public:
    X(int i, j) : a(i), b(a+j) {}
};

```

With this class, a declaration of `X x(1,1)` results in an assignment of 1 to `x::a` and 2 to `x::b`.

Base class constructors are called prior to the construction of any of the derived classes members. The values of the derived class can't be changed and then have an affect on the base class's creation.

```

class base
{
    int x;
public:
    base(int i) : x(i) {}
};

class derived : base
{
    int a;
public:
    derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                        // passed an uninitialized a
};

```

With this class setup, a call of derived `d(1)` will *not* result in a value of 10 for the base class member `x`. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
    ...
}
```

Destructors

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
class X
{
public:
    ~X(); // destructor for class X
};
```

If a destructor is not explicitly defined for a class, the compiler will generate one.

When destructors are invoked

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see page 128).

atexit, #pragma exit, and destructors

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the main function, are destroyed as they go out of scope. The order of execution at the end of a Borland C++ program in these regards is as follows:

- **atexit** functions are executed in the order they were inserted.
- **#pragma exit** functions are executed in the order of their priority codes.
- Destructors for global variables are called.

exit and destructors

When you call **exit** from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

abort and destructors

If you call **abort** anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are only necessary for objects allocated a specific address through calls to **new**.

```
class X {
    ...
    ~X();
    ...
};

void* operator new(size_t size, void *ptr)
{
    return ptr;
}

char buffer[sizeof(X)];

main()
{
```



```

X* pointer = new X;
X* exact_pointer;

exact_pointer = new(&buffer) X; // pointer initialized at
                               // address of buffer

...

delete pointer;                // delete used to destroy pointer
exact_pointer->X::~~X();        // direct call used to deallocate
}

```

Virtual destructors

A destructor can be declared as virtual. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a virtual destructor is itself virtual.

```

class color
{
public:
    virtual ~color();    // virtual destructor for color
};

class red : public color
{
public:
    ~red();              // destructor for red is also virtual
};

class brightred: public red
{
public:
    ~brightred();       // brightred's destructor also virtual
};

```

The previously listed classes and the following declarations

```

color *palette[3];

palette[0] = new red;
palette[1] = new brightred;
palette[2] = new color;

```

will produce these results

```

delete palette[0];
// The destructor for red is called followed by the
// destructor for color.

delete palette[1];
// The destructor for brightred is called, followed by ~red

```

```

// and ~color.
delete palette[2];
// The destructor for color is invoked.

```

However, in the event that no destructors were declared as virtual, `delete palette[0]`, `delete palette[1]`, and `delete palette[2]` would all call only the destructor for class `color`. This would incorrectly destruct the first two elements, which were actually of type `red` and `brightred`.

Overloaded operators

C++ lets you redefine the action of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators on page 79 can be overloaded except for

```
. .* :: ?:
```

The preprocessing symbols `#` and `##` also cannot be overloaded.

The keyword **operator** followed by the operator symbol is called the *operator function name*; it is used like a normal function name when defining the new (overloaded) action of the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function can't alter the number of arguments or the precedence and associativity rules (Table 2.10 on page 76) applying to normal operator use. Consider the class *complex*:

*This class was invented for illustrative purposes only. It isn't the same as the class **complex** in the run-time library.*

```

class complex {
    double real, imag;           // private by default
public:
    ...
    complex() { real = imag = 0; } // inline constructor
    complex(double r, double i = 0) { // another one
        real = r; imag = i;
    }
    ...
}

```

We could easily devise a function for adding complex numbers, say,

```
complex AddComplex(complex c1, complex c2);
```

but it would be more natural to be able to write:

```
complex c1(0,1), c2(1,0), c3;  
c3 = c1 + c2;
```

than

```
c3 = AddComplex(c1, c2);
```

The operator **+** is easily overloaded by including the following declaration in the class *complex*:

```
friend complex operator +(complex c1, complex c2);
```

and defining it (possibly inline) as:

```
complex operator +(complex c1, complex c2)  
{  
    return complex(c1.real + c2.real, c1.imag + c2.imag);  
}
```

Operator functions

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2); // same as c3 = c1 + c2
```

Apart from **new** and **delete**, which have their own rules (see the next sections), an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions **=**, **()**, **[]** and **->** must be nonstatic member functions.

Overloaded operators and inheritance

With the exception of the assignment function operator **=()** (see "Overloading the assignment operator **=**" on page 139), all overloaded operator functions for class **X** are inherited by classes derived from **X**, with the standard resolution rules for overloaded functions. If **X** is a base class for **Y**, an overloaded operator function for **X** may possibly be further overloaded for **Y**.

Overloading **new** and **delete**

The type `size_t` is defined in `stdlib.h`

The operators **new** and **delete** can be overloaded to provide alternative free storage (heap) memory-management routines. A user-defined operator **new** must return a **void*** and must have a **size_t** as its first argument. A user-defined operator **delete** must have a **void** return type and **void*** as its first argument; a second argument of type **size_t** is optional. For example,

```
#include <stdlib.h>

class X {
    ...
public:
    void* operator new(size_t size) { return newalloc(size); }
    void operator delete(void* p) { newfree(p); }
    X() { /* initialize here */ }
    X(char ch) { /* and here */ }

    ~X() { /* clean up here */ }
    ...
};
```

The *size* argument gives the size of the object being created, and **newalloc** and **newfree** are user-supplied memory allocation and deallocation functions. Constructor and destructor calls for objects of class **X** (or objects of classes derived from **X** that do not have their own overloaded operators **new** and **delete**) will invoke the matching user-defined **X::operator new()** and **X::operator delete()**, respectively.

The **X::operator new** and **X::operator delete** operator functions are static members of **X** whether explicitly declared as **static** or not, so they cannot be virtual functions.

The standard, predefined (global) **new** and **delete** operators can still be used within the scope of **X**, either explicitly with the global scope operator (**::operator new** and **::operator delete**), or implicitly when creating and destroying non-**X** or non-**X**-derived class objects. For example, you could use the standard **new** and **delete** when defining the overloaded versions:

```
void* X::operator new(size_t s)
{
    void* ptr = new char[s]; // standard new called
    ...
    return ptr;
}
```

```

void X::operator delete(void* ptr)
{
    ...
    delete (void*) ptr;    // standard delete called
}

```

The reason for the *size* argument is that classes derived from **X** inherit the **X::operator new**. The size of a derived class object may well differ from that of the base class.

Overloading unary operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a non-member function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either *x.operator@()* or *operator@(x)*, depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.



Under C++ 2.0, an overloaded operator++ or — is used for both prefix and postfix uses of the operator. For example:

```

struct foo
{
    operator::();
    operator--();
}

x;

void func()
{
    x++;    // calls x.operator++()
    ++x;   // calls x.operator++()

    x--;    // calls x.operator--()
    --x;   // calls x.operator--()
}

```

With C++ 2.1, when an **operator++** or **operator—** is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix operator ++ or operator —. You can only overload a postfix operator++ or operator— by defining it as a member function taking an *int* parameter or as a nonmember function taking one class and one *int* parameter. For example add the following lines to the previous code:

```
operator++(int);
operator--(int);
```

When only the prefix version of an **operator++** or **operator--** is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in the following warnings:

```
Warning: Overloaded prefix 'operator ++' used as a postfix operator
in function func()
```

```
Warning: Overloaded prefix 'operator --' used as a postfix operator
in function func()
```

Overloading binary operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually **friend**) taking two arguments. If **@** represents a binary operator, $x@y$ can be interpreted as either $x.\mathbf{operator}@(y)$ or $\mathbf{operator}@(x,y)$, depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

Overloading the assignment operator =

The assignment operator **=** can be overloaded by declaring a nonstatic member function. For example,

```
class String {
    ...
    String& operator = (String& str);
    ...
    String (String&);
    ~String();
}
```

This code, with suitable definitions of **String::operator =()**, allows string assignments $str1 = str2$, just like other languages. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any class **X**, there is no user-defined operator **=**, the operator **=** is defined by default as a member-by-member assignment of the members of class **X**:

```
X& X::operator = (const X& source)
{
```

```
    // memberwise assignment
}
```

Overloading the function call operator ()

The function call

primary-expression (<*expression-list*>)

is considered a binary operator with operands *primary-expression* and *expression-list* (possibly empty). The corresponding operator function is **operator()**. This function can be user-defined for a class **X** (and any derived classes) only by means of a nonstatic member function. A call *x*(*arg1*, *arg2*), where *x* is an object of class **X**, is interpreted as *x.operator()*(*arg1*,*arg2*).

Overloading the subscript operator []

Similarly, the subscripting operation

primary-expression [*expression*]

is considered a binary operator with operands *primary-expression* and *expression*. The corresponding operator function is **operator[]**; this can be user-defined for a class **X** (and any derived classes) only by means of a nonstatic member function. The expression *x*[*y*], where *x* is an object of class **X**, is interpreted as *x.operator[]*(*y*).

Overloading the class member access operator ->

Class member access using

primary-expression -> *expression*

is considered a unary operator. The function **operator->** must be a nonstatic member function. The expression *x->m*, where *x* is a class **X** object, is interpreted as (*x.operator->*())->*m*, so that the function **operator->**() must either return a pointer to a class object or return an object of a class for which **operator->** is defined.

Virtual functions

Virtual functions can only be member functions.

Virtual functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword

to declare a virtual function in a base class, then redefine it in any derived class, even if the number and type of arguments are the same. The redefined function is said to *override* the base class function. You can also declare the functions `int Base::Fun(int)` and `int Derived::Fun(int)` even when they are not virtual. The base class version is available to derived class objects via scope override. If they are virtual, only the function associated with the actual type of the object is available.

With virtual functions, you cannot change just the function type. It is illegal, therefore, to redefine a virtual function so that it differs only in the return type. If two functions with the same name have different arguments, C++ considers them different, and the virtual function mechanism is ignored.

If a base class **B** contains a virtual function **vf**, and class **D**, derived from **B**, contains a function **vf** of the same type, then if **vf** is called for an object *d* or **D**, the call made is `D::vf`, even if the access is via a pointer or reference to **B**. For example,

```
struct B {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    void f();
};
class D : public B {
    virtual void vf1(); // virtual specifier is legal but redundant
    void vf2(int);     // not virtual, since it's using a different
                        // arg list
    char vf3();        // Illegal: return-type-only change!
    void f();
};
void extf()
{
    D d;               // declare a D object
    B* bp = &d;       // standard conversion from D* to B*
    bp->vf1();         // calls D::vf1
    bp->vf2();         // call B::vf2 since D's vf2 has different args
    bp->f();           // calls B::f (not virtual)
}
```

The overriding function **vf1** in **D** is automatically virtual. The **virtual** specifier *can* be used with an overriding function declaration in the derived class, but its use is redundant.

The interpretation of a virtual function call depends on the type of the object for which it is called; with non-virtual function calls, the

interpretation depends only on the type of the pointer or reference denoting the object for which it is called.

➡ Virtual functions must be members of some class, but they cannot be static members. A virtual function can be a **friend** of another class.

A virtual function in a base class, like all member functions of a base class, must be defined or, if not defined, declared *pure*:

```
class B {  
    virtual void vf(int) = 0;    // = 0 means 'pure'
```

In a class derived from such a base class, each pure function must be defined or redeclared as pure (see the next section, “Abstract classes”).

If a virtual function is defined in the base it need not necessarily be redefined in the derived class. Calls will simply call the base function.

Virtual functions exact a price for their versatility: Each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at run time (late binding).

Abstract classes

An *abstract class* is a class with at least one pure virtual function. A virtual function is specified as pure by using the pure-specifier.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {    // abstract class  
    point center;  
    ...  
public:  
    where() { return center; }  
    move(point p) { center = p; draw(); }  
    virtual void rotate(int) = 0; // pure virtual function  
    virtual void draw() = 0;      // pure virtual function  
    virtual void hilite() = 0;    // pure virtual function  
    ...
```

```

}
shape x;          // ERROR: attempted creation of an object of
                  // an abstract class
shape* sptr;     // pointer to abstract class is OK
shape f();       // ERROR: abstract class cannot be a return
                  // type
int g(shape s);  // ERROR: abstract class cannot be a
                  //function argument type
shape& h(shape&); // reference to abstract class as return
                  // value or function argument is OK

```

Suppose that **D** is a derived class with the abstract class **B** as its immediate base class. Then for each pure virtual function **pvf** in **B**, if **D** doesn't provide a definition for **pvf**, **pvf** becomes a pure member function of **D**, and **D** will also be an abstract class.

For example, using the class `shape` previously outlined,

```

class circle : public shape { // circle derived from
                              // abstract class
    int radius;               // private
public:
    void rotate(int) { }      // virtual function defined:
                              // no action to rotate a
                              // circle
    void draw();              // circle::draw must be
                              // defined somewhere
}

```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a run-time error.

C++ scope

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement may appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

Class scope

The name *M* of a member of a class **X** has class scope “local to **X**,” it can only be used in the following situations:

- In member functions of **X**
- In expressions such as *x.M*, where *x* is an object of **X**
- In expressions such as *xptr->M*, where *xptr* is a pointer to an object of **X**
- In expressions such as **X::M** or **D::M**, where **D** is a derived class of **X**
- In forward references within the class of which it is a member.

Names of functions declared as friends of **X** are not members of **X**; their names simply have enclosing scope.

Hiding

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: **X::M**. A hidden file scope (global) name can be referenced with the unary operator **::**; for example, **::g**. A class name **X** can be hidden by the name of an object, function, or enumerator declared within the scope of **X**, regardless of the order in which the names are declared. However, the hidden class name **X** can still be accessed by prefixing **X** with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name *x* is immediately after its complete declaration but before its initializer, if one exists.

C++ scoping rules summary

The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

1. The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
2. If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.

3. If the name is used outside any function and class, or is prefixed by the unary scope access operator `::`, and if the name is not qualified by the binary `::` operator or the member selection operators `.` and `->`, then the name must be a global object, function, or enumerator.
4. If the name *n* appears in any of the forms `X::n`, `x.n` (where *x* is an object of **X** or a reference to **X**), or `ptr->n` (where *ptr* is a pointer to **X**), then *n* is the name of a member of **X** or the member of a class from which **X** is derived.
5. Any name not covered so far that is used in a static member function must be declared in the block in which it occurs or in an enclosing block, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks and global declarations of *n*. Names in different scopes are not overloaded.
6. Any name not covered so far that is used in a nonstatic member function of class **X** must be declared in the block in which it occurs or in an enclosing block, be a member of class **X** or a base class of **X**, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks, members of the function's class, and global declarations of *n*. The declaration of a member name hides declarations of the same name in base classes.
7. The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a non-defining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
8. A constructor initializer (see *ctor-initializer* in the class declarator syntax, Table 2.3 on page 37) is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

Templates

For a discussion of templates in the container class library see Chapter 6, page 224.

Templates, also called *generics* or *parameterized types*, allow you to construct a family of related functions or classes. In this section, we'll introduce the basic concept then some specific points.

Syntax:

Template-declaration:
template < template-argument-list > declaration

template-argument-list:
template-argument
template-argument-list, template argument

template-argument:
type-argument
argument-declaration

type-argument:
class identifier

template-class-name:
template-name < template-arg-list >

template-arg-list:
template-arg
template-arg-list, template-arg

template-arg:
expression
type-name

Function templates

Consider a function **max**(*x,y*) that returns the larger of its two arguments. *x* and *y* can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters *x* and *y* to be declared at compile time. Without using templates, many overloaded versions of **max()** are required, one for each data type to be supported, even though the code for each version is essentially identical. Each version compares the arguments and returns the larger. For example,

```
int max(int x, int y)
{
    return (x > y) ? x : y;
}

long max(long x, long y)
{
    return (x > y) ? x : y;
}

⋮
```

followed by other versions of **max**.

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the **#define** circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of **max(x,y)** to compare compatible types. Unfortunately, using the macro allows a comparison between an **int** and a **struct**, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be:

```
class Foo
{
public:
    int max(int, int); // Results in syntax error; this gets
expanded!!!
    // ...
};
```

By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

Function template definition

```
template <class T>
T max(T x, T y)
{
    return (x > y) ? x : y;
};
```

The data type is represented by the template argument: **<class T>**. When used in an application, the compiler generates the appropriate function according to the data type actually used in the call:

```
int i;
Myclass a, b;

int j = max(i,0); // arguments are integers
Myclass m = max(a,b); // arguments are type Myclass
```



Any data type (not just a class) can be used for **<class T>**. The compiler takes care of calling the appropriate **operator>()**, so you can use **max** with arguments of any type for which **operator>()** is defined.

Overriding a template function

The previous example is called a *function template* (or *generic function*, if you like). A specific instantiation of a function template is called a *template function*. You can override the generation of a template function for a specific type with a non-template function:

```
#include <string.h>

char *max(char *x, char *y)
{
    return(strcmp(x,y)>0) ?x:y;
}
```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't there is no way of deducing the actual values for the unused template arguments when the function is called.

Implicit and explicit template functions

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```
template<class T> T max(T a, T b)
{
    return (a > b) ? a : b;
}

void f(int i, char c)
{
    max(i, i);           // calls max(int ,int )
    max(c, c);          // calls max(char,char)
    max(i, c);          // no match for max(int,char)
    max(c, i);          // no match for max(char,int)
}
```

This code results in the following error messages.

```
Could not find a match for 'max(int,char)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)
```

If the user explicitly declares a template function, this function, on the other hand, will participate fully in overload resolution. For example:

```

template<class T> T max(T a, T b)
{
    return (a > b) ? a : b;
}

int    max(int,int);           // declare max(int,int) explicitly

void   f(int i, char c)
{
    max(i, i);                 // calls max(int ,int )
    max(c, c);                 // calls max(char,char)
    max(i, c);                 // calls max(int,int)
    max(c, i);                 // calls max(int,int)
}

```

Class templates

A class template (also called a *generic class* or *class generator*) allows you to define a pattern for class definitions. Generic container classes are good examples. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a *type* parameter to the class, the system will generate type-safe class definitions on the fly:

Class template definition

```

#include <iostream.h>

template <class T>
class Vector
{
    T *data;
    int size;

public:
    Vector(int);
    ~Vector() {delete[] data;}
    T& operator[](int i) {return data[i];}
};

// Note the syntax for out-of-line definitions:
template <class T>
Vector<T>::Vector(int n)
{
    data = new T[n];
    size = n;
};

main()
{
    Vector<int> x(5); // Generate a vector of ints
}

```



```

    for (int i = 0; i < 5; ++i)
        x[i] = i;
    for (i = 0; i < 5; ++i)
        cout << x[i] << ' ';
    cout << '\n';
    return 0;
}

// Output will be: 0 1 2 3 4

```

As with function templates, an explicit *template class* definition may be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol **Vector** must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

For a more complete implementation of a vector class, see the file `vectimp.h` in the container class library source code, found in the `\BORLANDC\CLASSLIB\INCLUDE` subdirectory. Also see Chapter 6, "The container class library," page 230.

Arguments Although these examples use only one template argument, multiple arguments are allowed. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Non-type template arguments such as *size* can have default arguments. The value supplied for a non-type template argument must be a constant expression:

```

const int N = 128;
int i = 256;

Buffer<int, 2*N> b1; // OK
Buffer<float, i> b2; // Error: i is not constant

```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

Angle brackets Take care when using the right angle bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the `>` between `x` and `100` would prematurely close the template argument list.

Type-safe generic lists

In general, when you need to write lots of nearly identical things, think templates. The problems with the following class definition, a generic list class,

```
class GList
{
public:
    void insert( void * );
    void *peek();
    // ...
};
```

are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what you'll get back out. You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public GList
{
public:
    void insert( Foo *f ) { GList::insert( f ); }
    Foo *peek() { return (Foo *)GList::peek(); }
    // ...
};
```

This is type-safe. **insert** will only take arguments of type pointer-to-**Foo** or object-derived-from-**Foo**, so the underlying container will only hold pointers that in fact point to something of type **Foo**. This means that the cast in **FooList::peek** is always safe, and you've created a true **FooList**. Now to do the same thing for a **BarList**, a **BazList**, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and be type-safe, once again, templates to the rescue:

Type-safe generic list class definition

```
template <class T> class List : public GList
{
public:
    void insert( T *t ) { GList::insert( t ); }
    T *peek() { return (T *)GList::peek(); }
    // ...
};

List<Foo> fList; // create a FooList class and an instance
                // named fList.
List<Bar> bList; // create a BarList class and an instance
```

```

        named bList.
List<Baz> zList; // create a BazList class and an instance
                named zList.

```

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. And there's no code generated by the type conversions from each wrapper class so there's no run-time overhead imposed by this type safety.

Eliminating pointers

Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of virtual function calls required, since the compiler knows the actual types of the objects. This is a big benefit if the virtual functions are small enough to be effectively inlined. It's difficult to inline virtual functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

Template definition that eliminates pointers

```

template <class T> aBase
{
    // ...
private:
    T buffer;
};

class anObject : public aSubject, public aBase<aFilebuf>
{
    // ...
};

```

All the functions in **aBase** can call functions defined in **aFilebuf** directly, without having to go through a pointer. And if any of the functions in **aFilebuf** can be inlined, you'll get a speed improvement, since templates allow them to be inlined.

Template compiler switches



The **-Jg** family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler. For template functions the switch applies to the function instances; for template classes, it will apply to all member functions and static data members of the template class. In all cases this switch applies only to compiler-generated template instances, and never to user-defined instances, although

it can be used to tell the compiler which instances will be user-defined so that they are not generated from the template.

-Jg Default value of the switch. All template instances first encountered when this switch value is in effect will be generated, such that if several compilation units generate the same template instance, the linker will merge them to produce a single copy of the instance. This is the most convenient approach to generating template instances, because it's almost entirely automatic. Note, though, that in order to be able to generate the template instances, the compiler must have the function body (in case of a template function) or bodies of member functions and definitions for static data members (in case of a template class).

-Jgd Instructs the compiler to generate public definitions for template instances. This is similar to **-Jg**, but if more than one compilation unit generates a definition for the same template instance, the linker will report public symbol re-definition errors.

-Jgx Instructs the compiler to generate external references to template instances. Some other compilation unit must generate a public definition for that template instance (using the **-Jgd** switch) so that the external references can be satisfied.

Using template switches

Using the **-Jg** family of switches, there are two basic approaches for generating template instances:

1. Include the function body (for a function template) or member function and static data member definitions (for a template class) in the header file that defines the particular template, and use the default setting of the template switch (**-Jg**). If some instances of the template are user-defined, the declarations (prototypes, for example) for them should be included in the same header, but preceded by **#pragma option -Jgx**, thus letting the compiler know that it should not generate those particular instances.

Here's an example of a template function header file:

```
// Declare a template function along with its body
template<class T> void sort(T* array, int size)
{
    ... body of template function goes here ...
}
```

```

// Sorting of 'int' elements done by user-defined instance
#pragma option -Jgx
extern void sort(int* array, int size);
// Restore the template switch to its original state
#pragma option -Jg.

```

If the preceding header file is included in a C++ source file, the 'sort' template can be used without worrying about how the various instances are generated (with the exception of 'sort' for int arrays, which is declared as a user-defined instance, and whose definition must be defined by the user).

2. Compile all of the source files comprising the program with the **-Jgx** switch (causing external references to templates to be generated); this way, template bodies don't need to appear in header files. In order to provide the definitions for all of the template instances, add a file (or files) to the program that includes the template bodies (including any user-defined instance definitions), and list all the template instances needed in the rest of the program, to provide the necessary public symbol definitions. Compile the file (or files) with the **-Jgd** switch.

Here's an example:

```

// vector.h
template <class elem, int size> class vector
{
    elem * value;
public:
    vector();
    elem & operator[](int index) { return value[index]; }
};

// MAIN.CPP
#include "vector.h"
// Tell the compiler that the template instances that follow
// will be defined elsewhere.
#pragma option -Jgx
// Use two instances of the 'vector' template class.
vector<int,100> int_100;
vector<char,10> char_10;
main()
{
    return int_100[0] + char_10[0];
}

```

```
}  
//  TEMPLATE.CPP  
#include <string.h>  
#include "vector.h"  
// Define any template bodies  
template <class elem, int size> vector<elem, size>::vector()  
{  
    value = new elem[size];  
    memset(value, 0, size * sizeof(elem));  
}  
// Generate the necessary instances  
#pragma option -Jgd  
typedef vector<int,100> fake_int_100;  
typedef vector<char,10> fake_char_10;
```


The preprocessor

Although Borland C++ uses an integrated single-pass compiler for its IDE and command-line versions, it is useful to retain the terminology associated with earlier multipass compilers.

With a multipass compiler, a first pass of the source text would pull in any include files, test for any conditional-compilation directives, expand any macros, and produce an intermediate file for further compiler passes. Since the IDE and command-line versions of the Borland C++ compiler perform this first pass with no intermediate output, Borland C++ provides an independent preprocessor, CPP.EXE, that does produce such an output file. The independent preprocessor is useful as a debugging aid, letting you see the net result of include directives, conditional compilation directives, and complex macro expansions.

The independent preprocessor is documented online.

The following discussion on preprocessor directives, their syntax and semantics, therefore, applies both to the CPP preprocessor and to the preprocessor functionality built into the Borland C++ compiler.

The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them.

The Borland C++ preprocessor includes a sophisticated macro processor that scans your source code before the compiler itself gets to work. The preprocessor gives you great power and flexibility in the following areas:

- Defining macros that reduce programming effort and improve your source code legibility. Some macros can also eliminate the overhead of function calls.

- Including text from other files, such as header files containing standard library and user-supplied function prototypes and manifest constants.
- Setting up conditional compilations for improved portability and for debugging sessions.

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program.

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

The full syntax for Borland C++'s preprocessor directives is given in the next table.

Table 4.1: Borland C++ preprocessing directives syntax

<i>preprocessing-file:</i> group	#pragma warn action abbreviation newline #pragma inline newline # newline
<i>group:</i> group-part group group-part	action: one of + - .
<i>group-part:</i> <pp-tokens> newline if-section control-line	abbreviation: nondigit nondigit nondigit
<i>if-section:</i> if-group <elif-groups> <else-group> endif-line	<i>lparen:</i> the left parenthesis character without preceding whitespace
<i>if-group:</i> #if constant-expression newline <group> #ifdef identifier newline <group> #ifndef identifier newline <group>	replacement-list: <pp-tokens>
<i>elif-groups:</i> elif -group elif -groups elif-group	<i>pp-tokens:</i> preprocessing-token pp-tokens preprocessing-token
<i>elif-group:</i> #elif constant-expression newline <group>	<i>preprocessing-token:</i> header-name (only within an #include directive) identifier (no keyword distinction) constant string-literal operator punctuator
<i>else-group:</i> #else newline <group>	each non-whitespace character that cannot be one of the preceding
<i>endif-line:</i> #endif newline	<i>header-name:</i> <h-char-sequence>
<i>control-line:</i> #include pp-tokens newline #define identifier replacement-list newline #define identifier lparen <identifier-list> replacement-list newline #undef identifier newline #line pp-tokens newline #error <pp-tokens> newline #pragma <pp-tokens> newline	<i>h-char-sequence:</i> h-char h-char-sequence h-char
	<i>h-char:</i> any character in the source character set except the newline (\n) or greater than (>) character
	<i>newline:</i> the newline character

Null directive

The null directive consists of a line containing the single character #. This directive is always ignored.

The #define and #undef directives

The **#define** directive defines a *macro*. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

Simple #define macros

In the simple case with no parameters, the syntax is as follows:

```
#define macro_identifier <token_sequence>
```

Each occurrence of *macro_identifier* in your source code following this control line will be replaced *in situ* with the possibly empty *token_sequence* (there are some exceptions, which are noted later). Such replacements are known as *macro expansions*. The token sequence is sometimes called the *body* of the macro.

Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.

An empty token sequence results in the effective removal of each affected macro identifier from the source code:

```
#define HI "Have a nice day!"
#define empty
#define NIL ""
...
puts(HI);          /* expands to puts("Have a nice day!"); */
puts(NIL);         /* expands to puts(""); */
puts("empty");    /* NO expansion of empty! */
/* NOR any expansion of the empty within comments! */
```

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of *nested macros*: The expanded text may contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, such a directive will not be recognized by the preprocessor:

```

#define GETSTD #include <stdio.h>
...
GETSTD /* compiler error */

```

GETSTD will expand to #include <stdio.h>. However, the preprocessor itself will not obey this apparently legal directive, but will pass it verbatim to the compiler. The compiler will reject #include <stdio.h> as illegal input. A macro won't be expanded during its own expansion. So #define A A won't expand indefinitely.

The #undef directive

You can undefine a macro using the **#undef** directive:

```
#undef macro_identifier
```

This line detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined.

No macro expansion occurs within **#undef** lines.

The state of being *defined* or *undefined* turns out to be an important property of an identifier, regardless of the actual definition. The **#ifdef** and **#ifndef** conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with **#define**, using the same or a different token sequence.

```

#define BLOCK_SIZE 512
...
buff = BLOCK_SIZE*blks; /* expands as 512*blks */
...
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be illegal "unknown" identifier */
...
#define BLOCK_SIZE 128 /* redefinition */
...
buf = BLOCK_SIZE*blks; /* expands as 128*blks */
...

```

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is *exactly* the same, token-by-token definition as the existing one. The preferred strategy where definitions may exist in other header files is as follows:

```

#ifdef BLOCK_SIZE
#define BLOCK_SIZE 512
#endif

```

The middle line is bypassed if `BLOCK_SIZE` is currently defined; if `BLOCK_SIZE` is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single space character.

Assembly language programmers must resist the temptation to write:

```

#define BLOCK_SIZE = 512 /* ?? token sequence includes the = */

```

The `-D` and `-U` options

Identifiers can be defined and undefined using the command-line compiler options `-D` and `-U` (see Chapter 5, “The command-line compiler,” in the *User’s Guide*). Identifiers can be defined, but not explicitly undefined, from the IDE Options | Compiler | Code Generation dialog box (see Chapter 2, “IDE basics,” also in the *User’s Guide*).

The command line

```

BCC -Ddebug=1; paradox=0; X -Umymym myprog.c

```

is equivalent to placing

```

#define debug 1
#define paradox 0
#define X
#undef mymym

```

in the program.

The Define option

Identifiers can be defined, but not explicitly undefined, from the Defines input box in the Code Generation | Options dialog box (under O | C | Code Generation) (see Chapter 2, “IDE basics,” in the *User’s Guide*).

Keywords and protected words

It is legal but ill-advised to use Borland C++ keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long   /* legal and possibly useful */
```

The following predefined global identifiers may *not* appear immediately following a **#define** or **#undef** directive:

Note the double underscores, leading and trailing.

```
__STDC__          __DATE__
__FILE__          __TIME__
__LINE__
```

Macros with parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note that there can be no whitespace between the macro identifier and the (. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a *formal argument* or *place holder*.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences and potential pitfalls (see page 164).

The optional *actual_arg_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the *formal_arg_list* of the **#define** line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual_arg_list*. For example,

```

#define CUBE(x) ((x)*(x)*(x))
...
int n,y;
n = CUBE(y);

```

results in the following replacement:

```
n = ((y) * (y) * (y));
```

Similarly, the last line of

```

#define SUM (a,b) ((a) + (b))
...
int i,j,sum;
sum = SUM(i,j);

```

expands to $sum = ((i) + (j))$. The reason for the apparent glut of parentheses will be clear if you consider the call

```
n = CUBE(y+1);
```

Without the inner parentheses in the definition, this would expand as $n = y+1*y+1 *y+1$, which is parsed as

```
n = y + (1*y) + (1*y) + 1; // != (y+1) cubed unless y=0 or y = -3!
```

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Note the following points when using macros with argument lists:

1. **Nested parentheses and commas:** The *actual_arg_list* may contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters:

```

#define ERRMSG(x, str) showerr("Error",x,str)
#define SUM(x,y) ((x) + (y))
...
ERRMSG(2, "Press Enter, then Esc");
/* expands to showerr("Error",2,"Press Enter, then Esc");
return SUM(f(i,j), g(k,l));
/* expands to return ((f(i,j)) + (g(k,l))); */

```

2. **Token pasting with ##:** You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers; for example, given the definition

```
#define VAR(i,j) (i##j)
```

then the call `VAR(x,6)` would expand to `(x6)`. This replaces the older (nonportable) method of using `(i/**/j)`.

3. **Converting to strings with #:** The `#` symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement. So, given the following macro definition:

```
#define TRACE(flag) printf(#flag "=%d\n",flag)
```

the code fragment

```
int highval = 1024;
TRACE(highval);
```

becomes

```
int highval = 1024;
printf("highval" "= %d\n", highval);
```

which, in turn, is treated as

```
int highval = 1024;
printf("highval=%d\n", highval);
```

4. **The backslash for line continuation:** A long token sequence can straddle a line by using a backslash (`\`). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions:

```
#define WARN "This is really a single-\
line warning"
...
puts(WARN);
/* screen will show: This is really a single-line warning */
```

5. **Side effects and other dangers:** The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once. Compare **CUBE** and **cube** in the following example:

```
int cube(int x) {
    return x*x*x;
}
#define CUBE(x) ((x)*(x)*(x))
...
int b = 0, a = 3;
b = cube(a++);
/* cube() is passed actual arg = 3; so b = 27; a now = 4.*/
```

Final value of *b* depends on what your compiler does to the expanded expression.

```
a = 3;
b = CUBE(a++);
/* expands as ((a++)*(a++)*(a++)); a now = 6 */
```

File inclusion with #include

The **#include** directive pulls in other named files, known as *include files*, *header files*, or *headers*, into the source code. The syntax has three forms:

The angle brackets are real tokens, not metasympols that imply that *header_name* is optional.

```
#include <header_name>
#include "header_name"
#include macro_identifier
```

The third variant assumes that neither `<` nor `"` appears as the first non-whitespace character following **#include**; further, it assumes that a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the `<header_name>` or `"header_name"` formats.

The first and second variant imply that no macro expansion will be attempted; in other words, *header_name* is never scanned for macro identifiers. *header_name* must be a valid DOS file name with an extension (traditionally .h for header) and optional path name and path delimiters.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the **#include** may therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *header_name*, only that directory will be searched.

The difference between the `<header_name>` and `"header_name"` formats lies in the searching algorithm employed in trying to locate the include file; these algorithms are described in the following two sections.

Header file
search with
<header_name>

The <header_name> variant specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.

Header file
search with
"header_name"

The "header_name" variant specifies a user-supplied include file; the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file is not found there, the search continues in the include directories as in the <header_name> situation.

The following example clarifies these differences:

```
#include <stdio.h>
/* header in standard include directory */

#define myinclud C:\BORLANDC\INCLUDE\MYSTUFF.H
/* Note: Single backslashes OK here; within a C statement you would
   need "C:\BORLANDC\INCLUDE\MYSTUFF.H" */

#include myinclud
/* macro expansion */

#include "myinclud.h"
/* no macro expansion */
```

After expansion, the second **#include** statement causes the preprocessor to look in C:\BORLANDC\INCLUDE\MYSTUFF.H and nowhere else. The third **#include** causes it to look for MYINCLUD.H in the current directory, then in the default directories.

Conditional compilation

Borland C++ supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with # (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The #if, #elif, #else, and #endif conditional directives

The conditional directives **#if**, **#elif**, **#else**, and **#endif** work like the normal C conditional operators. They are used as follows:

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>
...
<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif
...
```

If the *constant-expression-1* (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by *section-1*, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the Borland C++ compiler. Otherwise, if *constant-expression-1* evaluates to zero (false), *section-1* is ignored (no macro expansion and no compilation).

In the *true* case, after *section-1* has been preprocessed, control passes to the matching **#endif** (which ends this conditional interlude) and continues with *next-section*. In the *false* case, control passes to the next **#elif** line (if any) where *constant-expression-2* is evaluated. If true, *section-2* is processed, after which control moves on to the matching **#endif**. Otherwise, if *constant-expression-2* is false, control passes to the next **#elif**, and so on, until either **#else** or **#endif** is reached. The optional **#else** is used as an alternative condition for which all previous tests have proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#if** must be carefully balanced with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#if** can be matched with its correct **#endif**.

The constant expressions to be tested must evaluate to a constant integral value.

The operator `defined` The **defined** operator offers an alternative, more flexible way of testing whether combinations of identifiers are defined or not. It is valid only in **#if** and **#elif** expressions.

The expression **defined**(*identifier*) or **defined** *identifier* (parentheses are optional) evaluates to 1 (true) if the symbol has been previously defined (using **#define**) and has not been subsequently undefined (using **#undef**); otherwise, it evaluates to 0 (false). So the directive

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use **defined** repeatedly in a complex expression following the **#if** directive, such as

```
#if defined(mysym) && !defined(yoursym)
```

The **#ifdef** and **#ifndef** conditional directives

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not, that is, whether a previous **#define** command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if *identifier* is currently defined, and the same effect as

```
#if 0
```

if *identifier* is currently undefined.

#ifndef tests true for the “not-defined” condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if *identifier* is currently defined, and the same effect as

```
#if 1
```

if *identifier* is currently undefined.

The syntax thereafter follows that of the **#if**, **#elif**, **#else**, and **#endif** given in the previous section.

An identifier defined as NULL is considered to be defined.

The #line line control directive

You can use the **#line** command to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program. The syntax is

```
#line integer_constant <"filename">
```

indicating that the following source line originally came from line number *integer_constant* of *filename*. Once the *filename* has been registered, subsequent **#line** commands relating to that file can omit the explicit *filename* argument.

The inclusion of stdio.h means that the preprocessor output will be somewhat large.

```
/* TEMP.C: An example of the #line directive */
#include <stdio.h>
#line 4 "junk.c"
void main()
{
    printf(" in line %d of %s", __LINE__, __FILE__);
#line 12 "temp.c"
    printf("\n");
    printf(" in line %d of %s", __LINE__, __FILE__);
#line 8
    printf("\n");
    printf(" in line %d of %s", __LINE__, __FILE__);
}
```

If you run TEMP.C through CPP (cpp temp), you'll get an output file TEMP.I; it should look like this:

```
temp.c 1:
C:\BORLAND\BORLANDC\CPP\INCLUDE\STDIO.H 1:
C:\BORLAND\BORLANDC\CPP\INCLUDE\STDIO.H 2:
C:\BORLAND\BORLANDC\CPP\INCLUDE\STDIO.H 3:
...
C:\BORLAND\BORLANDC\CPP\INCLUDE\STDIO.H 212:
C:\BORLAND\BORLANDC\CPP\INCLUDE\STDIO.H 213:
```

We've eliminated most of the stdio.h portion.

```

temp.c 2:
temp.c 3:
junk.c 4: void main()
junk.c 5: {
junk.c 6: printf(" in line %d of %s",6,"junk.c");
junk.c 7:
temp.c 12: printf("\n");
temp.c 13: printf(" in line %d of %s",13,"temp.c");
temp.c 14:
temp.c 8: printf("\n");
temp.c 9: printf(" in line %d of %s",9,"temp.c");
temp.c 10: }
temp.c 11:

```

If you then compile and run TEMP.C, you'll get the output shown here:

```

    in line 6 of junk.c
    in line 13 of temp.c
    in line 9 of temp.c

```

Macros are expanded in **#line** arguments as they are in the **#include** directive.

The **#line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

The #error directive

The **#error** directive has the following syntax:

```
#error errmsg
```

This generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional that is true for the undesired case.

For example, suppose you **#define** MYVAL, which must be either 0 or 1. You could then include the following conditional in your source code to test for an incorrect value of MYVAL:

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

The #pragma directive

The #pragma directive permits implementation-specific directives of the form:

#pragma *directive-name*

With #pragma, Borland C++ can define whatever directives it desires without interfering with other compilers that support #pragma. If the compiler doesn't recognize *directive-name*, it ignores the #pragma directive without any error or warning message.

Borland C++ supports the following #**pragma** directives:

- #pragma argsused
- #pragma exit
- #pragma hdrfile
- #pragma hdrstop
- #pragma inline
- #pragma option
- #pragma saveregs
- #pragma startup
- #pragma warn
- #pragma intrinsic

Borland C++ only

#pragma argsused

The **argsused** pragma is only allowed between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

#pragma exit and #pragma startup

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the main

function is called), or program exit (just before the program terminates through `_exit`).

The syntax is as follows:

```
#pragma startup function-name <priority>
#pragma exit function-name <priority>
```

The specified *function-name* must be a previously declared function taking no arguments and returning **void**; in other words, it should be declared as

```
void func(void);
```

Priorities from 0 to 63 are used by the C libraries, and should not be used by the user.

The optional *priority* parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100. For example,

```
#include <stdio.h>

void startFunc(void)
{
    printf("Startup function.\n");
}

#pragma startup startFunc 64
/* priority 64 --> called first at startup */

void exitFunc(void)
{
    printf("Wrapping up execution.\n");
}

#pragma exit exitFunc
/* default priority is 100 */

void main(void)
{
    printf("This is main.\n");
}
```

*Note that the function name used in **pragma startup** or **exit** must be defined (or declared) before the pragma line is reached.*

#pragma hdrfile

This directive sets the name of the file in which to store precompiled headers. The default file name is `TCDEF.SYM`. The syntax is

```
#pragma hdrfile "filename.SYM"
```

See Appendix D, "Precompiled headers" in the *User's Guide* for more details.

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option **-H=filename** or the Precompiled Header (O|C|Code Generation) to change the name of the file used to store precompiled headers.

#pragma hdrstop

This directive terminates the list of header files that are eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers. (See Appendix D in the *User's Guide* for more on precompiled headers.)

#pragma inline

This directive is equivalent to the **-B** command-line compiler option or the IDE inline option. It tells the compiler that there is inline assembly language code in your program (see Chapter 12, "BASM and inline assembly"). The syntax is

#pragma inline

This is best placed at the top of the file, since the compiler restarts itself with the **-B** option when it encounters **#pragma inline**. Actually, you can leave off both the **-B** option and the **#pragma inline** directive, and the compiler will restart itself anyway as soon as it encounters **asm** statements. The purpose of the option and the directive is to save some compilation time.

#pragma intrinsic

#pragma intrinsic is documented in Appendix A, "The Optimizer" in the *User's Guide*.

#pragma option

Use **#pragma option** to include command-line options within your program code. The syntax is

#pragma option [options...]

The command-line compiler options are defined in Chapter 5 in the *User's Guide*.

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. Any of the toggle options (such as **-a** or **-K**) can be turned on and off as on the command line. For these toggle options, you can also put a period following the option to return

the option to its command-line, configuration file, or option-menu setting. This allows you to temporarily change an option, then return it to its default, without you having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a **pragma option** include

-B	-H	-Q
-c	-lfilename	-S
-dname	-Lfilename	-T
-Dname = string	-lxset	-Uname
-efilename	-M	-V
-E	-o	-X
-Fx	-P	-Y

You can use **#pragmas**, **#includes**, **#define**, and some **#ifs** before

1. The use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an **#if**, **#ifdef**, **#ifndef** or **#elif** directive.
2. The occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can *only* appear in a **#pragma option** command before these events. These options are

-Efilename	-m*	-u
-f*	-npath	-W
-i#	-ofilename	-z*

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

-1	-h	-r
-2	-k	-rd
-a	-N	-v
-ff	-O	-y
-G	-p	-Z

The following options can be changed at any time and take effect immediately:

See page 352 for more on using **#pragma option** with far objects.

-A	-gn	-zE
-b	-jn	-zF
-C	-K	-zH
-d	-wxxx	

They can additionally appear followed by a dot (.) to reset the option to its command-line state.

#pragma saveregs

The **saveregs** pragma guarantees that a **huge** function will not change the value of any of the registers when it is entered. This directive is sometimes needed for interfacing with assembly language code. The directive should be placed immediately before the function definition. It applies to that function alone.

#pragma warn

The **warn** directive lets you override specific **-wxxx** command-line options or check **Display Warnings** settings in the **Options | Compiler | Messages** dialog boxes.

For example, if your source code contains the directives

```
#pragma warn +xxx  
#pragma warn -yyy  
#pragma warn .zzz
```

the *xxx* warning will be turned on (even if on the **Options | Compiler | Messages** menu it was toggled to *Off*), the *yyy* warning will be turned off, and the *zzz* warning will be restored to the value it had when compilation of the file began.

A complete list of the three-letter abbreviations and the warnings to which they apply is given in Chapter 5, “The command-line compiler” in the *User’s Guide*.

Predefined macros

Borland C++ predefines certain global identifiers, each of which is discussed in this section. Except for `__cplusplus` and `_Windows`, each of these starts and ends with two underscore characters (`_ _`). These macros are also known as *manifest constants*.

__BCPLUSPLUS__

This macro is specific to Borland’s C and C++ family of compilers. It is only defined for C++ compilation. If you’ve selected C++

compilation, it is defined as 0x0300, a hexadecimal constant. This numeric value will increase in later releases.

`__BORLANDC__`

This macro is specific to Borland's C and C++ family of compilers. It is defined as 0x0400, a hexadecimal constant. This numeric value will increase in later releases.

`__CDECL__`

This macro is specific to Borland's C and C++ family of compilers. It signals that the `-p` flag was not used (the C radio button in the Entry/Exit Code Generation dialog box). Set to the integer constant 1 if calling was not used; otherwise, undefined.

The following six symbols are defined based on the memory model chosen at compile time.

<code>__COMPACT__</code>	<code>__MEDIUM__</code>
<code>__HUGE__</code>	<code>__SMALL__</code>
<code>__LARGE__</code>	<code>__TINY__</code>

Only one is defined for any given compilation; the others, by definition, are undefined. For example, if you compile with the small model, the `__SMALL__` macro is defined and the rest are not, so that the directive

```
#if defined(_SMALL_)
```

will be true, while

```
#if defined(_LARGE_)
```

(or any of the others) will be false. The actual value for any of these defined macros is 1.

`__cplusplus`

This macro is defined as 1 if in C++ mode; it's undefined otherwise. This allows you to write a module that will be compiled sometimes as C and sometimes as C++. Using conditional compilation, you can control which C and C++ parts are included.

__DATE__

This macro provides the date the preprocessor began processing the current source file (as a string literal).

Each inclusion of `__DATE__` in a given file contains the same value, regardless of how long the processing takes. The date appears in the format *mmm dd yyyy*, where *mmm* equals the month (Jan, Feb, and so forth), *dd* equals the day (1 to 31, with the first character of *dd* a blank if the value is less than 10), and *yyyy* equals the year (1990, 1991, and so forth).

__DLL__



This macro is specific to Borland's C and C++ family of compilers. It is defined to be 1 if you compile a module with the `-WD` command-line compiler option or are using the Windows DLL All Functions Exportable radio button (O|C|C|Entry/Exit Code) to generate code for Windows DLLs; otherwise it remains undefined.

__FILE__

This macro provides the name of the current source file being processed (as a string literal). This macro changes whenever the compiler processes an `#include` directive or a `#line` directive, or when the include file is complete.

__LINE__

This macro provides the number of the current source-file line being processed (as a decimal constant). Normally, the first line of a source file is defined to be 1, through the `#line` directive can affect this. See page 169 for information on the `#line` directive.

__MSDOS__

This macro is specific to Borland's C/C++ family of compilers. It provides the integer constant 1 for all compilations.

__OVERLAY__

This macro is specific to Borland's C and C++ family of compilers. It is predefined to be 1 if you compile a module with the **-Y** option (enable overlay support). If you don't enable overlay support, this macro is undefined.

__PASCAL__

This macro is specific to Borland's C and C++ family of compilers. It signals that the **-p** flag or the Pascal calling convention (**O|C|C|Exit/Entry**) has been used. The macro is set to the integer constant 1 if used; otherwise, it remains undefined.

__STDC__

This macro is defined as the constant 1 if you compile with the ANSI compatibility flag (**-A**) or ANSI radio button (Source Options); otherwise, the macro is undefined.

__TCPLUSPLUS__

This macro is specific to Borland's C and C++ family of compilers. It is only defined for C++ compilation. If you've selected C++ compilation, it is defined as 0x0300, a hexadecimal constant. This numeric value will increase in later releases.

__TEMPLATES__

This macro is specific to Borland's C and C++ family of compilers. It is defined as 1 for C++ files (meaning that Borland C++ supports templates); it's undefined otherwise.

__TIME__

This macro keeps track of the time the preprocessor began processing the current source file (as a string literal).

As with `__DATE__`, each inclusion of `__TIME__` contains the same value, regardless of how long the processing takes. It takes the format `hh:mm:ss`, where `hh` equals the hour (00 to 23), `mm` equals minutes (00 to 59), and `ss` equals seconds (00 to 59).

`__TURBOC__`

This macro is specific to Borland's C and C++ family of compilers. It is defined as `0x0400`, a hexadecimal constant. This numeric value will increase in later releases.

`_Windows`

Indicates that Windows-specific code is being generated. This macro is defined if you compile a module with any of the `-W` command-line compiler options enabled (generate Windows applications). If you don't enable any of these options, this macro is undefined.

Using C++ streams

This chapter is divided into two sections: a brief, practical overview of using C++ stream I/O, and a reference section to the C++ stream class library.

Stream input/output in C++ (commonly referred to as *iostreams*, or merely *streams*) provide all the functionality to the **stdio** library in C. *iostreams* are used to convert typed objects into readable text, and vice versa. Streams may also read and write binary data. The C++ language allows you to define or overload I/O functions and operators that are then called automatically for corresponding user-defined types.

What is a stream?

A stream is an abstraction referring to any flow of data from a source (or *producer*) to a *sink* (or *consumer*). We also use the synonyms *extracting*, *getting*, and *fetching* when speaking of inputting characters from a source; and *inserting*, *putting*, or *storing* when speaking of outputting characters to a sink. Classes are provided that support console output (`constrea.h`), memory buffers (`iostream.h`), files (`fstream.h`), and strings (`strstrea.h`) as sources or sinks (or both).



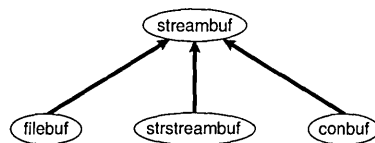
The iostream library

The **iostream** library has two parallel families of classes: those derived from **streambuf**, and those derived from **ios**. Both are low-level classes, each doing a different set of jobs. All stream classes have at least one of these two classes as a base class. Access from **ios**-based classes to **streambuf**-based classes is through a pointer.

The streambuf class

The **streambuf** class provides an interface to physical devices. **streambuf** provides general methods for buffering and handling streams when little or no formatting is required. **streambuf** is a useful base class employed by other parts of the **iostream** library, though you can also derive classes from it for your own functions and libraries. The classes **conbuf**, **filebuf** and **strstreambuf** are derived from **streambuf**.

Figure 5.1
Class **streambuf** and its
derived classes



The ios class

The class **ios** (and hence any of its derived classes) contains a pointer to a **streambuf**. It performs formatted I/O with error-checking using a **streambuf**.

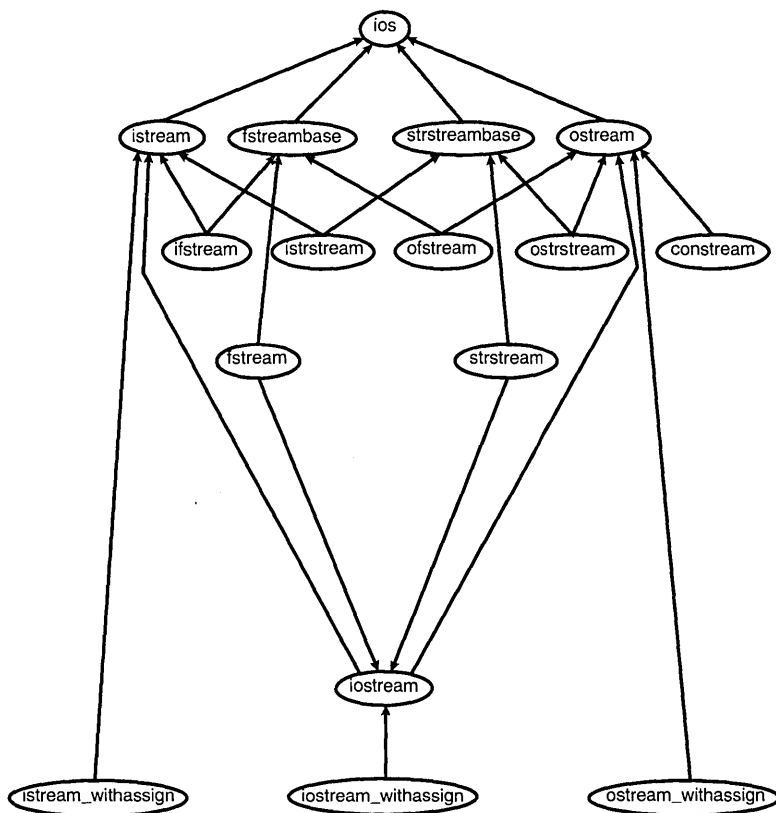
An inheritance diagram for all the **ios** family of classes is found in Figure 5.2. For example, the **ifstream** class is derived from the **istream** and **fstreambase** classes, and **istrstream** is derived from **istream** and **strstreambase**. This diagram is not a simple hierarchy because of the generous use of *multiple inheritance*. With multiple inheritance, a single class can inherit from more than one base class. (The C++ language provides for *virtual inheritance* to avoid multiple declarations.) This means, for example, that all the members (data and functions) of **iostream**, **istream**, **ostream**, **fstreambase**, and **ios** are part of objects of the **fstream** class. All classes in the **ios**-based tree use a **streambuf** (or a **filebuf** or **strstreambuf**, which are special cases of a **streambuf**) as its source and/or sink.

C++ programs start with four predefined open streams, declared as objects of **withassign** classes as follows:

```
extern istream_withassign cin; // Corresponds to stdin
extern ostream_withassign cout; // Corresponds to stdout
extern ostream_withassign cerr; // Corresponds to stderr
extern ostream_withassign clog; // A buffered cerr
```

Figure 5.2
Class **ios** and its derived classes

By accepted practice, the arrows point **from** the derived class **to** the base class.



Output

Stream output is accomplished with the *insertion* (or *put to*) operator, `<<`. The standard left shift operator, `<<`, is overloaded for output operations. Its left operand is an object of type **ostream**. Its right operand is any type for which stream output has been defined (that is, fundamental types or any types you have overloaded it for). For example,

```
cout << "Hello!\n";
```

writes the string "Hello!" to **cout** (the standard output stream, normally your screen) followed by a new line.

The **<<** operator associates from left to right and returns a reference to the **ostream** object for which it is invoked. This allows several insertions to be cascaded as follows:

```
int i = 8;
double d = 2.34;
cout << "i = " << i << ", d = " << d << "\n";
```

This will write the following to standard output:

```
i = 8, d = 2.34
```

Fundamental types

The fundamental data types directly supported are **char**, **short**, **int**, **long**, **char*** (treated as a string), **float**, **double**, **long double**, and **void***. Integral types are formatted according to the default rules for **printf** (unless you've changed these rules by setting various **ios** flags). For example, the following two output statements give the same result:

```
int i;
long l;
cout << i << " " << l;
printf("%d %ld", i, l);
```

The pointer (**void ***) inserter is used to display pointer addresses:

```
int i;
cout << &i;           // display pointer address in hex
```

Read the description of the **ostream** class (page 205) for other output functions.

Output formatting

Formatting for both input and output is determined by various *format state* flags contained in the class **ios**. The format flags are as follows:

```
public:
enum {
    skipws,      // skip whitespace on input
    left,        // left-adjust output
    right,       // right-adjust output
```

```

internal,    // pad after sign or base indicator
dec,        // decimal conversion
oct,        // octal conversion
hex,        // hexadecimal conversion
showbase,   // show base indicator on output
showpoint,  // show decimal point (floating-point output)
uppercase,  // uppercase hex output
showpos,    // show '+' with positive integers
scientific, // suffix floating-point numbers with exponential (E)
            // notation on output
fixed,      // use fixed decimal point for floating-point numbers
unitbuf,    // flush all streams after insertion
stdio,      // flush stdout, stderr after insertion
};

```

These flags are read and set with the **flags**, **setf**, and **unsetf** member functions (see class **ios** starting on page 199).

Manipulators

A simple way to change some of the format variables is to use a special function-like operator called a *manipulator*. Manipulators take a stream reference as an argument and return a reference to the same stream. You can embed manipulators in a chain of insertions (or extractions) to alter stream states as a side effect without actually performing any insertions (or extractions). For example,

Parameterized manipulators must be called for each stream operation.

```

#include <iostream.h>
#include <iomanip.h> // Required for parameterized manipulators.

int main(void) {
    int i = 6789, j = 1234, k = 10;

    cout << setw(6) << i << j << i << k << j;
    cout << "\n";
    cout << setw(6) << i << setw(6) << j << setw(6) << k;
    return(0);
}

```

Produces this output:

```

678912346789101234
6789 1234 10

```

setw is a *parameterized manipulator* declared in *iomanip.h*. Other parameterized manipulators, **setbase**, **setfill**, **setprecision**, **setiosflags** and **resetiosflags**, work in the same way. To make use

of these, your program must include `iomanip.h`. You can write your own manipulators without parameters:

```
#include <iostream.h>

// Tab and prefix the output with a dollar sign.
ostream& money( ostream& output) {
    return output << "\t$";
}

int main(void) {
    float owed = 1.35, earned = 23.1;
    cout << money << owed << money << earned;
    return(0);
}
```

produces the following output:

```
$1.35    $23.1
```

The non-parameterized manipulators **dec**, **hex**, and **oct** (declared in `iomanip.h`) take no arguments and simply change the conversion base (and leave it changed):

```
int i = 36;
cout << dec << i << " " << hex << i << " " << oct << i << endl;
cout << dec; // Must reset to use decimal base.
// displays 36 24 44
```

Table 5.1
Stream manipulators

Manipulator	Action
dec	Set decimal conversion base format flag.
hex	Set hexadecimal conversion base format flag.
oct	Set octal conversion base format flag.
ws	Extract whitespace characters.
endl	Insert newline and flush stream.
ends	Insert terminal null in string.
flush	Flush an ostream.
setbase(int n)	Set conversion base format to base <i>n</i> (0, 8, 10, or 16). 0 means the default: decimal on output, ANSI C rules for literal integers on input.
resetiosflags(long f)	Clear the format bits specified by <i>f</i> .
setiosflags(long f)	Set the format bits specified by <i>f</i> .
setfill(int c)	Set the fill character to <i>c</i> .
setprecision(int n)	Set the floating-point precision to <i>n</i> .
setw(int n)	Set field width to <i>n</i> .

The manipulator **endl** inserts a newline character and flushes the stream. You can also flush an **ostream** at any time with

```
ostream << flush;
```

Filling and padding

The fill character and the direction of the padding depend on the setting of the fill character and the left, right, and internal flags.

The default fill character is a space. You can vary this by using the function `fill`:

```
int i = 123;
cout.fill('*');
cout.width(6);
cout << i;           // display ***123
```

The default direction of padding gives right-justification (pad on the left). You can vary these defaults (and other format flags) with the functions `setf` and `unsetf`:

```
int i = 56;
...
cout.width(6);
cout.fill('#');
cout.setf(ios::left, ios::adjustfield);
cout << i;           // display 56####
```

The second argument, `ios::adjustfield`, tells `setf` which bits to set. The first argument, `ios::left`, tells `setf` what to set those bits to. Alternatively, you can use the manipulators `setfill`, `setiosflags`, and `resetiosflags` to modify the fill character and padding mode. See `ios` data members on page 199 for a list of masks used by `setf`.

Input

Stream input is similar to output but uses the overloaded right shift operator, `>>`, known as the *extraction* (*get from*) operator, or *extractor*. The left operand of `>>` is an object of type class `istream`. As with output, the right operand can be of any type for which stream input has been defined.

By default, `>>` skips whitespace (as defined by the `isspace` function in `ctype.h`), then reads in characters appropriate to the type of the input object. Whitespace skipping is controlled by the `ios::skipws` flag in the format state's enumeration. The `skipws` flag is normally set to give whitespace skipping. Clearing this flag (with `setf`, for example) turns off whitespace skipping. There is also a special "sink" manipulator, `ws`, that lets you discard whitespace.

Consider the following example:

```
int i;
double d;
cin >> i >> d;
```

When the last line is executed, the program skips any leading whitespace. The integer value (*i*) is then read. Any whitespace following the integer is ignored. Finally, the floating-point value (*d*) is read.

For type **char** (**signed** or **unsigned**), the effect of the **>>** operator is to skip whitespace and store the next (non-whitespace) character. If you need to read the next character, whether it is whitespace or not, you can use one of the **get** member functions (see the discussion of **istream**, beginning on page 202).

For type **char*** (treated as a string), the effect of the **>>** operator is to skip whitespace and store the next (non-whitespace) characters until another whitespace character is found. A final null character is then appended. Care is needed to avoid “overflowing” a string. You can alter the default width of zero (meaning no limit) using **width** as follows:

```
char array[SIZE];
cin.width(sizeof(array));
cin >> array;           // Avoids overflow.
```

For all input of fundamental types, if only whitespace is encountered nothing is stored in the target, and the *istream* state is set to *fail*. The target will retain its previous value; if it was uninitialized, it remains uninitialized.

I/O of user-defined types

To input or output your own defined types, you must overload the extraction and insertion operators. Here is an example:

```
#include <iostream.h>

struct info {
    char *name;
    double val;
    char *units;
};

// You can overload << for output as follows:
ostream& operator << (ostream& s, info& m) {
```

```

        s << m.name << " " << m.val << " " << m.units;
        return s;
    };

    // You can overload >> for input as follows:
    istream& operator >> (istream& s, info& m) {
        s >> m.name >> m.val >> m.units;
        return s;
    };

int main(void) {
    info x;
    x.name = new char[15];
    x.units = new char[10];

    cout << "\nInput name, value and units:";
    cin >> x;
    cout << "\nMy input:" << x;
    return(0);
}

```

Simple file I/O

The class **ofstream** inherits the insertion operations from **ostream**, while **ifstream** inherits the extraction operations from **istream**. The file-stream classes also provide constructors and member functions for creating files and handling file I/O. You must include `fstream.h` in all programs using these classes.

Consider the following example that copies the file `FILE.IN` to the file `FILE.OUT`:

```

#include <fstream.h>

int main(void) {
    char ch;
    ifstream f1("FILE.IN");
    ofstream f2("FILE.OUT");

    if (!f1) cerr << "Cannot open FILE.IN for input";
    if (!f2) cerr << "Cannot open FILE.OUT for output";
    while (f2 && f1.get(ch))
        f2.put(ch);
    return(0);
}

```

Note that if the **ifstream** or **ofstream** constructors are unable to open the specified files, the appropriate stream error state is set.

The constructors allow you to declare a file stream without specifying a named file. Later, you can associate the file stream with a particular file:

```
ofstream ofile;           // creates output file stream
...
ofile.open("payroll"); // ofile connects to file "payroll"
// do some payrollling...

ofile.close();           // close the ofile stream
ofile.open("employee"); // ofile can be reused...
```

By default, files are opened in text mode. This means that on input, carriage-return/linefeed sequences are converted to the '\n' character. On output, the '\n' character is converted to a carriage-return/linefeed sequence. These translations are not done in binary mode. The file opening mode is set with an optional second parameter to the **open** function, chosen from the following table:

Table 5.2
File modes

Mode bit	Action
ios::app	Append data—always write at end of file.
ios::ate	Seek to end of file upon original open.
ios::in	Open for input (default for ifstream s).
ios::out	Open for output (default for ofstream s).
ios::binary	Open file in binary mode.
ios::trunc	Discard contents if file exists (default if ios::out is specified and neither ios::ate nor ios::app is specified).
ios::nocreate	If file does not exist, open fails.
ios::noreplace	If file exists, open for output fails unless ate or app is set.

String stream processing

The functions defined in `strstream.h` support in-memory formatting, similar to **scanf** and **sprintf**, but much more flexible. All of the **istream** functions are available for the class **istream** (input string stream); likewise for output: **ostream** inherits from **ostream**.

Given a text file with the following format:

```
101 191 Cedar Chest
102 1999.99 Livingroom Set
```

Each line can be parsed into three components: an integer ID, a floating-point price, and a description. The output produced is:

```
1: 101 191.00 Cedar Chest
2: 102 1999.99 Livingroom Set
```

Here is the program:

```
#include <fstream.h>
#include <strstream.h>
#include <iomanip.h>
#include <string.h>

int main(int argc, char **argv) {
    int id;
    float amount;
    char description[41];
    ifstream inf(argv[1]);

    if (inf) {
        char inbuf[81];
        int lineno = 0;

        // Want floats to print as fixed point
        cout.setf(ios::fixed, ios::floatfield);

        // Want floats to always have decimal point
        cout.setf(ios::showpoint);

        while (inf.getline(inbuf,81)) {
            // 'ins' is the string stream:
            istrstream ins(inbuf,strlen(inbuf));
            ins >> id >> amount >> ws;
            ins.getline(description,41); // Linefeed not copied.
            cout << ++lineno << ": "
                 << id << '\t'
                 << setprecision(2) << amount << '\t'
                 << description << "\n";
        }
    }
    return(0);
}
```

Note the use of format flags and manipulators in this example. The calls to **setf** coupled with **setprecision** allow floating-point numbers to be printed in a money format. The manipulator **ws** skips whitespace before the description string is read.

Screen output streams

The class **constream**, derived from **ostream** and defined in `constrea.h`, provides the functionality of `conio.h` for use with C++ streams. This allows you to create output streams that write to specified areas of the screen, in specified colors, and at specific locations.



As with `conio.h` functions, `constreams` are not compatible with Windows. The screen area created by **constream** is not bordered or otherwise distinguished from the surrounding screen.

Console stream manipulators are provided to facilitate formatting of console streams. These manipulators work in the same way as the corresponding function provided by `conio.h`. For a detailed description of the manipulators' behavior and valid arguments, see the *Library Reference*.

Table 5.3
Console stream manipulators

Manipulator	conio function	Action
creol	creol	Clears to end of line in text window.
delline	delline	Deletes line in the text window.
highvideo	highvideo	Selects high-intensity characters.
inline	inline	Inserts a blank line in the text window.
lowvideo	lowvideo	Selects low-intensity characters.
normvideo	normvideo	Selects normal-intensity characters.
setattr(int)	textattr	Sets screen attributes.
setbk(int)	textcolor	Sets new character color.
setclr(int)	textcolor	Set the color.
setcrstyp(int)	_setcursortype	Selects cursor appearance.
setxy(int, int)	gotoxy	Positions the cursor at the specified position.

Typical use of parameterized manipulators.

```
#include <constrea.h>

int main(void) {
    constream win1;

    win1.window(1, 1, 40, 20); // Initialize the desired space.
    win1.clrscr();           // Clear this rectangle.

    // Use the parameterized manipulator to set screen attributes.
    win1 << setattr((BLUE<<4) | WHITE)
```

```

    << "This text is white on blue.";

    // Use this parameterized manipulator to specify output area.
    win1 << setxy(10, 10)
        << "This text is in the middle of the window.";
    return(0);
}

```

You can create multiple constreams, each writing to its own portion of the screen. Then, you can output to any them without having to reset the window each time.

```

#include <constrea.h>

int main(void) {
    constream demo1, demo2;

    demo1.window( 1, 2, 40, 10 );
    demo2.window( 1, 12, 40, 20 );

    demo1.clrscr();
    demo2.clrscr();

    demo1 << "Text in first window" << endl;
    demo2 << "Text in second window" << endl;
    demo1 << "Back to the first window" << endl;
    demo2 << "And back to the second window" << endl;
    return(0);
}

```

Stream class reference

The stream class library in C++ consists of several classes. This reference presents some of the most useful details of these classes, in alphabetical organization. The following cross-reference lists tell which classes belong to which header files.

constrea.h:	conbuf, constream
iostream.h:	ios, iostream, iostream_withassign, istream, istream_withassign, ostream, ostream_withassign, streambuf.
fstream.h:	filebuf, fstream, fstreambase, ifstream, ofstream.
strstrea.h:	istrstream, ostrstream, strstream, strstreambase, strstreambuf.

Specializes **streambuf** to handle console output.

constructor conbuf()

Makes an unattached **conbuf**.

Member functions

clreol void clreol()

Clears to end of line in text window.

clrscr void clrscr()

Clears the defined screen.

delline void delline()

Deletes a line in the window.

gotoxy void gotoxy(int x, int y)

Positions the cursor in the window at the specified location.

highvideo void highvideo()

Selects high-intensity characters.

incline void incline()

Inserts a blank line.

lowvideo void lowvideo()

Selects low-intensity characters.

normvideo void normvideo()

Selects normal-intensity characters.

overflow virtual int overflow(int = EOF)

Flushes the conbuf to its destination.

setcursortype void setcursortype(int cur_type)

Selects the cursor appearance.

textattr void textattr(int newattribute)

Selects cursor appearance.

textbackground	void textbackground(int <i>newcolor</i>) Selects the text background color.
textcolor	void textcolor(int <i>newcolor</i>) Selects character color in text mode.
textmode	static void textmode(int <i>newmode</i>) Puts the screen in text mode.
wherex	int wherex() Gets the horizontal cursor position.
wherey	int wherey() Gets the vertical cursor position.
window	void window(int <i>left</i> , int <i>top</i> , int <i>right</i> , int <i>bottom</i>) Defines the active window.

constream

<constrea.h>

Provides console output streams. This class is derived from **ostream**.

constructor	constream() Provides an unattached output stream to the console.
--------------------	---

Member functions

clrscr	void clrscr() Clears the screen.
rdbuf	conbuf *rdbuf() Returns a pointer to this constream's assigned conbuf.
textmode	void textmode(int <i>newmode</i>) Puts the screen in text mode.
window	void window(int <i>left</i> , int <i>top</i> , int <i>right</i> , int <i>bottom</i>) Defines the active window.

Specializes **streambuf** to handle files.

constructor filebuf();

Makes a **filebuf** that isn't attached to a file.

constructor filebuf(int *fd*);

Makes a **filebuf** attached to a file as specified by file descriptor *fd*.

constructor filebuf(int *fd*, char *, int *n*);

Makes a **filebuf** attached to a file and uses a specified *n*-character buffer.

Member functions

attach filebuf* attach(int)

Attaches this closed **filebuf** to opened file descriptor.

close filebuf* close()

Flushes and closes the file. Returns 0 on error.

fd Returns the file descriptor or EOF.

is_open int is_open();

Returns nonzero if the file is open.

open filebuf* open(const char*, int *mode*, int *prot* = filebuf::openprot);

Opens the given file and connects to it.

overflow virtual int overflow(int = EOF);

Flushes a buffer to its destination. Every derived class should define the actions to be taken.

seekoff virtual streampos seekoff(streamoff, ios::seek_dir, int);

Moves the file pointer relative to the current position.

setbuf virtual streambuf* setbuf(char*, int);

Specifies a buffer for this **filebuf**.

sync virtual int sync();

Establishes consistency between internal data structures and the external stream representation.

underflow virtual int underflow();

Makes input available. This is called when no more data exists in the input buffer. Every derived class should define the actions to be taken.

fstream

<fstream.h>

This stream class, derived from **fstreambase** and **iostream**, provides for simultaneous input and output on a **filebuf**.

constructor fstream();

Makes an **fstream** that isn't attached to a file.

constructor fstream(const char*, int, int = filebuf::openprot);

Makes an **fstream**, opens a file, and connects to it.

constructor fstream(int);

Makes an **fstream**, connects to an open file descriptor.

constructor fstream(int, char*, int);

Makes an **fstream** connected to an open file and uses a specified buffer.

Member functions

open void open(const char*, int, int = filebuf::openprot);

Opens a file for an **fstream**.

rdbuf filebuf* rdbuf();

Returns the **filebuf** used.

fstreambase

<fstream.h>

This stream class, derived from **ios**, provides operations common to file streams. It serves as a base for **fstream**, **ifstream**, and **ofstream**.

constructor fstreambase();

Makes an **fstreambase** that isn't attached to a file.

fstreambase

- constructor** `fstreambase(const char*, int, int = filebuf::openprot);`
Makes an **fstreambase**, opens a file, and connects to it.
- constructor** `fstreambase(int);`
Makes an **fstreambase**, connects to an open file descriptor.
- constructor** `fstreambase(int, char*, int);`
Makes an **fstreambase** connected to an open file and uses a specified buffer.

Member functions

- attach** `void attach(int);`
Connects to an open file descriptor.
- close** `void close();`
Closes the associated **filebuf** and file.
- open** `void open(const char*, int, int = filebuf::openprot);`
Opens a file for an **fstreambase**.
- rdbuf** `filebuf* rdbuf();`
Returns the **filebuf** used.
- setbuf** `void setbuf(char*, int);`
Uses a specified buffer.

ifstream

`<fstream.h>`

This stream class, derived from **fstreambase** and **istream**, provides input operations on a **filebuf**.

- constructor** `ifstream();`
Makes an **ifstream** that isn't attached to a file.
- constructor** `ifstream(const char*, int = ios::in, int = filebuf::openprot);`
Makes an **ifstream**, opens an input file in protected mode, and connects to it. The existing file contents are preserved; new writes are appended.

constructor ifstream(int);

Makes an **ifstream**, connects to an open file descriptor.

constructor ifstream(int fd, char *, int);

Makes an **ifstream** connected to an open file and uses a specified buffer.

Member functions

open void open(const char*, int, int = filebuf::openprot);

Opens a file for an **ifstream**.

rdbuf filebuf* rdbuf();

Returns the filebuf used.

ios

<iostream.h>

Provides operations common to both, input and output. Its derived classes (**istream**, **ostream**, **iostream**) specialize I/O with high-level formatting operations. The **ios** class is a base for **istream**, **ostream**, **fstreambase**, and **strstreambase**.

constructor ios(); **protected**

Constructs an **ios** object that has no corresponding **streambuf**.

constructor ios(streambuf *);

Associates a given **streambuf** with the stream.

Data members

```
static const long  adjustfield; // left | right | internal
static const long  basefield;  // dec | oct | hex
static const long  floatfield; // scientific | fixed
streambuf         *bp;         // the associated streambuf   protected
int               x_fill;      // padding character on // output protected
long              x_flags;     // formatting flag bits protected
int               x_precision; // floating-point precision on // output protected
```

int	state;	// current state of the	
		// streambuf	protected
ostream	*x_tie;	// the tied ostream, if any	protected
int	x_width;	// field width on output	protected

Member functions

bad int bad();
Nonzero if error occurred.

bitalloc static long bitalloc();
Acquires a new flag bit set. The return value may be used to set, clear, and test the flag. This is for user-defined formatting flags.

clear void clear(int = 0);
Sets the stream state to the given value.

eof int eof();
Nonzero on end of file.

fail int fail();
Nonzero if an operation failed.

fill char fill()
Returns the current fill character.

fill char fill(char);
Resets the fill character; returns the previous one.

flags long flags();
Returns the current format flags.

flags long flags(long);
Sets the format flags to be identical to the given **long**; returns previous flags. Use **flags(0)** to set the default format.

good int good();
Nonzero if no state bits set (that is, no errors appeared).

init void init(streambuf *); **protected**
Provides the actual initialization.

- precision** `int precision();`
Returns the current floating-point precision.
- precision** `int precision(int);`
Sets the floating-point precision; returns previous setting.
- rdbuf** `streambuf* rdbuf();`
Returns a pointer to this stream's assigned streambuf.
- rdstate** `int rdstate();`
Returns the stream state.
- self** `long setf(long);`
Sets the flags corresponding to those marked in the given **long**; returns previous settings.
- self** `long setf(long _setbits, long _field);`
The bits corresponding to those marked in *_field* are cleared, and then reset to be those marked in *_setbits*.
- setstate** `protected: void setstate(int);`
Sets all status bits.
- sync_with_stdio** `static void sync_with_stdio();`
Mixes `stdio` files and `iostreams`. This should not be used for new code.
- tie** `ostream* tie();`
Returns the *tied stream*, or zero if none. Tied streams are those that are connected such that when one is used, the other is affected. For example, **cin** and **cout** are tied; when **cin** is used, it flushes **cout** first.
- tie** `ostream* tie(ostream*);`
Ties another stream to this one and returns the previously tied stream, if any. When an input stream has characters to be consumed, or if an output stream needs more characters, the tied stream is first flushed automatically. By default, **cin**, **cerr** and **clog** are tied to **cout**.
- unsetf** `long unsetf(long);`
Clears the bits corresponding to those marked in the given **long**; returns previous settings.

- width** int width();
Returns the current width setting.
- width** int width(int);
Sets the width as given; returns the previous width.
- xalloc** static int xalloc();
Returns an array index of previously unused words that can be used as user-defined formatting flags.

iostream

<iostream.h>

This class, derived from **istream** and **ostream**, is simply a mixture of its base classes, allowing both input and output on a stream. It is a base for **fstream** and **strstream**.

- constructor** iostream(streambuf *);
Associates a given **streambuf** with the stream.

iostream_withassign

<iostream.h>

This class is an **iostream** with an added assignment operator.

- constructor** iostream_withassign();
Default constructor (calls **iostream**'s constructor).

Member functions

None (although the = operator is overloaded).

istream

<iostream.h>

Provides formatted and unformatted input from a **streambuf**. The >> operator is overloaded for all fundamental types, as explained in the narrative at the beginning of the chapter. This **ios** class is a base for **fstream**, **iostream**, **istrstream**, and **istream_withassign**.

constructor `istream(istreambuf *)`;

Associates a given **istreambuf** with the stream.

Member functions

gcount `int gcount()`;

Returns the number of characters last extracted.

get `int get()`;

Extracts the next character or EOF.

get `istream& get(signed char*, int len, char = '\n');`
`istream& get(unsigned char*, int len, char = '\n');`

Extracts characters into the given **char *** until the delimiter (third parameter) or end-of-file is encountered, or until (*len* - 1) bytes have been read. A terminating null is always placed in the output string; the delimiter never is. Fails only if no characters were extracted.

get `istream& get(signed char&);`
`istream& get(unsigned char&);`

Extracts a single character into the given character reference.

get `istream& get(istreambuf&, char = '\n');`

Extracts characters into the given **istreambuf** until the delimiter is encountered.

getline `istream& getline(signed char *buffer, int, char = '\n');`
`istream& getline(unsigned char *buffer, int, char = '\n');`

Same as **get**, except the delimiter is also extracted. The delimiter is not copied to *buffer*.

ignore `istream& ignore(int n = 1, int delim = EOF);`

Causes up to *n* characters in the input stream to be skipped; stops if **delim** is encountered.

peek `int peek()`;

Returns next char without extraction.

putback `istream& putback(char);`

Pushes back a character into the stream.

istream

read `istream& read(signed char*, int);`
`istream& read(unsigned char*, int);`

Extracts a given number of characters into an array. Use **gcount()** for the number of characters actually extracted if an error occurred.

seekg `istream& seekg(long);`

Moves to an absolute position (as returned from **tellg**).

seekg `istream& seekg(long, seek_dir);`

Moves to a position relative to the current position, following the definition: **enum seek_dir {beg, cur, end};**

tellg `long tellg();`

Returns the current stream position.

istream_withassign

<iostream.h>

This class is an **istream** with an added assignment operator.

constructor `istream_withassign();`

Default constructor (calls **istream**'s constructor).

Member functions

None (although the = operator is overloaded).

istrstream

<strstream.h>

Provides input operations on a **strstreambuf**. This class is derived from **strstreambase** and **istream**.

constructor `istrstream(const char *);`

Makes an **istrstream** with a specified string (a null character is never extracted).

constructor `istrstream(const char *, int n);`

Makes an **istrstream** using up to *n* bytes of a specified string.

ofstream

<fstream.h>

Provides input operations on a **filebuf**. This class is derived from **fstreambase** and **ostream**.

- constructor** `ofstream();`
 Makes an **ofstream** that isn't attached to a file.
- constructor** `ofstream(const char*, int = ios::out, int = filebuf::openprot);`
 Makes an **ofstream**, opens a file, and connects to it.
- constructor** `ofstream(int);`
 Makes an **ofstream**, connects to an open file descriptor.
- constructor** `ofstream(int fd, char*, int);`
 Makes an **ofstream** connected to an open file and uses a specified buffer.

 Member functions

- open** `void open(const char*, int, int = filebuf::openprot);`
 Opens a file for an **ofstream**.
- rdbuf** `filebuf* rdbuf();`
 Returns the **filebuf** used.

ostream

<iostream.h>

Provides formatted and unformatted output to a **streambuf**. The `<<` operator is overloaded for all fundamental types, as explained on page 183. This **ios**-based class is a base for **constream**, **iostream**, **ofstream**, **ostrstream**, and **ostream_withassign**.

- constructor** `ostream(streambuf *);`
 Associates a given **streambuf** with the stream.

ostream

Member functions

- flush** ostream& flush();
Flushes the stream.
- put** ostream& put(char);
Inserts the character.
- seekp** ostream& seekp(long);
Moves to an absolute position (as returned from **tellp**).
- seekp** ostream& seekp(long, seek_dir);
Moves to a position relative to the current position, following the definition: **enum seek_dir {beg, cur, end};**
- tellp** long tellp();
Returns the current stream position.
- write** ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);
Inserts *n* characters (nulls included).

ostream_withassign

<iostream.h>

This class is an **ostream** with an added assignment operator.

- constructor** ostream_withassign();
Default constructor (calls **ostream**'s constructor).

Member functions

None (although the = operator is overloaded).

ostrstream

<strstream.h>

Provides output operations on a **strstreambuf**. This class is derived from **strstreambase** and **ostream**.

constructor ostream();

Makes a dynamic **ostream**.

constructor ostream(char*, int, int = ios::out);

Makes a **ostream** with a specified *n*-byte buffer. If **mode** is **ios::app** or **ios::ate**, the get/put pointer is positioned at the null character of the string.

Member functions

pcount char *pcount();

Returns the number of bytes currently stored in the buffer.

str char *str();

Returns and freezes the buffer. You must deallocate it if it was dynamic.

streambuf

<iostream.h>

This is a buffer-handling class. Your applications gain access to buffers and buffering functions through a pointer to **streambuf** that is set by **ios**. **streambuf** is a base for **filebuf** and **strstreambuf**.

constructor streambuf();

Creates an empty buffer object.

constructor streambuf(char *, int);

Uses the given array and size as the buffer.

Member functions

allocate int allocate(); **protected**

Sets up a buffer area.

base char *base(); **protected**

Returns the start of the buffer area.

blen int blen(); **protected**

Returns the length of buffer area.

eback	<code>char *eback();</code> Returns the base of putback section of get area.	protected
ebuf	<code>char *ebuf();</code> Returns the end+1 of the buffer area.	protected
egptr	<code>char *egptr();</code> Returns the end+1 of the get area.	protected
epptr	<code>char *epptr();</code> Returns the end+1 of the put area.	protected
gbump	<code>void gbump(int);</code> Advances the get pointer.	protected
gptr	<code>char *gptr();</code> Returns the next location in get area.	protected
in_avail	<code>int in_avail();</code> Returns the number of characters remaining in the input buffer.	
out_waiting	<code>int out_waiting();</code> Returns the number of characters remaining in the output buffer.	
pbase	<code>char *pbase();</code> Returns the start of put area.	protected
pbump	<code>void pbump(int);</code> Advances the put pointer.	protected
pptr	<code>char *pptr();</code> Returns the next location in put area.	protected
sbumpc	<code>int sbumpc();</code> Returns the current character from the input buffer, then advances.	
seekoff	<code>virtual streampos seekoff(streamoff, ios::seek_dir, int = (ios::in ios::out));</code> Moves the get or put pointer (the third argument determines which one or both) relative to the current position.	

seekpos	virtual streampos seekpos(streampos, int = (ios::in ios::out)); Moves the get or put pointer to an absolute position.	
setb	void setb(char *, char *, int = 0); Sets the buffer area.	protected
setbuf	virtual streambuf* setbuf(signed char *, int); streambuf* setbuf(unsigned char *, int); Connects to a given buffer.	
setg	void setg(char *, char *, char *); Initializes the get pointers.	protected
setp	void setp(char *, char *); Initializes the put pointers.	protected
sgetc	int sgetc(); Peeks at the next character in the input buffer.	
sgetn	int sgetn(char*, int n); Gets the next <i>n</i> characters from the input buffer.	
snextc	int snextc(); Advances to and returns the next character from the input buffer.	
sputbackc	int sputbackc(char); Returns a character to input.	
sputc	int sputc(int); Puts one character into the output buffer.	
sputn	int sputn(const char*, int n); Puts <i>n</i> characters into the output buffer.	
stoss	void stoss(); Advances to the next character in the input buffer.	
unbuffered	void unbuffered(int); Sets the buffering state.	protected
unbuffered	int unbuffered(); Returns non-zero if not buffered.	protected

strstreambase

<strstrea.h>

Specializes **ios** to string streams. This class is entirely protected except for the member function **strstreambase::rdbuf()**. This class is a base for **strstream**, **istrstream**, and **ostrstream**.

constructor `strstreambase();` **protected**

Makes an empty **strstreambase**.

constructor `strstreambase(char *, int, char *start);` **protected**

Makes an **strstreambase** with a specified buffer and starting position.

Member functions

rdbuf `strstreambuf * rdbuf();`

Returns a pointer to the **strstreambuf** associated with this object.

strstreambuf

<strstrea.h>

Specializes **streambuf** for in-memory formatting.

constructor `strstreambuf();`

Makes a dynamic **strstreambuf**. Memory will be dynamically allocated as needed.

constructor `strstreambuf(void * (*)(long), void (*)(void *));`

Makes a dynamic buffer with specified allocation and free functions.

constructor `strstreambuf(int n);`

Makes a dynamic **strstreambuf**, initially allocating a buffer of at least *n* bytes.

constructor `strstreambuf(signed char *, int, signed char *end = 0);`
`strstreambuf(unsigned char *, int, unsigned char *end = 0);`

Makes a static **strstreambuf** with a specified buffer. If *end* is not null, it delimits the buffer.

Member functions

doallocate	virtual int doallocate(); Performs low-level buffer allocation.
freeze	void freeze(int = 1); If the input parameter is nonzero, disallows storing any characters in the buffer. Unfreeze by passing a zero.
overflow	virtual int overflow(int = EOF); Flushes a buffer to its destination. Every derived class should define the actions to be taken.
seekoff	virtual streampos seekoff(streamoff, ios::seek_dir, int); Moves the pointer relative to the current position.
setbuf	virtual streambuf* setbuf(char*, int); Specifies the buffer to use.
str	char *str(); Returns a pointer to the buffer and freezes it.
underflow	virtual int underflow(); Makes input available. This is called when a character is requested and the strstreambuf is empty. Every derived class should define the actions to be taken.

strstream

<strstrea.h>

	Provides for simultaneous input and output on a strstreambuf . This class is derived from strstreambase and iostream .
constructor	strstream(); Makes a dynamic strstream .
constructor	strstream(char*, int <i>n</i> , int <i>mode</i>); Makes a strstream with a specified <i>n</i> -byte buffer. If <i>mode</i> is ios::app or ios::ate , the get/put pointer is positioned at the null character of the string.

Member
function

str char *str();

Returns and freezes the buffer. The user must deallocate it if it was dynamic.

The container class libraries

For more information about templates, see Chapter 3, "C++ specifics."

Borland C++ version 3.0 includes *two* complete container class libraries: an enhanced version of the Object-based library supplied with version 2.0, plus a brand-new implementation based on templates. This chapter describes both libraries. We assume that you are familiar with the syntax and semantics of C++ and with the basic concepts of object-oriented programming (OOP). To understand the template-based version (called BIDS, for Borland International Data Structures), you should be acquainted with C++'s new template mechanism.

The chapter is divided into seven parts:

- A review of the difference between versions 2.0 and 3.0 of the class libraries
- An overview of the Object- and template-based libraries
- A survey of the Object container classes, introducing the basic concepts and terminology
- An overview of the BIDS library
- The CLASSLIB directory and how to use it
- The new debugging tools
- An alphabetic reference guide to the Object container library, listing each class and its members

What's new since version 2.0?

The version 2.0 container library is an **Object**-based implementation. Both container objects and the elements stored in them are all ultimately derived from the class **Object**. Further, the data structures used to implement each container class were fixed and (usually) hidden from the programmer. This provides a simple, effective model for most container applications. Version 3.0 therefore offers an enhanced, code-compatible version of the previous **Object**-based container library. We call this the Object container class library. In addition, a more flexible (but more complex), template-based container library, called BIDS (Borland International Data Structures), is supplied with version 3.0. Through the power of templates, BIDS lets you vary the underpinning data structure for a container and lets you store arbitrary objects in a container. With the appropriate template parameters, BIDS can actually emulate the Object container library.

Before we review the differences between the Object and BIDS models, we'll list the changes to the Object container library since version 2.0:

- New **Btree** and **PriorityQueue** classes.
- New **TShouldDelete** class gives the programmer control over container/element ownership. You can control the fate of objects when they are detached from a container and when the container is flushed (using the new **flush** method) or destroyed.
- New memory management classes, **MemBlocks** and **MemStack**, for efficient memory block and memory stack (mark-and-release) allocations.
- New **PRECONDITION** and **CHECK** macros provide sophisticated **assert** mechanisms to speed application development and debugging.
- New **Timer** class gives you a stopwatch for timing program execution (not in Microsoft Windows).
- New DLL library and support for Windows applications.

When you choose Container Class Library in the IDE's Link Libraries dialog box, the Object-based libraries will be automatically linked in.

Existing Borland C++ version 2.0 container class code will still run with the version 3.0 libraries. The new Object container class libraries, in directory /CLASSLIB, are distinguished by the prefix TC: TCLASSx.LIB, TCLASDBx.LIB, TCLASDLL.LIB, and TCLASS.DLL, where x specifies the memory model, and DB indicates the special debug versions. To reduce verbiage, we will

often refer to this container implementation as the Object or TC version.

To use the template-based libraries, you must explicitly add the appropriate BIDS(DB)x.LIB library to your project or makefile.

The corresponding libraries for the new template-based container classes are distinguished by the prefix BIDS: BIDSx.LIB, BIDSDBx.LIB, BIDSDDL.LIB, and BIDS.DLL. Let's review the reasons for having two sets of container libraries. The use of all these libraries is covered on page 238.

Why two sets of libraries?

The Object container classes have been retained and enhanced to provide code compatibility with the version 2.0 library. They provide a gentler learning curve than the template-based BIDS library. The Object container code offers faster compilation but slightly slower execution than the template version. The project files for the example and demo programs are set up to use the Object version of the container libraries.

BIDS exploits the new exciting templates feature of C++ 2.1. It offers you considerable flexibility in choosing the best underlying data structure for a given container application. With the Object version, each container is implemented with a fixed data structure, chosen to meet the space/speed requirements of most container applications. For example, a **Bag** object is implemented with a hash table, and a **Deque** object with a double list. With BIDS you can fine-tune your application by varying the container implementation with the minimum recoding—often a single typedef will suffice. You can switch easily from *StackAsList* to *StackAsVector* and test the results. In fact, you'll see that by setting appropriate values for $\langle T \rangle$, a generic class parameter, you can implement the Object model exactly. With BIDS, you can even choose between polymorphic and non-polymorphic implementations of the Object container model. Such choices between execution speed (non-polymorphic) and future flexibility (polymorphic) can be tested without major recoding.

Existing code based on the Object container classes will compile and run perfectly using the new BIDS classes, just by linking in the appropriate library.

Both the Object and BIDS versions provide the same functional interface. For example, the **push** and **pop** member functions work the same for all **Stack** objects. This makes the new template-based libraries highly compatible with existing code written for the Object library.

The objects stored in Object library containers must be derived from the class **Object**. To store **ints**, say, you would have to derive an **Integer** class from **Object** (you'll see how later). With BIDS you have complete freedom and direct control over the types of objects stored in a container. The stored data type is simply a value passed as a template parameter. For instance, `BI_ListImp<int>` gives you a list of **ints**, and `BI_IBtreeImp<Emp>` gives you a B-tree of pointers to **Emp** (a user-defined **struct**).

Regardless of which container class model you elect to use, you should be familiar with container terminology, the Object class hierarchy, and the functionality provided for each container type. Although the classes in the BIDS library have different naming conventions and special template parameters, the prototypes and functionality of each class member are the same as those in the Object library.

Container basics

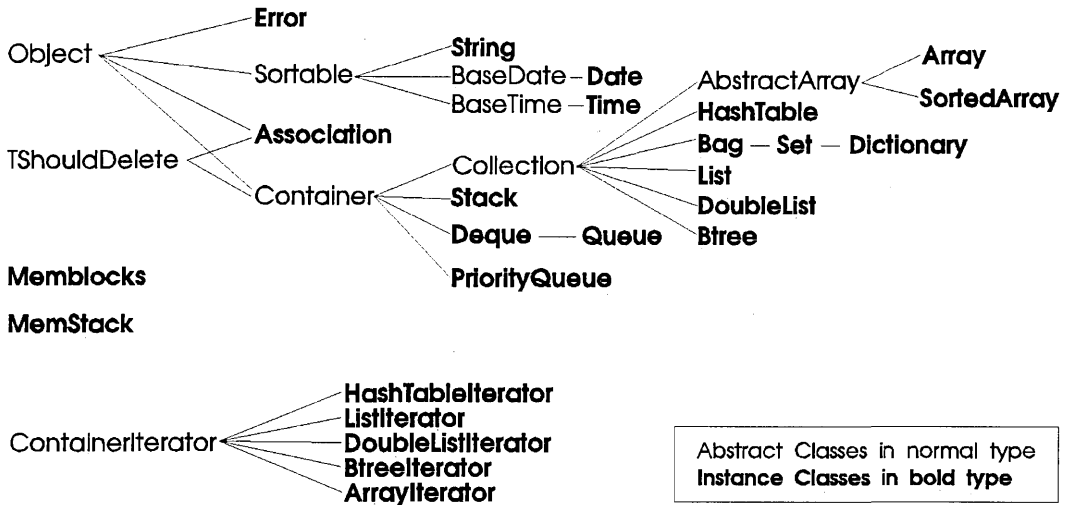
If you are fully versed in the Borland C++ 2.0 version of the container library, you should first check out the Object library enhancements before moving to the templates section on page 224.

Using ObjectBrowser in Windows, you can see the class hierarchy. Here we highlight the basic structure.

We start by describing the Object container class hierarchy as enhanced for Borland C++ version 3.0. This hierarchy offers a high degree of modularity through inheritance and polymorphism. You can use these classes as they are, or you can extend and expand them to produce an object-oriented software package specific to your needs.

At the top of the class hierarchy is the **Object** class (see Figure 6.1), an *abstract* class that cannot be *instantiated* (no objects of its type can be declared). An abstract class serves as an umbrella for related classes. As such, it has few if any data members, and some or all of its member functions are *pure virtual functions*. Pure virtual functions serve as placeholders for functions of the same name and signature intended to be defined eventually in derived classes. In fact, any class with at least one pure virtual function is, by definition, an abstract class.

Figure 6.1: Class hierarchies in CLASSLIB



Note that **TShouldDelete** provides a second base (multiple inheritance) for both **Container** and **Association**.

A class derived from an abstract class can provide a body defining the inherited pure virtual function. If it doesn't, the derived class remains abstract, providing a base for further derivations. When you reach a derived class with no pure virtual functions, the class is called a non-abstract or *instance* class. As the name implies, instance classes can be instantiated to provide usable objects.

An abstract class can be the base for both abstract and instance classes. For example, you'll see that **Container**, an abstract class derived from the abstract class **Object**, is the base for both **Collection** (abstract) and **Stack** (instance).

To enhance your understanding of the classes, you can review their declarations in the source code files in the CLASSLIB directory.

As you read this chapter, bear in mind that a derived class inherits and can access all non-private data members and member functions from all its ancestral base classes. For example, the **Array** class does not need to explicitly define a function to print an array, because its immediate parent class **AbstractArray** does so. The **Container** class, an ancestor further up the class tree, defines a different print function that can also be used with an array, because an array *is* a container. To determine all the member functions available to an object, you will have to ascend the class hierarchy tree. Because the public interface is intended to be sufficient for applications, object-oriented programming makes a knowledge of private data members unnecessary; therefore,

private members (with a few exceptions) are not documented in this chapter.

Object-based and other classes

The **Object**-based hierarchy contains classes derived from the class **Object** (together with some other utility classes). **Object** provides a minimal set of members representing what every derived object must do; these are described in the reference section under **Object** (page 279). Both the containers-as-objects and the objects they store are objects derived (ultimately) from **Object**. Later you'll see that the template-based containers can contain objects of any data type, not just those derived from **Object**.

Class categories

The classes in or near the **Object** hierarchy can be divided into three groups:

- The *non-container* classes include **Object** itself, and those classes derived from **Object**, such as **String** and **Date**, which cannot store other objects.
- The *container* classes (also derived from **Object**), such as **Array** and **List**, which can store objects of other, usually non-container, class types.
- The helper and utility classes not derived from **Object**, such as **TShouldDelete**, **ListIterator** and **MemStack**.

Let's look at each category in more detail, although as with most OOP topics, they are closely related.

Non-container classes

The basic non-container classes provided are **Object** and its three children: **Error** (instance), **Sortable** (abstract), and **Association** (instance). Recall that the main purpose of these classes is to provide objects that can be stored as data elements in containers. To this end, all **Object**-derived classes provide a hashing function allowing any of their objects to be stored in a hash table.

Error class

For details on **Error** see page 271 in the reference section.

Error is not a normal class; it exists solely to define a unique, special object called **theErrorObject**. A pointer to **theErrorObject** carries the mnemonic **NOOBJECT**. **NOOBJECT** is rather like a null pointer, but serves the vital function of occupying empty slots in a container. For example, when an **Array** object is created (not to be confused with a traditional C array), each of its elements will initially contain **NOOBJECT**.

Sortable class

Sortable is an abstract class from which sortable classes must be derived. Some containers, known as *ordered collections*, need to maintain their elements in a particular sequence. Collections such as **SortedArray** and **PriorityQueue**, for example, must have consistent methods for comparing the “magnitude” of objects. **Sortable** adds the pure virtual function **isLessThan** to its base, **Object**. Classes derived from **Sortable** need to define **isLessThan** and **isEqual** (inherited from **Object**) for their particular objects. Using these members, the relational operators **<**, **==**, **>**, **>=**, and so on, can be overloaded for sortable objects. Typical sortable classes are **String**, **Date**, and **Time**, the objects of which are ordered in the natural way. Of course, string ordering may depend on your locale, but you can always override the comparison functions (another plus for C++).

For more details on **Sortable** see page 286 in the reference section.

Distinguish between the container object and the objects it contains: **Sortable** is the base for non-container objects; it is not a base for ordered collections. Every class derived from **Object** inherits the **isSortable** member function so that objects can be queried as to their “sortability.”

Association class

For details on **Association** see page 248 in the reference section.

Association is a non-container, instance class providing special objects to be stored (typically) in **Dictionary** collections. An **Association** object, known as an *association*, is a pair of objects known as the *key* and the *value*. The *key* (which is unique in the dictionary) can be used to retrieve the *value*. Every class derived from **Object** inherits the **isAssociation** member function so that objects can report whether they are associations or not.

Container classes

In the **Object**-based library, all the container storage and access methods assume that the stored elements are derived from **Object**. They are actually stored as references or pointers to **Object** offering the advantages and disadvantages of polymorphism. Most of the container access member functions are virtual, so a container does not need to “know” how its contained elements were derived. A container can, in theory, contain mixed objects of different class types, so proper care is needed to maintain type-safe linkage. Every class has member functions called **isA** and **nameOf**, which allow objects to announce their class ID and name. As you’ve seen, there are also **isSortable** and **isAssociation** member functions for testing object types.

All the container classes are derived from the abstract **Container** class, a child of **Object**. **Container** encapsulates just a few simple properties upon which more specialized containers can be built. The basic container provides the following functionality:

- Displays its elements
- Calculates its hash value
- Pure virtual slot for counting the number of items with **getItemsInContainer**
- Pure virtual slot for flushing (emptying) the container with **flush**
- Performs iterations over its elements
- Reports and changes the ownership of its elements (inherited from **TShouldDelete**)

So far, our containers have no store, access, or detach methods. (We can flush the container but we cannot detach individual elements.) Nor is there a **hasMember** member function, that is, a general way of determining whether a given object is an element of the container. This is a deliberate design decision. As we move up the hierarchy, you'll see that what distinguishes the various derived container classes are the storage and access rules that actually define each container's underlying data structure. Thus **push** and **pop** member functions are added for **Stack**, indexing operators are added for **Array**, and so on. There is not enough in common to warrant having generic add and retrieve methods at the **Container** level. There is no one perfect way of extracting common properties from groups of containers, and therefore no perfect container class hierarchy. The **Object**-based container hierarchy is just one possible design based on reasonable compromises. The BIDS version, as you'll see, offers a different perspective.

The first three **Container** functions listed previously are fairly self-explanatory. We'll discuss the important subjects of ownership and iteration in the next two sections.

Containers and ownership

Before you use the **Container** family, you must understand the concept of ownership. As in real life, a C++ container starts out empty and must be filled with objects before the objects can be said to be in the container. Unlike the real world, however, when objects are placed in the container, they are, by default, *owned* by

the container. The basic idea is that when a container owns its objects, the objects are destroyed when the container is destroyed.

Recall that containers are themselves objects subject to the usual C++ scoping rules, so local containers come and go as they move in and out of scope. Care is needed, therefore, to prevent unwanted destruction of a container's contents, so provision is made for changing ownership. A container can, throughout its lifetime, relinquish and regain ownership of its objects as often as it likes by calling the **ownsElements** member function (inherited from **TShouldDelete**). The fate of its objects when the container disappears is determined by the ownership status ruling at the time of death. Consider the following:

```
void test()
{
    Array a1( 10 );    // construct an array
    Array a2( 10 );    // and another

    a1.ownsElements( 1 ); // array a1 owns its objects (the default)
    a2.ownsElements( 0 ); // array a2 relinquishes ownership

    // load and manipulate the arrays here
}
```

When **test** exits, *a1* will destroy its objects, but the objects in *a2* will survive (subject, of course, to their own scope). The `a1.ownsElements(1)` call is not really needed since, by default, containers own their contents.

Ownership also plays a role when an object is removed from a container. The pure virtual function **Container::flush** is declared as

```
virtual void flush( DeleteType dt = DefDelete ) = 0;
```

flush empties the container but whether the flushed objects are destroyed or not is controlled by the *dt* argument. **DeleteType** is an **enum** defined in **TShouldDelete**. A value of *NoDelete* means preserve the flushed objects regardless of ownership; *Delete* means destroy the objects regardless of ownership; *DefDelete*, the default value, means destroy the objects only if owned by the container. Similarly **Collection** (derived from **Container**) has a **detach** member function, declared as

```
virtual void detach( Object& obj, DeleteType dt = NoDelete )
= 0;
```


which looks for *obj* in the collection and removes it if found. Again, the fate of the detached object is determined by the value *dt*. Here, the default is not to destroy the detached object.

Collection::destroy is a variant that calls **detach** with *DefDelete*.

A related problem occurs if you destroy an object that resides in a container without “notifying” the container. The safest approach is to use the container’s methods to detach and destroy its contents.

Important! If you declare an automatic object (an object that’s local to your routine) and place that object in a global container, your local object will be destroyed when the routine leaves the scope in which it was declared. To prevent this, *you must only add heap objects* (objects that aren’t local to the current scope) *to global containers*. Similarly, when you remove an object from a global container, you are responsible for destroying it and freeing the space in which it resides.

Container iterators

You saw earlier that **Container**, the base for all containers in the **Object**-based library, supports iteration. *Iteration* means traversing or scanning a container, accessing each stored object in turn to perform some test or action. The separate **ContainerIterator**-based hierarchy provides this functionality. Iterator classes are derived from this base to provide iterators for particular groups of containers, so you’ll find **HashTableIterator**, **ListIterator**, **BtreeIterator**, and so on.

*Under **ContainerIterator** on page 263 in the reference section, you see how the pre- and post-increment operators ++ are overloaded to simplify your iterator programs.*

There are two flavors of iterators: internal and external. Each container inherits the three member functions: **firstThat**, **lastThat**, and **forEach**, via the **Object** and **Container** classes. As the names indicate, these let you scan through a container either testing each element for a condition or performing an action on each of the container’s elements. When you invoke one of these three member functions, the appropriate iterator object is created for you *internally* to support the iteration. Most iterations can be performed in this way since the three iterating functions are very flexible. They take a pointer-to-function argument together with an arbitrary parameter list, so you can do almost anything. For even more flexibility, there are *external* iterators that you can build via the **initIterator** member function. With these, you have to set up your own loops and test for the end-of-container.

Returning to the container class hierarchy, we look at three classes derived directly from `Container`: **Stack**, **Deque**, and **PriorityQueue**.

Sequence classes

The instance classes **Stack**, **Deque** (and its offspring **Queue**), and **PriorityQueue** are containers collectively known as *sequence* classes. A sequence class is characterized by the following properties:

1. Objects can be inserted and removed.
2. The order of insertions and deletions is significant.
3. Insertions and extractions can occur only at specific points, as defined by the individual class. In other words, access is nonrandom and restricted.

Sequences (like all containers) know how many elements they have (using `getItemInContainer`) and if they are empty or not (using `isEmpty`). However, they cannot usually determine if a given object is a member or not (there is still no general `hasMember` or `findMember` member function). Stacks, queues, priority queues, and deques vary in their access methods as explained in more detail in the reference section.

Sequence is not itself a class because sequences do not share enough in common to warrant a separate base class. However, you might find it helpful to consider the classes together when reviewing the container hierarchy.

Collections

The next level of specialization is the abstract class **Collection**, derived from **Container**, and poised to provide a slew of widely used data structures. The key difference between collections and containers is that we now have general `hasMember` and `findMember` member functions.

From **Collection** we derive the *unordered* collections **Bag**, **HashTable**, **List**, **DoubleList**, and **AbstractArray**, and the *ordered* collection **Btree**. In turn, **AbstractArray** spawns the unordered **Array** and the ordered **SortedArray**. **Bag** serves as the base for **Set** which in turn is the base for **Dictionary**. These collections all refine the storage and retrieval methods in their own fashions.

Unordered collections With unordered collections, any objects derived from **Object** can be stored, retrieved, and detached. The objects do not have to be sortable because the access methods do not depend on the relative “magnitude” of the elements. Classes that fall into this category are

- **HashTable**
- **Bag, Set, and Dictionary**
- **List and DoubleList**
- **Array**

Ordered collections An ordered collection depends on relative “magnitude” when adding or retrieving its elements. Hence these elements must be objects for which the **isLessThan** member function is defined. In other words, the elements in an ordered collection must be derived from the class **Sortable**. The following are ordered collections:

- **Btree**
- **SortedArray**

The BIDS template library

The BIDS container class library can be used as a springboard for creating useful classes for your individual needs. Unlike the Object container library, BIDS lets you fine-tune your applications by varying the underlying data structures for different containers with minimum reprogramming. This extends the power of encapsulation: the implementor can change the internals of a class with little recoding and the user can easily replace a class with one that provides a more appropriate algorithm. The BIDS class library achieves this flexibility by using the C++ **template** mechanism.

For a basic description of C++ templates see page 145 in Chapter 3.

With BIDS, the container is considered as an ADT (abstract data type), and its underlying data structure is independently treated as an FDS (fundamental data structure). BIDS also allows separate selections of the type of objects to be stored, and whether to store the objects themselves or pointers to objects.

Templates, classes, and containers

Computer science has devoted much attention to devising suitable data structures for different applications. Recall Wirth's equation, $\text{Programs} = \text{Algorithms} + \text{Data Structures}$, which stresses the equal importance of data structures and their access methods.

As used in current OOP terminology, a container is simply an object that implements a particular data structure, offering member functions for adding and accessing its data elements (usually other objects) while hiding from the user as much of the inner detail as possible. There are no simple rules to determine the best data structure for a given program. Often, the choice is a compromise between competing space (RAM) and time (accessibility) considerations, and even here the balance can shift suddenly if the number of elements or the frequency of access grows or falls beyond a certain number.

Container implementation

Often, you can implement the desired container properties in many ways using different underlying data structures. For example, a stack, characterized by its Last-In-First-Out (LIFO) access, can be implemented as a vector, a linked list, or perhaps some other structure. The vector-based stack is appropriate when the maximum number of elements to be stacked is known in advance. A vector allocates space for all its elements when it is created. The stack as a list is needed when there is no reasonable upper bound to the size of the stack. The list is a slower implementation than the vector, but it doesn't use any more memory than it needs for the current state of the stack.

The way objects are stored in the container also affects size and performance: they can be stored directly by copying the object into the data structure, or stored indirectly via pointers. The type of data to be stored is a key factor. A stack of `ints`, for example, would probably be stored directly, where large `structs` would call for indirect storage to reduce copying time. For "in-between" cases, however, choosing strategies is not always so easy. Performance tuning requires the comparison of different container implementations, yet traditionally this entails drastic recoding.

The template solution

The template approach lets you develop a stack-based application, say, using vectors as the underlying structure. You can change this to a list implementation without major recoding (a single **typedef** change, in fact). The BIDS library also lets you experiment with object-storage strategies late in the development cycle. Each container-data structure combination is implemented with both direct and indirect object storage, and the template approach allows a switch of strategy with minimal rewriting. The data type of the stored elements is also easy to change using template parameters, and you are free to use any data type you want.

As you'll see, BIDS offers container/data structure combinations that match those of the **Object**-based version. For example, the **Object** library implements **Stack** using lists, so **Stack** can be simulated exactly with the template class **BI_TCStackAsList**. Let's look at the template approach in more detail.

ADTs and FDSs

We discussed earlier the stack and its possible implementations as a linked list or as a vector. The potential for confusion is that stacks, lists, and vectors are all commonly referred to as data structures. However, there is a difference. We can define a stack abstractly in terms of its LIFO accessibility, but it's difficult to envision a list without thinking of specifics such as nodes and pointers. Likewise, we picture a vector as a concrete sequence of adjacent memory locations. So we call the stack an ADT (abstract data type) and we call the list and vector FDSs (fundamental data structures). The BIDS container library offers each of the standard ADTs implemented with a choice of appropriate FDSs. Table 6.1 indicates the combinations provided:

Table 6.1
ADTs as fundamental data structures

FDS	Stack	Queue	Deque	ADT Bag	Set	Array	Sorted Array
Vector	•	•	•	•	•	•	•
List	•						
DoubleList		•	•				

The abstract data types involved are Stacks, Queues, Deques, Bags, Sets, and Arrays. Each ADT can be implemented several different ways using the fundamental data structures Vector, List, and DoubleList as indicated by a bullet (•) in the table. Thus, all ADTs are implemented as vectors. In addition, Stacks are

implemented as a list; Queues and Deques implemented as doubly-linked lists. (Not shown in the table are the sorted and counted FDSs from which various ADTs can be developed.)

There is nothing sacred about these combinations; you can use the template classes to develop your own ADT/FDS implementations.

Class templates ADTs are implemented in both direct and indirect versions. The direct versions store the objects themselves, while the indirect versions store pointers to objects. You can store whatever objects you want as elements in these FDSs using the power of templates. Here are the ADT and FDS templates we provide:

Table 6.2
FDS class templates

Class template	Description
BI_VectorImp<T>	vector of Ts
BI_VectorIteratorImp<T>	iterator for a vector of Ts
BI_CVectorImp<T>	counted vector of Ts
BI_SVectorImp<T>	sorted vector of Ts
BI_IVectorImp<T>	vector of pointers to T
BI_IVectorIteratorImp<T>	iterator for a vector of pointers to T
BI_ICVectorImp<T>	counted vector of pointers to T
BI_ISVectorImp<T>	sorted vector of pointers to T
BI_ListImp<T>	list of Ts
BI_SListImp<T>	sorted list of Ts
BI_IListImp<T>	list of pointers to T
BI_ISListImp<T>	sorted list of pointers to T
BI_DoubleListImp<T>	double-linked list of Ts
BI_SDoubleListImp<T>	sorted double-linked list of Ts
BI_IDoubleListImp<T>	double-linked list of pointers to T
BI_ISDoubleListImp<T>	sorted double-linked list of pointers to T



Each basic FDT has a direct and indirect iterator; to save space we have shown only the Vector iterators.

The *BI_* prefix stands for Borland International and the suffix *Imp* for implementation. The indirect versions have the prefix *BI_I* with the extra *I* for Indirect. The extra prefixes *S* and *C* mean Sorted and Counted respectively. The template parameter *T* in *<T>* represents the data type of the objects to be stored. You instantiate the template by supplying this data type. For example, `BI_ListImp<double>` gives you a list of **doubles**, and `BI_IBtreeImp<Cust>` gives you a B-tree of pointers to **Cust** (a user-defined type). See Table 6.3 on page 228 for a summary of these abbreviations. For direct object storage, the type *T* must have meaningful copy semantics and a default constructor. Indirect containers, however, hold pointers to *T*, and pointers always have

good copy semantics. This means that indirect containers can contain objects of any type.

Table 6.3
Abbreviations in CLASSLIB
names

Abbreviation	Description
BI_	Borland International_
I	Indirect
C	Counted
S	Sorted
O	Object-based, non-polymorphic
TC	Object-based, polymorphic (compatible with original Turbo C library)

For details see the discussion under **Sortable** on page 287.

For the sorted FDSs (**BI_SVectorImp**, **BI_ISVectorImp**, and so on), **T** must have valid == and < operators to define the ordering of the elements. It should be clear that the IS variants refer to the objects being sorted, not that the pointers to the objects are sorted.

Each implementation of an ADT with an FDS is named using the convention *(ADT)As (FDS) <T>*, as follows:

Table 6.4
ADT class templates

Class name	Description
BI_StackAsVector<T>	Stack of Ts as a vector
BI_QueueAsVector<T>	Queue of Ts as a vector
BI_DequeAsVector<T>	Deque of Ts as a vector
BI_BagAsVector<T>	Bag of Ts as a vector
BI_SetAsVector<T>	Set of Ts as a vector
BI_ArrayAsVector<T>	Array of Ts as a vector
BI_SArrayAsVector<T>	Sorted array of Ts as a vector
BI_IStackAsVector<T>	Stack of pointers to T as a vector
BI_IQueueAsVector<T>	Queue of pointers to T as a vector
...	and so on
BI_StackAsList<T>	Stack of Ts as a list
BI_IStackAsList<T>	Stack of pointers to T as a list
BI_QueueAsDoubleList<T>	Queue of Ts as a double list
BI_DequeAsDoubleList<T>	Deque of Ts as a double list
BI_IQueueAsDoubleList<T>	Queue of pointers to T as a double list
BI_IDequeAsDoubleList<T>	Deque of pointers to T as a double list

There are also **BI_Oxxx** and **BI_TCxxx** variants discussed soon.

Again, the <T> argument, either a class or predefined data type, provides the data type for the contained elements. Each of the bullets (•) in Table 6.1 can be mapped to two templates (direct and indirect versions) with names following this convention.

Container class compatibility

Each template must be instantiated with a particular data type as the type of the element that it will hold. This allows the compiler to generate the correct code for dealing with any possible type of element without restricting the elements to just those derived from **Object**.

Each ADT is also used to instantiate two classes that imitate the behavior of the Object class libraries. Here is a list of them:

Table 6.5
Object-based FDS classes

Class name	Description
BI_OStackAsVector	Stack of pointers to Object, as a vector
BI_OQueueAsVector	Queue of pointers to Object, as a vector
BI_ODequeAsVector	Deque of pointers to Object, as a vector
BI_OBagAsVector	Bag of pointers to Object, as a vector
BI_OSetAsVector	Set of pointers to Object, as a vector
BI_OArrayAsVector	Array of pointers to Object, as a vector
BI_OSArrayAsVector	Sorted array of pointers to Object, as a vector
BI_TCStackAsVector	Polymorphic stack of pointers to Object, as a vector
BI_TCQueueAsVector	Polymorphic queue of pointers to Object, as a vector
BI_TCDequeAsVector	Polymorphic deque of pointers to Object, as a vector
BI_TCBagAsVector	Polymorphic bag of pointers to Object, as a vector
BI_TCSetAsVector	Polymorphic set of pointers to Object, as a vector
BI_TCArrayAsVector	Polymorphic array of pointers to Object, as a vector
BI_TCSArrayAsVector	Polymorphic sorted array of pointers to Object, as a vector
BI_OStackAsList	Stack of pointers to Object, as a list
BI_TCStackAsList	Polymorphic stack of pointers to Object, as a list
BI_OQueueAsDoubleList	Queue of pointers to Object, as a double list
BI_ODequeAsDoubleList	Deque of pointers to Object, as a double list
BI_TCQueueAsDoubleList	Polymorphic queue of pointers to Object, as a double list
BI_TCDequeAsDoubleList	Polymorphic deque of pointers to Object, as a double list

*The **TCxxx** versions offer the same behavior and interfaces as the Object library.*

Note that these versions have no explicit $\langle T \rangle$ parameters; they use the fixed data types shown (pointers to **Object**). The **BI_Oxxx** (O for Object library) versions of these classes have no virtual functions. This makes it easier for the compiler to generate inline function expansions, which in turn makes the **BI_Oxxx** versions of the containers somewhat faster than the corresponding polymorphic **BI_TCxxx** (TC for Turbo C) versions. The obverse of the coin is that the **BI_Oxxx** versions do not share the polymorphic behavior of the Object container library.

In the Object container library, **Stack** implements a stack as a polymorphic list of pointers to **Object**. The BIDS class **BI_TCStackAsList** therefore mirrors the Object-based class **Stack**. Even with **BI_TCStackAsVector**, the public interface and semantics are the same as for the Object-based **Stack**. The user “sees” the ADT while the FDS is “hidden.” For these reasons, we will not repeat the alphabetic list of Object-based classes and member functions for the BIDS library.

Consider your many choices when writing container code with the BIDS model. You can gain speed over future flexibility by using the non-polymorphic classes, such as **BI_OStackAsList** or **BI_OStackAsVector**. Or you can retain the polymorphism of the Object-based hierarchy by using the **BI_TCxxx** classes.

Header files

Each group of FDSs is defined in its own header file, which contains templates for both the direct and the indirect versions. The names of the headers are as follows:

```
vectimp.h  
listimp.h  
dlistimp.h
```

In `vectimp.h`, for example, you’ll find declarations for all the vector, counted vector, and sorted vector templates, together those for a direct and indirect vector iterator.

Note also the `stdtempl.h` file that defines the useful template functions **min**, **max**, and **range**. If you are new to templates, this file offers a useful, gentle introduction to the subject.

Each ADT family is defined in its own header file, named as follows:

stacks.h
queues.h
deques.h
bags.h
sets.h
arrays.h

Note the plural form that distinguishes the BIDS include files from the Object-based include file

The file `stacks.h`, for example, defines the following templates:

```
BI_StackAsVector<T>  
BI_IStackAsVector<T>  
BI_OStackAsVector  
BI_TCStackAsVector  
BI_StackAsList<T>  
BI_IStackAsList<T>  
BI_OStackAsList  
BI_TCStackAsList
```

Tuning an application

Consider the following example:

```
typedef BI_StackAsVector<int> intStack;  
  
int main()  
{  
    intStack is;  
    for( int i = 0; i < 10; i++ )  
        is.push( i );  
    for( i = 0; i < 10; i++ )  
        cout << is.pop() << endl;  
    return(0);  
}
```

Here we are implementing a stack of `ints` using a vector as the underlying data structure. If you later determine that a list would be a more suitable implementation for the stack, you can simply replace the **typedef** with the following:

```
typedef BI_StackAsList<int> intStack;
```

After recompilation, the stack implementation is changed from vector to list. Similarly, you can try a stack of pointers to `int`, with:

```
typedef BI_IStackAsList<int> intStack;
```

FDS implementation

Each FDS is implemented as two templates, one that provides the direct version, and one that provides the indirect version. The indirect version makes use of an **InternallxxxImp** class. The

following simplified extract from `listimp.h` will give you an idea how the different list FDSs are implemented. Note that **BI_ListElement<T>** is an internal template class used to implement the node (data of type *T* and pointer to next node) of a list. The direct list of objects of type *T* is implemented by the template class **BI_ListImp<T>**, which also provides the base for **BI_SListImp<T>** (sorted lists). The example shows how the **add** member function is implemented in the direct, indirect, sorted and unsorted lists.

```

template <class T> class BI_ListElement
{
public:
    BI_ListElement( T t, BI_ListElement<T> *p ) : data(t)
        { next = p->next; p->next = this; }
// constructor
    ...
    BI_ListElement<T> *next;    // pointer to next node
    T data;                    // object at node
    ...
};

template <class T> class BI_ListImp
// linked list (unsorted) of type T objects; assumes T has meaningful
// copy semantics and a default constructor
{
public:
    ...
    void add( T t ) { new BI_ListElement<T>( t, &head ); }
// adds objects at head of list (shown inline here to save space)
    T peekHead() const { return head.next->data; }
    ...
};

template <class T> class BI_SListImp : public BI_ListImp<T>
// sorted list; assumes T has meaningful copy semantics and a default
// constructor
{
public:
    ...
    void add( T t ) { new BI_ListElement<T>( t, findPred(t) ); }
// adds object in sorted position
    ...
};

template <class T, class List> class BI_InternalIListImp :
    public List {
    ...
    void add( T *t ) { List::add ( t ); }
};

```

```

    };
    // The work is done in this intermediate class used as base for
    // BI_IListImp; list is unsorted so we use List::add

    template <class T> class BI_IListImp :
        public BI_InternalIListImp<T, BI_ListImp< void * > >
        { ... };
    /* unsorted list of pointers to objects of type T; since pointers
     * always have meaningful copy semantics, this class can handle any
     * object type; add comes straight from BI_InternalIListImp
     */
    template <class T> class BI_ISListImp :
        public BI_InternalIListImp<T>, BSListImp< void * >> { ... };
    /* sorted list of pointers to objects of type T; since pointers
     * always have meaningful copy semantics, this class can handle any
     *
     * object type;
     */

```

In addition to the template classes shown here, `listimp.h` also declares **BI_ListIteratorImp<T>** and **BI_IListIteratorImp<T>**, the iterators for direct and indirect lists.

In the next section on ADTs, you'll see how the different stack implementations in `stacks.h` pull in the vector and list FDSs declared in `vectimp.h` and `listimp.h`.

The double list templates, in `dlistimp.h`, follows the same pattern. The sorted versions of list and double list provide exactly the same interface as the non-sorted ones, except that the **add** member function adds new elements in sorted order. This speeds up subsequent access and also makes it easier to implement priority queues.

`vectimp.h` also follows a similar pattern to `listimp.h`, implementing **BI_VectorImp<T>** (direct) and **BI_IVectorImp<T>** (indirect). These are low-level vectors with no notion of **add** or **detach**. To support more sophisticated ADTs, the counted vector, **BI_CVectorImp<T>**, derived from **BI_VectorImp<T>**, is provided. This maintains a pointer to the last valid entry in the underlying Vector. It has an **add** member function that inserts its argument at the top (the next available slot), and a **detach** member function that removes its argument and compresses the array. **BI_CVectorImp<T>** provides the base for the sorted vector template **BI_SVectorImp<T>**. With a sorted vector, you can run through the indices from 0 to the last valid entry, and the objects will emerge in sort order. Here's a simplified extract from `vectimp.h`:

```

// extract from vectimp.h

    template <class T> class BI_VectorImp { ... };
// direct uncounted, unsorted vector

    template <class T> class BI_CVectorImp : public BI_VectorImp<T>
// direct counted, unsorted vector
    {
    public:
        ...
        void add( T t );
// add at top of array; inc count; resize array if necessary
        void detach( T t, int del = 0 );
        void detach( unsigned loc, int del = 0 );
// detach given object or object at loc
        ...
    };

    template <class T> class BI_SVectorImp : public
BI_CVectorImp<T>
// direct counted, sorted vector
    {
    public:
        void add( T t );
// add at position that maintains sort
    };

    template <class T, class Vect> class BI_InternalIVectorImp :
    public Vect {...};
// interdiate base for BI_IVectorImp: no add

    template <class T> class BI_IVectorImp :
    public BI_InternalIVectorImp<T, BI_VectorImp<void *> >
    {...};
// indirect uncounted, unsorted vector: no add

    template <class T, class Vect> class BI_InternalICVectorImp :
    public BI_InternalIVectorImp<T, Vect>
// intermediate base for BI_ICVector
    {
    public:
        void add( T *t) { Vect::add(t); }
        ...
    };

    template <class T> class BI_ICVectorImp :
    public BI_InternalICVectorImp<T, BI_CVectorImp<void *> >
    { ... };
// indirect counted vector; can contain any object type

    template <class T> class BI_ISVectorImp :
    public BI_InternalICVectorImp<T, BI_SVectorImp<void *> >

```

```

    { ... };
    // indirect sorted vector

```

ADT implementation

Each ADT is implemented as several templates. For example, the following provides an implementation of a stack of objects of type *T* using vectors as the FDS:

```

// simplified extract from stacks.h

template <class Vect, class T> class BI_StackAsVectorImp
{
public:
    ...
    void push( T t ) { data[current++] = t; }
    ...
protected:
    Vect data;
    unsigned current;
};

```

The first parameter, **class *Vect***, is either a direct vector or an indirect vector, depending on whether the stack being created is direct or indirect, so *Vect* will be either **BI_VectorImp<*T0*>** or **BI_IVectorImp<*T0*>**. The type *T* represents the type of objects to be stored in the stack. For a direct *Vect*, *T* should be the same as *T0*; for an indirect *Vect*, *T* must be of type pointer to *T0*. A direct stack implemented as a vector looks like this:

```

template <class T> class BI_StackAsVector :
    public BI_StackAsVectorImp< BI_VectorImp<T>, T >
{
public:
    friend class BI_StackAsVectorIterator<T>;
    ...
};

template <class T> class BI_StackAsVectorIterator :
    public BI_VectorIteratorImp<T> {...};

```

That is, a **BI_StackAsVector** is implemented by using a **BI_StackAsVectorImp**, whose “implementation” is of type **BI_VectorImp<*T*>**, and whose elements are of type *T*. **BI_StackAsVector** has its own iterator, derived from underpinning FDS iterator with the contained-object type *T* as parameter.

An indirect stack implemented as a vector looks like this:

```

template <class T> class BI_IStackAsVector :

```

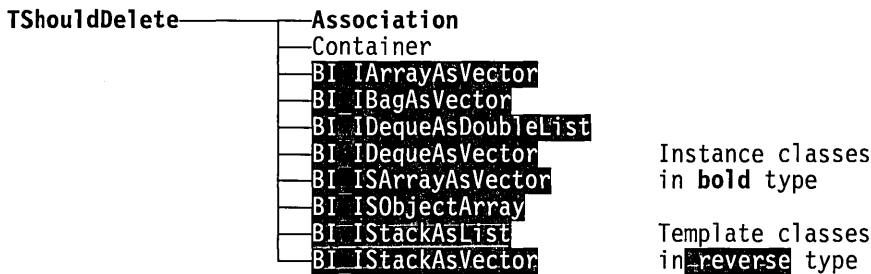
```

        public BI_StackAsVectorImp< BI_IVectorImp<T>, T* >,
        public virtual TShouldDelete
    {...};

```

That is, an **BI_IStackAsVector** is implemented by using a **BI_StackAsVectorImp**, whose “implementation” is of type **BI_IVectorImp<T>**, and whose elements are of type pointer to *T*. The **TShouldDelete** base provides the ownership control discussed in the Object-based class reference section. **TShouldDelete** also serves as a second base for the following classes.

Figure 6.2: TShouldDelete hierarchy



The **BI_OStackAsVector** and **BI_TCStackAsVector** versions (stacks of pointers to **Objects**, emulating the Object container library) now follow easily:

```

    class BI_OStackAsVector
    // non-polymorphic stack with vector of pointers to Objects
    {
    public:
        ...
        void push( Object *t ) { ostack.push(t); }
    // ostack is type BI_IStackAsVector<Object>
    // so we are pushing pointers to Object
        ...
    private:
        BI_IStackAsVector<Object> ostack;
    };

    class BI_TCStackAsVector : public Container
    // polymorphic stack with vector of pointers to Objects
    // inherits from the Object-based Container class
    // Provides identical interface and functionality as Object-based
    Stack // class // class but underlying data structure is Vector not List
    {
    public:

```

```

...
    void push( Object& o ) { stk.push( &o ); }
// stk is type BI_OStackAsVector
// so we are pushing Objects
...
private:
    BI_OStackAsVector stk;
};

```

We end the section with some short examples using the BIDS classes.

Source

Uses the template facility to pick a specific FDS for the array ADT.

In the "sorted array" FDS, the index of a particular array element depends on its value, not on the order in which it was entered.

*If the ADT used **BI_ArrayAsVector<String>**, the elements would appear in the order they were added to the array.*

```

#include <iostream.h>
#include <strstream.h>
#include <arrays.h>
#include <strng.h>

int main()
{
    typedef BI_SArrayAsVector<String> lArray;
    larray a(2);
    for (int i = a.arraySize(); i; i--)
    {
        ostrstream os;
        os << "string " << i << ends;
        a.add( *(new String(os.str())); )
    }
    cout << "array elements;\n";
    for (i = 0; i < a.arraySize(); ++i)
    {
        cout<< a[i] << endl;
    }
    return(0);
}

```

Output

```

string 1
string 2
string 3

```

Source

Doubly-linked list with indirect storage as FDS.

Pointers to String objects in the deque container must be dereferenced when extracting from the deque.

```

#include <iostream.h>
#include <strstream.h>
#include <deque.h>
#include <strng.h>

typedef BI_IDequeAsDoubleList<String> lDeque;

int main()
{
    lDeque d;
    for (int i = 1; i < 4; i++)
    {

```



```

ostream os;
os << "string " << i << ends;
// use alternating left, right insertions
if(i&1)
    d.putLeft(new String(os.str()));
else
    d.putRight(new String(os.str()));
}
cout << "Deque Contents:" << endl;
while (!d.isEmpty())
{
    cout << *d.getLeft() << endl;
}
return(0);
}

```

Output

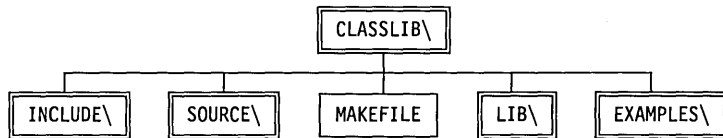
```

Deque Contents:
string 3
string 1
string 2
string 4

```

The class library directory

The files in the class library are set up in the following directory structure:



The CLASSLIB directory is under the BORLANDC directory. The contents of the directories in the class library are discussed in more detail in the following sections.

The INCLUDE directory

The INCLUDE directory contains the header files necessary to compile a program that uses the class library. You must put this directory on the include search path when you compile your program. Modify Options | Directories | Include if you have changed the default setup.

For each BIDS ADT (abstract data type), such as `Stack`, there is a header file called `stacks.h`. The Object-based class **Stack** is declared in `stack.h`. If the identifier `TEMPLATES` is `#defined`, either in an `.h` file or via the command line `_D` option, then when `stack.h` is preprocessed, the Object-based declarations for **Stack** are bypassed and the template versions are included. In particular, if `TEMPLATES` is `#defined`, **Stack** is `#defined` as **BI_TCStackAsList**, so any code written for the Object-based **Stack** will be compiled with the BIDS version.

The SOURCE directory

The SOURCE directory contains the source files that implement many of the member functions of the classes in the library. These source files are provided as a guide for implementing new classes.

You also need these source files if you want to build a library in a different memory model than the one provided on the release disk. The supplied makefile builds a class library of the specified memory model and places that library in the LIB directory. To build a library using the large memory model, type `make -DMDL=x` (where `x` is `s`, `c`, `m`, `d`, or `h`) in the SOURCE directory. Use `-DDBG` for the debug version. When you enter either command, the makefile invokes the compiler to build all the files in the class library using the large model. Then the library file archiver, `TLIB`, will create a library `TCLASSL.LIB` in `CLASSLIB\LIB`.

Important!

When you take a library that you have built and use it in one of the sample projects, you must update the project. See Chapter 4, “Managing multi-file projects” in the *User’s Guide* for more information. You must also be sure to compile your project with precisely the same switches and options you used to build the library. If you don’t have the same options, you will get warnings from the linker when the executable file is built.

The LIB directory

The LIB directory contains the compiled source modules archived into a library. You must put this directory on the library search path when you link your program. For information about modifying the library search path, see Chapter 3, “The Options menu,” or Chapter 5, “The command-line compiler” (for IDE or command-line options) in the *User’s Guide*.

The Object-based container classes are in `TCLASSx.LIB`, where `x` is the memory-model designation (`S` for small, `C` for compact, `M`

for medium, and L for large). For each of these there are debugging versions TCLASDBx.LIB. For Windows applications, you must link your code with TCLASDLL.LIB and TCLASS.DLL.

The EXAMPLES directory

The EXAMPLES directory contains the example programs and their project files. You can compile these programs to see how the parts of the class library are put together to form an application. Most of the examples use one or two of the classes in the hierarchy; other examples are more complex. Here is a list of the example programs and the classes that they use:

1. STRNGMAX: A very simple example using **String**.
2. REVERSE: An intermediate example using **Stack** and **String**.
3. LOOKUP: An intermediate example using **Dictionary** and **Association**.
4. QUEUETST: An intermediate example using **Queue** and introducing a non-hierarchical class, **Time**.
5. DIRECTORY: An advanced example illustrating derived user classes with **SortedArray**.

Preconditions and checks

*See the Library Reference for a definition of the function **assert**.*

Version 3.0 offers some new debugging tools. The class libraries TCLASDBx.LIB and BIDSDBx.LIB (where x represents the memory model, S, C, L, etc) provide the debugging versions of TCLASSx.LIB and BIDSx.LIB.

checks.h defines two macros, PRECONDITION(*arg*) and CHECK(*arg*). Each macro takes an arbitrary expression as an argument, just like **assert**. At runtime, if the expression evaluates to 0, an error message is displayed and execution terminates. If the expression evaluates to a nonzero value, execution continues in the normal fashion.

Use PRECONDITION on entry to a function to check the validity of the arguments and to do any other checking to determine that the function has been invoked correctly.

Use CHECK for internal checking during the course of execution of the function.

Compilation of `PRECONDITION` and `CHECK` is controlled by the value of a manifest constant named `__DEBUG`. If `__DEBUG` has the value 0, `PRECONDITION` and `CHECK` are set to empty macros. In other words, setting `__DEBUG` to 0 removes all debugging. If `__DEBUG` has the value 1, `PRECONDITION` macros are expanded into the tests described above, but `CHECK` macros are empty. So, setting `__DEBUG` to 1 enables `PRECONDITION`s and disables `CHECK`s. Setting `__DEBUG` to 2 or more, or not defining it at all, enables both forms of testing. Table 6.6 summarizes the available debugging modes:

Table 6.6
Class debugging modes

<code>__DEBUG</code>	<code>PRECONDITION</code>	<code>CHECK</code>
0	Off	Off
1	On	Off
>1	On	On
undefined	On	On

When developing a class, set `__DEBUG` to 2 or leave it undefined. This gives you maximum checking when the code is still being worked on. When the class works properly, but the application that is going to use the class hasn't been completed, set `__DEBUG` to 1, so that incorrect calls from the application can be caught, without the additional overhead of the internal checking within the class. Once everything is working, set `__DEBUG` to 0 to remove all checking. Two versions of the `.LIB` file are provided that contain the class library code: one with `PRECONDITION`s enabled, and one with no debugging. These are named `TCLASDBX.LIB` and `TCLASSX.LIB`, where `X` is replaced with the letter for the appropriate memory model: `s`, `c`, `m`, `l`, or `h`. The `.LIB` with `DB` in its name is the one with `PRECONDITION`s enabled.

Container class reference

This section describes each class in the library as follows. We give the include file where it is defined, a diagram showing the parent of each class and immediate offspring, some introductory remarks, data members and member functions (with prototypes) listed alphabetically, what friendly relations exist, and, where appropriate, an example of the class's use. The members listed in the *See also* section belong to the class under discussion unless scope-qualified. Thus in the section on class `X`, you could find *See also* `foo`, `Y::foo`, and so on. The first `foo` refers to `X::foo`. Class

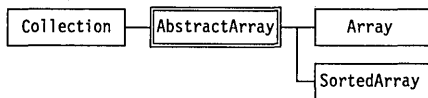
derivations and class members are **public** unless otherwise noted as **protected**. We do not document destructors since they all perform the usual way. Most container classes have virtual destructors.

Some modifiers used for Windows compatibility are not shown in the reference section

The modifiers `_FAR`, `_CLASSDEF`, and `_CLASSTYPE`, defined in `_defs.h`, are used widely in prototypes and definitions to provide Windows compatibility. To avoid clutter, we do not show them in the following prototypes. For example, `Object _FAR & foobar();` in a header file would appear as `Object& foobar();` in the reference section.

AbstractArray

abstarray.h



The abstract class **AbstractArray** offers random access to the elements of the collection via an indexing mechanism that maps a range of integers to the array elements. Indexes can be positive or negative integers with arbitrary lower and upper bounds (within the range of `int`). Arrays derived from **AbstractArray** can be dynamically resized as elements are added to them. The data member *delta* determines how many additional elements are assigned to the array when overflow occurs. **AbstractArray** exists because the derived classes **SortedArray** and **Array** have enough in common to warrant combining the common properties into an abstract base class. Since the derived classes differ only in the implementation of the member functions **detach** and the subscript operator, the remaining functions can be encapsulated in **AbstractArray**.

Data members

delta

`sizeType delta;` **protected**

delta represents the additional number of elements that will be assigned to the array if overflow occurs. If *delta* is zero, the array will not be resized following overflow.

lastElementIndex

`int lastElementIndex;` **protected**

The index value of the last element added to the array. For an empty array this data member has the value (*lowerbound* - 1).

lowerbound `int lowerbound;` **protected**

The lower bound of the array index, returned by the **lowerBound** member function. *lowerbound* is the minimum legal value of the absolute index.

See also: **lowerBound**

upperbound `int upperbound;` **protected**

The current upper bound of the array index, returned by the **upperBound** member function. *upperbound* is the maximum legal value of the absolute index.

See also: **upperBound**

Member functions

destroy `void destroy(int atIndex);`

Removes the object at the given index. Whether the object itself is destroyed or not depends on the array's ownership status. If the array currently *owns* the object, the object will be destroyed, otherwise the object survives. **destroy** is implemented with `detach(atIndex, DefDelete)`.

arraySize `sizeType arraySize() const;`

Returns the current number of cells allocated (*upperbound* – *lowerbound* + 1).

constructor `AbstractArray(int anUpper, int aLower = 0, sizeType aDelta = 0);`

Constructs and “zeroes” an array, given the upper and lower index bounds. The default lower bound is 0, the traditional origin for C arrays. The default *delta* is also zero, giving a fixed, nonresizable array. If *delta* is nonzero, run-time array overflow invokes the **reallocate** member function to provide more space (in increments of *delta*). A PRECONDITION is set to test if the lower bound is greater than or equal to the lower bound.

detach `virtual void detach(int atIndex, DeleteType dt = NoDelete);`
`virtual void detach(Object& toDetach, DeleteType dt = NoDelete);`

The first version removes the object at *atIndex*; the second version removes the object *toDetach*. The value of *dt* and the current ownership setting determine whether the object itself will be deleted. *DeleteType* is defined in the base class **TShouldDelete** as enum { NoDelete, DefDelete, Delete }. The default value of *dt*, *NoDelete*, means that the object will not be deleted regardless of ownership. With *dt* set to *Delete*, the object will be deleted

AbstractArray

regardless of ownership. If *dt* is set to *DefDelete*, the object will only be deleted if the array owns its elements.

See also: **TShouldDelete::ownsElements**

initIterator virtual ContainerIterator& initIterator() const;

Creates an external iterator for this array.

See also: **ContainerIterator** class

isEqual int isEqual(const Object& testObject) const;

Returns 1 if the *testObject* array is equal to the calling array. Equal means that the two arrays have the same object ID, the arrays' dimensions are equal, and that their components are equal in each index position. Otherwise, **isEqual** returns 0.

lowerBound int lowerBound() const;

Returns the array's *lowerbound*.

objectAt Object& objectAt(int atIndex) const; **protected**

Returns a reference to the element at the given index.

See also: **operator []**

operator () Object& operator [] (int atIndex) const;

Returns a reference to the object at the given array index.

printContentsOn void printContentsOn(ostream& outputStream) const;

Prints an array, with header and trailer, to the given stream.

ptrAt Object *ptrAt(int atIndex) const; **protected**

Returns a pointer to the element at the given index.

reallocate void reallocate(sizeType newSize); **protected**

If *delta* is zero, **reallocate** gives an `__EEXPANDFS` error. Otherwise, **reallocate** tries to create a new array of size *newSize* (adjusted upwards to the nearest multiple of *delta*). The existing array is copied to the expanded array and then deleted. Unused elements in the new array are zeroed. An `__ENOMEM` error is invoked if there is insufficient memory for the reallocation.

removeEntry void removeEntry(int loc); **protected**

Reduces the array by one element. Elements from index (*loc* + 1) upwards are copied to positions *loc*, (*loc* + 1), and so on. The original element at *loc* is lost.

setData void setData(int loc, Object *data); **protected**

The given *data* replaces the existing element at the index *loc*.

squeezeEntry void squeezeEntry(int squeezePoint); **protected**

Reduces the array by one element. As for **removeEntry** but *squeezePoint* is an index relative to the lower bound

upperBound int upperBound() const;

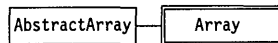
Returns the array's current *upperbound*.

Friends

ArrayIterator is a friend of **AbstractArray**

Array

array.h



The instance class **Array** is derived from class **AbstractArray**. An **Array** object defines an array in which the ordering of the elements is arbitrary. That is, the element at index *i* of the array need have no relationship to the element at index *i + 1*.

Array adds the functions **add** and **addAt**. While **add** stores a given object at the next free place in the array (expanding the array if necessary), **addAt** stores the object at a specified index.

Example

Source

```

#include <iostream.h>
#include <array.h>
#include <strng.h>
#include <assoc.h>

int main()
{
    Array a(2);

    String *s1 = new String("a string");
    String *s2 = new String("another string");
    Association *a1 = new Association(*s1,*s2);

    // Put some objects in the array
    a.add(*s1);
    a.add(*s2);
  
```


Array

```
a.add(*a1);

// Print as a Container
cout << "As a container:\n" << a << endl << endl;

// Print as an Array
cout << "As an array:\n";
a.printContentsOn(cout);

// Print as elements
cout << "\nAs elements:\n";
for (int i = 0; i < a.arraySize(); ++i)
    cout << a[i] << endl;
return(0);
}
```

Output

```
As a container:
Array { a string,
       another atring,
       Association { a string, another string }
}

As an array:
Array { a string,
       another atring,
       Association { a string, another string }
}

As elements:
a string
another string
Association { a string, another string}
```

Member functions

add virtual void add(Object& toAdd);

Adds the given object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and *delta* is nonzero, the array is expanded (by sufficient multiples of *delta* bytes) to accommodate the addition. If *delta* is zero, overflow gives an error.

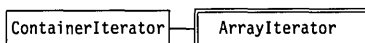
addAt void addAt(Object& toAdd, int atIndex);

Writes the given object at the specified index. If that index is occupied, the previous object is deleted. If *atIndex* is beyond the upper bound, the array is expanded if *delta* is nonzero. If *delta* is zero, attempting to **addAt** beyond the upper bound gives an error.

- constructor** `Array(int anUpper, int aLower = 0, sizeType Delta = 0);`
 Constructs and “zeroes” an array by calling the base **AbstractArray** constructor.
 See also: **AbstractArray::AbstractArray**
- isA** `virtual classType isA() const;`
 Returns *arrayClass*, the **Arrays** type ID.
- nameOf** `virtual char *nameOf() const;`
 Returns “Array”, the **Array** type ID string.

ArrayIterator

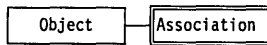
abstarry.h



Provides iterator functions to traverse objects of the class **AbstractArray** and its derived classes. **ArrayIterator** is a friend class of **AbstractArray**

Member functions

- constructor** `ArrayIterator(const AbstractArray& toIterate);`
 Creates an iterator object for the given array.
 See also: **restart**
- current** `virtual Object& current();`
 Returns the object at the current index of the iterator. If the current index doesn't refer to a valid object, NOOBJECT is returned.
- operator ++** `virtual Object& operator ++ ();`
`virtual Object& operator ++ (int);`
 See **ContainerIterator operator ++**
- operator int()** `virtual operator int();`
 Conversion operator to test for end of iterator position.
- restart** `virtual void restart();`
 Sets the current index of the iterator to the first nonempty object in the array.



The **Association** class provides a simple mechanism for associating two objects, known as the *value* object and the *key* object, in one **Association** type object. These combined objects are typically stored in a **Dictionary** type object, which provides member functions to retrieve the value when given the key, providing the basic tools for many data-retrieval applications.

Member functions

- constructor** `Association(Object& key, Object& value);`
 Constructs an association object from the given key and value objects.
- constructor** `Association(const Association& a);`
 Copy constructor.
- hashValue** `virtual hashValueType hashValue() const;`
 Returns the hash value of the association's key. See **HashTable::hashValue** for more details.
- isA** `virtual classType isA() const;`
 Returns *associationClass*, the **Association** type ID.
- isAssociation** `virtual int isAssociation() const;`
 Returns 1 for association objects (and 0 for other object types).
- isEqual** `virtual int isEqual(const Object& toObject) const;`
 Returns 1 if *toObject* and the calling association have equal keys, otherwise returns 0.
- key** `Object& key() const;`
 Returns the key object of the association.
- nameOf** `virtual char *nameOf() const;`
 Returns "Association", the **Association** type ID string.
- printOn** `virtual void printOn(ostream& outputStream) const;`

operator << is a friend of *Object*. See page 281.

Prints the association on the given output stream. **printOn** is really for internal use by the overloaded operator <<.

value Object& value() const;

Returns the value object of the association.

Example

Source

```
// File TASSOC.CPP: Illustrates the Association class

#include <string.h>    // For strlen()
#include <string.h>
#include <assoc.h>
#include <iostream.h>

void identify(Object&);

main()
{
    char s1[21], s2[81];

    // Read a key
    cout << "Enter a key: ";
    cin >> s1;
    cin.get();        // Eat newline
    String str1(s1);
    identify(str1);

    // Read a value
    cout << "Enter a value: ";
    cin.getline(s2,81);
    s2[strlen(s2) - 1] = '\0';
    String str2(s2);
    identify(str2);

    Association a1(str1,str2);
    identify(a1);
    Association a2 = a1;
    identify(a2);

    cout << "Equal: " << a1.isEqual(a2) << endl;
}

void identify(Object& o)
{
    // Echo an object and its type
    cout << "Value: " << o
         << ", Object type: " << o.nameOf()
         << endl << endl;
}
```

Output

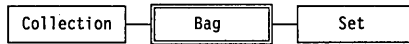
```
Enter a key: class
```

```
Value: class, Object type: String
Enter a value: A group of related objects
Value: A group of related objects, Object type: String

Value: Association { class, A group of related objects }
, Object type: Association

Value: Association { class, A group of related objects }
, Object type: Association

Equal: 1
```



A **Bag** is an unordered collection that may contain more than one of the same object. **Bag** also provides the base class for **Set**. Unlike **Bags**, **Sets** can contain only one copy of a any given object.

Member functions

add virtual void add(Object& toAdd);

Adds the given object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and *delta* is nonzero, the array is expanded (by sufficient multiples of *delta* bytes) to accommodate the addition. If *delta* is zero, overflow gives an error.

constructor Bag(sizeType bagSize = DEFAULT_BAG_SIZE);

Constructs an empty bag. *bagSize* represents the initial number of slots allocated.

detach virtual void detach(Object& toDetach, DeleteType dt = NoDelete);

See **Array::detach**.

findMember virtual Object& findMember(Object& toFind) const;

Returns the given object if found, otherwise returns NOOBJECT.

firstThat virtual Object& firstThat(condFuncType testFuncPtr, void *paramList) const;

See also: **Container::firstThat**, **Object::firstThat**

flush void flush(DeleteType dt = DefDelete);

Removes all the elements from the bag without destroying the bag. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the bag determines their fate, as explained in the **detach** member function. You can also set *dt* to *Delete* and *NoDelete*.

See also: **detach**

forEach void forEach(void (*actionFuncPtr)(Object& o, void *), void *args);

See also: **Container::forEach**

getItemsInContainer countType getItemsInContainer() const;

Returns the number of items in the bag.

hasMember virtual int hasMember(const Object& obj) const;

Returns 1 if the given object is found in the bag, otherwise returns 0.

initIterator ContainerIterator& initIterator() const;

Creates and returns an iterator for this bag.

See also: **ContainerIterator** class

isA virtual classType isA() const;

Returns *bagClass* the **Bag** type ID.

isEmpty int isEmpty() const;

Returns 1 if a container has no elements; otherwise returns 0.

lastThat virtual Object& lastThat(condFuncType testFuncPtr, void *paramList) const;

Returns a reference to the last object in the container that satisfies a given condition. You supply a *testFuncPtr* that returns true for a certain condition. You can pass arbitrary arguments via the *paramList* argument. NOOBJECT is returned if no object in the container meets the condition. Note that you are not involved directly with iterators: **firstThat** and **lastThat** create their own internal iterators, so you can simply treat them as “search” functions.

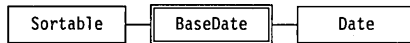
See also: **firstThat, Object::firstThat, Container::lastThat**

nameOf virtual char *nameOf() const;

Returns “Bag”, the **Bag** type ID string.

ownsElements int ownsElements();
void ownsElements(int del);

See **TShouldDelete::ownsElements**



BaseDate is an abstract class derived from **Sortable** that provides basic date manipulation functions.

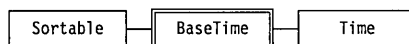
Member functions

- | | | |
|--------------------|--|------------------|
| constructor | BaseDate(); | protected |
| | Creates a BaseDate object with the current system date. | |
| constructor | BaseDate(unsigned char M, unsigned char D, unsigned Y); | protected |
| | Creates a BaseDate object with the given month, day, and year. | |
| constructor | BaseDate(const BaseDate& BD); | protected |
| | Copy constructor. | |
| Day | unsigned Day() const; | |
| | Returns the day of the month. | |
| hashValue | virtual hashValueType hashValue() const; | |
| | Returns the hash value of the date object. See HashTable::hashValue for more details. | |
| isA | virtual classType isA() const = 0; | |
| | A pure virtual function to return a classtype ID (to be defined in derived classes). | |
| isEqual | virtual int isEqual(const Object& testDate) const; | |
| | Returns 1 if the object represents the same date as <i>testDate</i> . Otherwise returns 0. | |
| isLessThan | virtual int isLessThan(const Object& testDate) const; | |
| | Returns 1 if the object precedes <i>testDate</i> on the calendar. | |

Month	unsigned Month() const; Returns the month.
nameOf	virtual char *nameOf() const = 0; Pure virtual function to be defined by derived classes to return their object ID string.
printOn	virtual void printOn(ostream& outputStream) const = 0; Pure virtual function to be defined in derived classes to print the date object on the given stream. printOn is for internal use by the overloaded operator <<.
<i>operator << is a friend of Object. See page 281.</i>	
SetDay	void SetDay(unsigned char D); Sets the day to <i>D</i> .
SetMonth	void SetMonth(unsigned char M); Sets the month to <i>M</i> .
SetYear	void SetYear(unsigned Y); Sets the year to <i>Y</i> .
Year	unsigned Year() const; Returns the year.

BaseTime

ltime.h



BaseTime is an abstract class derived from **Sortable** that provides basic time manipulation functions.

Member functions

constructor	BaseTime();	protected
	Creates a BaseTime object with the current system time.	
constructor	BaseTime(const BaseTime& BT);	protected
	Copy constructor.	

BaseTime

- constructor** BaseTime(unsigned char H, unsigned char M = 0, unsigned char S = 0, unsigned char HD = 0); **protected**
Creates a **BaseTime** object with the given hour, minutes, seconds, and hundredths of seconds.
- hashValue** virtual hashValueType hashValue() const;
Returns the hash value of the **BaseTime** object. See **HashTable::hashValue** for more details.
- hour** unsigned hour() const;
Returns the hour.
- hundredths** unsigned hundredths() const;
Returns the hundredths of a second.
- isA** virtual classType isA() const = 0;
Pure virtual function for a derived class to return its class ID.
- isEqual** virtual int isEqual(const Object& testTime) const;
Returns 1 if this object equals *testTime*; otherwise returns 0.
- isLessThan** virtual int isLessThan(const Object& testTime) const;
Returns 1 if this object is less than *testTime*; otherwise returns 0.
- minute** unsigned minute() const;
Returns the minute.
- nameOf** virtual char *nameOf() const = 0;
Pure virtual function to be defined by derived classes to return their object ID string.
- printOn** virtual void printOn(ostream& outStream) const = 0;
operator << is a friend of Object. See page 281. Pure virtual function to be defined in derived classes to print the time object on the given stream. **printOn** is for internal use by the overloaded operator <<.
- second** unsigned second() const;
Returns the seconds.
- setHour** void setHour(unsigned char H);
Sets the hour to *H*.

setHundredths void setHundredths(unsigned char HD);

Sets the hundredths of a second to *HD*.

setMinute void setMinute(unsigned char M);

Sets the minutes.

setSecond void setSecond(unsigned char S);

Sets the seconds.

Btree

btree.h



The class **Btree**, derived from **Collection**, implements the B-tree, a popular data structure offering efficient storage and retrieval with large, dynamic volumes of data. (A detailed account of Borland C++ development of B-tree theory is beyond the scope of this manual: see BTREE.CPP and D. E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.3.). **Btree** makes use of several auxiliary, noncontainer friend classes: **Node**, **Item**, **InnerNode**, and **LeafNode** (the last two being derived from **Node**). You can study these in btree.h. Here, we will just outline the members of the **Btree** class, which should suffice for most applications.

Member functions

add void add(Object&);

Add the given object to the B-tree.

constructor Btree(int ordern = 3);

Creates a B-tree of order *ordern* (default order is 3).

decrNofKeys void decrNofKeys();

protected

Decrements the itemsInContainer data member

detach void detach(Object& toDetach, DeleteType dt = NoDelete);

Removes the given object from the B-tree. The fate of the removed object depends on the argument *dt*. See **TShouldDelete** for details.

findMember virtual Object& findMember(const Object& toFind) const;

Returns the given object if found, otherwise returns NOOBJECT.

flush void flush(DeleteType dt = DefDelete);

Flushes (empties) the B-tree. The fate of the removed objects depends on the argument *dt*. See **TShouldDelete** for details.

hasMember virtual int hasMember(const Object& obj) const;

Returns 1 if the given object is found in the B-tree, otherwise returns 0.

hashValue virtual hashValueType hashValue() const;

Returns the hash value of this B-tree. See **HashTable::hashValue** for more details.

i_add long i_add(const Object& obj); **protected**

Adds the given object to the tree and returns the index in the tree at which the object was inserted.

incrNofKeys void incrNofKeys(); **protected**

Increments the itemsInContainer data member

initerator virtual ContainerIterator& initIterator() const;

Creates an iterator for this B-tree.

See also: **Container::initerator**

isA virtual classType isA() const;

Returns *btreeClass*, the **Btree** class ID

isEqual virtual int isEqual(const Object& testObject) const;

Returns 1 if testObject is the same as this object.

nameOf virtual char *nameOf() const;

Returns "Btree", the **Btree** class ID string

operator () Object& operator[](long i) const;

Returns the root at index i

order int order();

Returns the order of the B-tree.

printOn virtual void printOn(ostream& outputStream) const;

operator << is a friend of Object. See page 281.

Sends the formatted B-tree data to the given output stream. **printOn** is for internal use by the overloaded operator <<.

rank `long rank(const Object& obj) const;`
 Returns the rank of the given object in the B-tree.

Friends

Node, **InnerNode**, and **LeafNode** are friends of **Btree**.

BtreeIterator

btree.h



The class **BtreeIterator** is derived from **ContainerIterator**. Its members follow the same scheme as those for the other container iterators.

Member functions

constructor `BtreeIterator(const Btree& toIterate);`

See **ContainerIterator** constructor

current `virtual Object& current();`

See **ContainerIterator::current**

operator ++ `virtual Object& operator ++();`
`virtual Object& operator ++(int);`

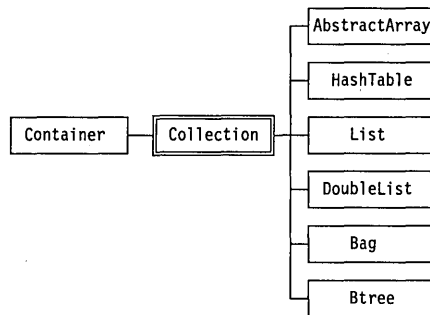
See **ContainerIterator::operator ++**

operator int `virtual operator int();`

Conversion operator to test for end of iterator position.

restart `virtual void restart();`

See **ContainerIterator::restart**



Collection is an abstract class derived from the abstract class **Container**. This means that although **Collection** is more specialized than **Container**, it still cannot be used directly for creating useful objects but exists only as a further stepping stone towards usable, derived instance classes.

Collection inherits five pure virtual functions (**flush**, **initliterator**, **isA**, **nameOf** and **getItemsInContainer**), that simply await definitions down the road by derived instance classes.

Collection extends the functionality of **Container** in several areas by adding both virtual and pure virtual member functions. The extra pure virtual functions are **add** and **detach**. Instance classes ultimately derived from **Collection**, therefore, will need to provide appropriate member functions for adding and removing elements.

The other (non-pure) virtual member functions added by **Collection** are **destroy**, **hasMember**, and **findMember**. The last two provide the key difference between **Collection** and **Container**. A **Collection**-derived object can determine if any given object is a member (with **hasMember**) and, by using an iterator, can locate a member object within the collection (with **findMember**).

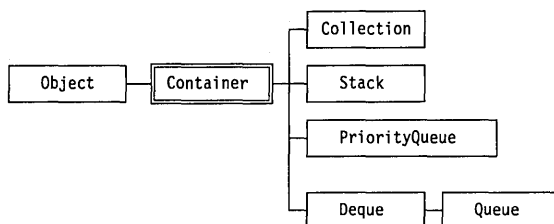
The offspring of **Collection** refine these access methods in various ways, and add other functions. In most applications, you will be dealing directly with a particular derived class of **Collection**, chosen to match your needs: sorted and unsorted arrays, hash tables, bags, sets, dictionaries, and single and double lists. However, it is useful to have a feel for how these instance classes build up from abstract classes, and why it is useful to have intermediate abstract classes.

Member functions

- add** `virtual void add(Object& o) = 0;`
 Pure virtual function to be defined in derived classes to add an object to a collection.
- constructor** Uses the **Container** base constructor.
- destroy** `void destroy(const Object& o);`
 Removes an object from a **Collection**. Whether the object itself is destroyed or not depends on the ownership status of the collection. If the collection currently *owns* the object, the object will be destroyed, otherwise the object survives. **destroy** is implemented with `detach(o, DefDelete);`
 See also: **TShouldDelete::ownsElements**
- detach** `virtual void detach(Object& o, DeleteType dt = NoDelete) = 0;`
 Pure virtual function to be defined in derived classes to remove an object from a collection. The destruction of the object depends both on the ownership status and the value (*Delete*, *NoDelete*, or *DefDelete*) passed via the *dt* argument.
 See also: **destroy, TShouldDelete::ownsElements**
- findMember** `virtual Object& findMember(const Object& testObject) const;`
 Returns the test object if it is in the collection, otherwise returns **NOOBJECT**.
- hasMember** `virtual int hasMember(const Object& o) const;`
 Returns **1** if the collection contains the given object.

Container

contain.h



Container

The abstract class **Container**, derived directly from **Object**, is the base for all the container classes. **Container** has a second pure virtual base class (not shown) called **TShouldDelete**. **Container** provides the following functionality:

1. A container can store objects of other classes, known as elements or items. (The objects in a container are sometimes called “members” of the container, but this usage can lead to ambiguities in C++.) A container can *flush* itself by removing all its elements.
2. A container can determine the number of objects it holds. Empty containers are allowed.
3. Container is also derived from **TShouldDelete** (multiple inheritance), which lets you control the *ownership* of a container’s elements. By default, a container owns its elements, meaning that it will destroy them when its destructor is called or when it is flushed.
4. A container can create external *iterators*, objects of type **ContainerIterator**, which can be used to traverse the container, element by element. With external iterators, you need to handle the scanning of the elements yourself. Other iterators, known as internal iterators, are generated automatically by certain member functions. These do their own loop tests and can perform arbitrary actions on each element (**forEach**). Member functions are also available for scanning the container until a certain condition is satisfied (**firstThat**, **lastThat**).
5. A container can test if it is equal to another container.
6. A container can display its elements on streams in a formatted way. A **printOn** function is provided from which the usual overloaded **<<** output operator can be obtained.

Strictly speaking, some of the above member functions are pure virtual functions that need to be defined in derived classes. See **Collection** class for a more detailed discussion.

Specialized containers are derived to two ways: directly derived are the classes **Stack**, **PriorityQueue**, and **Deque** (from which **Queue** is derived). Derived indirectly via another abstract class, **Collection**, are **AbstractArray**, **HashTable**, **Bag**, **Btree**, **List**, and **DoubleList**.

```
itemsInContainer countType itemsInContainer; protected  
Holds the current number of elements in the container.  
See also: getItemsInContainer
```

Member functions

constructor

```
Container();
```

Creates an empty container.

firstThat

```
virtual Object& firstThat( condFuncType testFuncPtr, void *paramList )
const;
```

Returns a reference to the first object in the container that satisfies a given condition. You supply a *testFuncPtr* that returns true for a certain condition. You can pass arbitrary arguments via the *paramList* argument. NOOBJECT is returned if no object in the container meets the condition. Note that you are not involved directly with iterators: **firstThat** and **lastThat** create their own internal iterators, so you can simply treat them as “search” functions.

See also: **lastThat**, **Object::firstThat**

flush

```
virtual void flush( DeleteType dt = DefDelete ) = 0;
```

A pure virtual function to be defined in derived classes. Flushing means removing all the elements from the container without destroying it. The value of *dt* determines whether the elements themselves are destroyed. By default, the ownership status of the container determines their fate. You can also set *dt* to *Delete* and *NoDelete*.

See also: **TShouldDelete::ownsElements**

forEach

```
virtual void forEach( iterFuncType actionFuncPtr, void *args );
```

forEach creates an internal iterator to execute the given action function for each element in the container. The *args* argument lets you pass arbitrary data to the action function.

getItemsInContainer

```
virtual countType getItemsInContainer() const = 0;
```

Pure virtual function to be defined by derived classes to return the number of elements in a container.

hashValue

```
virtual hashValueType hashValue() const = 0;
```

A pure virtual function to be defined by derived classes to return the hash value of an object. See **HashTable::hashValue** for more details.

initIterator

```
virtual ContainerIterator& initIterator() const = 0;
```

Pure virtual function to be defined in derived classes to initialize an external container iterator.

isA virtual classType isA() const = 0;

Pure virtual function to be defined in derived classes to return their class ID.

isEmpty virtual int isEmpty() const = 0;

Pure virtual function to be defined in derived classes. Returns 1 if a container has no elements; otherwise returns 0.

isEqual virtual int isEqual(const Object& testObject) const;

Returns 1 if the *testObject* is a container of the same type and size as this container, and with the same objects in the same order. Otherwise returns 0.

lastThat virtual Object& lastThat(condFuncType testFuncPtr, void *paramList) const;

Returns a reference to the last object in the container that satisfies a given condition. You supply a *testFuncPtr* that returns true for a certain condition. You can pass arbitrary arguments via the *paramList* argument. NOOBJECT is returned if no object in the container meets the condition. Note that you are not involved directly with iterators: **firstThat** and **lastThat** create their own internal iterators, so you can simply treat them as “search” functions.

See also: **firstThat, Object::firstThat**

nameOf virtual char *nameOf() const = 0;

Pure virtual function to be defined by derived classes to return their object type ID string (usually the unique class name).

printHeader virtual void printHeader(ostream& outputStream) const;

Sends a standard header for containers to the output stream (called by **printOn**).

See also: **printOn, printSeparator, printTrailer**

printOn virtual void printOn(ostream& outputStream) const;

operator << is a friend of Object. See page 281.

Sends a formatted representation of the container to the given output stream. **printOn** is for internal use by the overloaded operator <<.

See also: **printHeader, printSeparator, printTrailer**

printSeparator virtual void printSeparator(ostream& outputStream) const;

Sends to the output stream a separator (comma) between elements in a container (called by **printOn**).

See also: **printOn**, **printHeader**, **printTrailer**

printTrailer virtual void printTrailer(ostream& outputStream) const;

Sends to the output stream a standard trailer (a closing brace) for a container (called by **printOn**).

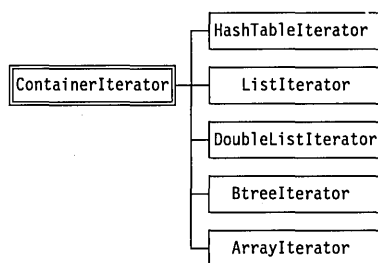
See also: **printOn**, **printHeader**, **printSeparator**

Friends

ContainerIterator is a friend of **Container**.

ContainerIterator

contain.h



ContainerIterator is an abstract class declared as a friend of **Container**. Container classes have **initIterator** member functions that create **ContainerIterator**-derived objects. These provide the basic mechanisms for traversing the elements in a container: incrementing through the container; returning positional information; testing for conditions, and so on. The member functions for **ContainerIterator** are all pure virtual and are defined in derived classes. See page 222 for more on the **ContainerIterator** hierarchy.

Member functions

current virtual Object& current() = 0;

Pure virtual function to be defined in derived classes to return the current element. If the current element is empty or invalid, **NOOBJECT** is returned.

ContainerIterator

operator int virtual operator int() = 0;

Pure virtual function to be defined by derived classes to provide a conversion operator to test for end of iteration condition.

operator ++ virtual Object& operator ++() = 0;
virtual Object& operator ++(int) = 0;

Advances the iterator one position in the container. The first version returns the object referred to *before* incrementing; the second version returns the object referred to *after* incrementing. The **int** argument is a dummy used to distinguish the two operators (see the section on Operator Overloading in the Programmer's Guide).

restart virtual void restart() = 0;

Pure virtual function to be refined in derived classes to set the current index of the iterator to the first nonempty element in the container.

Date

ldate.h



The **Date** instance class is a direct descendant of the abstract class **BaseDate**, defining a **printOn** function. You can vary **Date** for different national conventions without disturbing **BaseDate**.

Member functions

constructor Date();

Calls the **BaseDate** constructor to create a date object with today's date.

constructor Date(unsigned char M, unsigned char D, unsigned Y);

Calls the **BaseDate** constructor to create a date object with the given date.

constructor Date(const Date& aDate);

Copy constructor.

isA virtual classType isA() const;

Returns *dateClass*, the **Date** class ID.

nameOf virtual char *nameOf() const;

Returns "Date", the **Date** class ID string.

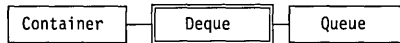
printOn virtual void printOn(ostream& outputStream) const;

operator << is a friend of Object. See page 281.

Sends a formatted date to the given output stream. The format is full month name, day, year, for example January 1, 1990. **printOn** is really for internal use by the overloaded operator <<.

Deque

deque.h



The instance class **Deque** (pronounced "deck"), derived from **Container**, implements a double-ended queue so it is one of the sequence classes. Objects can be examined, inserted, and removed at both the left and the right ends but nowhere else. You can use the member functions **peekLeft** and **peekRight** to examine the objects currently at the left and the right ends. **putLeft** and **putRight** insert objects at the ends. The **getLeft** and **getRight** members also access the end objects but detach them from the deque. The fate of the objects removed from the deque is determined by the same ownership and *DeleteType* considerations discussed in the **TShouldDelete** class (recall that **TShouldDelete** is a virtual base class for **Container**). **Deque** also acts as the base class for **Queue**.

Example

Source

```

#include <deque.h>
#include <string.h>

main()
{
    Deque d;
    String *s1 = new String("one");
    String *s2 = new String("two");
    String *s3 = new String("three");
    String *s4 = new String("four");

    // Populate the deque
    d.putLeft(*s1);
    d.putRight(*s2);
    d.putLeft(*s3);
    d.putRight(*s4);

    // Print to cout
    cout << "As a container:\n" << d << endl;
}
  
```

Deque

```
// Empty to cout
cout << "As a Deque:\n";
while (!d.isEmpty())
{
    cout << d.getLeft() << endl;
}

// Should be empty
cout << "\nShould be empty:\n" << d;
}
```

Output

```
As a container:
Deque { three,
      one,
      two,
      four }

As a Deque:
three
one
two
four

Should be empty:
Deque { }
```

Member functions

flush virtual void flush(DeleteType dt = DefDefault);

Flushes (empties) the deque without destroying it. The fate of any objects thus removed depends on the current ownership status and the value of the *dt* argument.

See also: **TShouldDelete::ownsElements**

getItemsInContainer virtual countType getItemsInContainer() const;

Returns the number of items in the deque.

getLeft Object& getLeft();

Returns the object at the left end and removes it from the deque. Returns NOOBJECT if the deque is empty.

See also: **TShouldDelete** class

getRight Object& getRight();

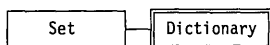
As for **getLeft**, except that the right end of the deque is returned.

See also: **getLeft**

- initIterator** virtual ContainerIterator& initIterator() const;
Initializes an iterator for the deque.
See also: **Container::initIterator**
- isA** virtual classType isA() const;
Returns *dequeClass*, the **Deque** class ID.
- isEmpty** virtual int isEmpty() const;
Returns 1 if a container has no elements; otherwise returns 0.
- nameOf** virtual char *nameOf() const;
Returns “Deque”, the **Deque** class ID string.
- peekLeft** Object& peekLeft() const;
Returns the object at the left end (head) of the deque. The object stays in the deque.
- peekRight** Object& peekRight();
Returns the object at the right end (tail) of the deque. The object stays in the deque.
- putLeft** void putLeft(Object& obj);
Adds (pushes) the given object at the left end (head) of the deque.
- putRight** void putRight(Object& obj)
Adds (pushes) the given object at the right end (tail) of the deque.

Dictionary

dict.h



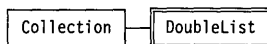
A dictionary is a special collection of **Association** type objects. The instance class **Dictionary** is derived from **Collection** via **Bag** and **Set**, implying that no duplicate association objects are allowed in a dictionary. **Dictionary** overrides the **add** function and adds a **lookup** function to the members inherited from **Set**. **lookup** allows you to retrieve the value object of an association stored in the dictionary if you supply the key.

 Member functions

- add** virtual void add(Object& assoc);
 Adds the given association (*assoc*) to the dictionary. If the given argument is not of type **Association**, a runtime error occurs.
- constructor** Dictionary(unsigned sz = DEFAULT_HASH_TABLE_SIZE);
 Invokes the base **Set** constructor to create an empty dictionary of size *sz*.
- isA** virtual classType isA() const;
 Returns *dictionaryClass*, the **Dictionary** class ID.
- lookup** Association& lookup(const Object& toLookUp) const;
 Returns the association matching the *toLookUp* key. If no match is found, **NOOBJECT** is returned.
- nameOf** virtual char *nameOf() const;
 Returns "Dictionary", the **Dictionary** class ID string.

 DoubleList

dbllist.h



The instance class **DoubleList**, derived from **Collection**, implements the classical doubly-linked list data structure (see D. E Knuth's *The Art of Computer Programming*, Volume 1, 2.2.5). Briefly, each node object of a doubly-linked list has two links, one pointing to the next node and one pointing to the previous node. The extreme nodes are called the *head* and the *tail*. As with the **Deque** class, you can examine, add, and remove objects at either end of the list.

 Member functions

- add** virtual void add(Object& toAdd);
 Add the given object at the beginning of the list.
- addAtHead** void addAtHead(Object& toAdd);
 Adds the given object at the beginning (head) of the list.

- addAtTail** void addAtTail(Object& toAdd);
 Adds the given object at the end (tail) the list.
- constructor** DoubleList();
 Creates a new, empty doubly-linked list.
- destroyFromHead** void destroyFromHead(const Object& toDestroy);
 Detaches the first occurrence of the given object encountered by searching from the beginning of the list. The object is destroyed only if it is owned by the list.
- destroyFromTail** void destroyFromTail(const Object& toDestroy);
 Detaches the first occurrence of the given object encountered by searching from the tail of the list towards the head. The object is destroyed only if it is owned by the list.
- detach** virtual void detach(Object& toDetach, DeleteType dt = NoDelete);
 Calls detachFromHead(toDetach, dt);
- detachFromHead** void detachFromHead(const Object& toDetach, DeleteType dt = NoDelete);
 Removes the first occurrence of the given object encountered by searching from the beginning of the list. The *dt* argument determines if the detached object is itself destroyed. See **TShouldDelete** for details.
- detachFromTail** void detachFromTail(const Object& toDetach, DeleteType dt = NoDelete);
 Removes the first occurrence of the object starting at the tail of the list and scanning towards the head. The *dt* argument determines if the detached object is itself destroyed. See **TShouldDelete** for details.
- flush** virtual void flush(DeleteType dt = DefDelete);
 Flushes (empties) the list without destroying it. The fate of the objects thus removed is determined by the *dt* argument as explained at **TShouldDelete**. The default value of *dt* means that the removed objects will be destroyed only if the list owns these objects.
 See also: **TShouldDelete::ownsElements**
- initIterator** virtual ContainerIterator& initIterator() const;
 Creates and returns a forward (from head to tail) iterator for the list.
- isA** virtual classType isA() const;
 Returns *doubleListClass*, the **DoubleList** class ID.

DoubleList

- nameOf** virtual char *nameOf() const;
Returns "DoubleList", the **DoubleList** class ID string.
- peekAtHead** Object& peekAtHead() const;
Returns the object at the head of the list (without removing it).
- peekAtTail** Object& peekAtTail() const;
Returns the object at the tail of the list (without removing it).

Friends

DoubleListIterator is a friend of **DoubleList**

DoubleListIterator

dbllist.h



DoubleListIterator, derived from **ContainerIterator**, implements the special iterators for traversing doubly-linked lists in either direction. This class adds overloading of the pre- and postdecrement operator `--` to allow reverse iteration. For more details on iterators, see **ContainerIterator**, and **DoubleList::initIterator**.

Member functions

- constructor** DoubleListIterator(const DoubleList& toIterate, int atHead = 1);
Creates an iterator for the given list. The iterator will begin at the head of the list if *atHead* is 1, otherwise it starts at the tail.
- current** virtual Object& current();
Returns the object at the current index of the iterator. If the current index exceeds the upper bound, NOOBJECT is returned.
- operator ++** virtual Object& operator ++ (int);
virtual Object& operator ++ ();
See **ContainerIterator operator ++**
- operator --** Object& operator -- (int);
Object& operator -- ();

Moves the iterator back one position in the list. The object returned is either the current object (postdecrement) or the object at the new position (predecrement), or NOOBJECT if no valid object at the relevant position. The first version gives postdecrement, the second gives predecrement. The **int** argument is a dummy serving only to distinguish the two operators.

operator int virtual operator int();

Conversion operator to test for the end of an iteration condition.

restart virtual void restart();

Moves the iterator back to its starting position at the head of the list.

See also: **DoubleListIterator** constructor

Error

object.h



The class **Error** is a special instance class derived from **Object**. There is just one instance of class **Error**, namely **theErrorObject**. Pointing to this global object is the static object pointer *Object::ZERO*. NOOBJECT is defined as **(Object::ZERO)* in object.h. The operator **Object::operator new** returns a pointer to **theErrorObject** if an attempt to allocate an object fails. You may test the return value of the **new** operator against *Object::ZERO* to see whether the allocation failed.

NOOBJECT is rather like a null pointer, but serves the vital function of occupying empty slots in a container. For example, when an **Array** object is created (not to be confused with a traditional C array), each of its elements will initially contain NOOBJECT.

Member functions

delete void operator delete(void *);

Invokes a runtime error if an attempt to delete the **Error** object is detected.

isA virtual classtype isA() const;

Returns *errorClass*, the **Error** class ID.

isEqual virtual int isEqual(const Object& testObject const);

Returns 1 if the test object is the **Error** object.

nameOf virtual char *nameOf() const;

Returns the **Error** class ID string.

printOn virtual void printOn(ostream& outputStream) const;

operator << is a friend of Object. See page 281.

Prints the string "Error\n" on the given stream. **printOn** is for internal use by the overloaded operator <<.

HashTable

hashtbl.h



The instance class **HashTable** provides an implementation of an unordered collection in which objects are added and retrieved via a hashing function. A hash table provides a fixed array with *size* slots (usually a prime number), one for each possible hash value modulo *size*. A hashing function computes the hash value for each object (or a key part of that object) to be added, and this determines the slot to which the new object is assigned.

For each containable object of class **X**, the member function **X::HashValue** returns a value (of type *hashValueType*) between 0 and 65535, which is as "unique" as possible. This "raw" hash value is reduced modulo *size*. We'll use the term *hash value* to refer to this reduced value in the range 0 to *size* - 1. This hash value serves as an index into the hash table. The internal organization of the table is hidden, but it may help you to consider the slots as pointers to lists.

It should be clear that if you want to store more than *size* objects, the hash value cannot be unique for each object. So two cases arise when an object is added: if the slot is empty, a new list is assigned to the slot and the object is stored in the list; if the slot is already occupied by an object with the same hash value (known as a *collision*), the new object is stored in the existing list attached to the slot. When it comes to locating an object, the hashing function computes its hash value to access the appropriate slot. If the slot is empty, NOOBJECT is returned, otherwise a **List::findMember** call locates the object.

Choosing the best **HashValue** function and table *size* is a delicate compromise between avoiding too many collisions and taking up too much memory. (Other hashing techniques are available, but the modulo

prime method is the most common. For more on hash table theory, see D. E. Knuth's *The Art of Computer Programming*, Volume 3, 6.4.). Hashing is widely used by compilers to maintain symbol tables.

Member functions

- add** `virtual void add(Object& objectToAdd);`
 Adds the given object to the hash table.
- constructor** `HashTable(sizeType aPrime = DEFAULT_HASH_TABLE_SIZE);`
 Creates an empty table. The *aPrime* argument is a prime number used in the hashing function (the default is defined in resource.h).
- detach** `virtual void detach(Object& objectToDetach, DeleteType dt = NoDelete);`
 Removes the given object from the hash table. Whether the object itself is destroyed or not depends on the *dt* argument, as explained in **TShouldDelete::ownsElements**.
- findMember** `virtual Object& findMember(const Object& testObject) const;`
 Returns the target object if found, otherwise returns NOOBJECT.
- flush** `virtual void flush(DeleteType dt = DefDelete);`
 Removes all the elements from the table without destroying it. The value of *dt* determines whether the elements themselves are destroyed. By default (*dt* = *DefDelete*), the ownership status of the table determines the fate of all its elements, as explained in **TShouldDelete::ownsElements**. You can set *dt* to *Delete* to force destruction of the flushed elements regardless of ownership. If *dt* is set to *NoDelete*, the flushed elements will survive regardless of ownership.
 See also: **TShouldDelete::ownsElements**
- hashValue** `virtual hashValueType hashValue() const;`
 Returns the raw hash value of this table. This must not be confused with the hash values calculated *by* the hash table for each of the objects it stores. When an object *x* of class **X** is added or retrieved from a hash table *h*, the raw hash value used is *x.hashValue()*. The true hash value (usually modulo *size*) is obtained from the hash table object via *h.getHashValue(x)*. Only classes with a proper **hashValue** member function can provide objects for storage in a hash table. All standard **Object**-derived classes in the library have meaningful hashing functions provided. For example, **BaseDate::hashValue** (unless overridden) returns the value *YY + MM +*

HashTable

DD from which the (private) member function **HashTable::getHashValue** computes a hash value (using mod *size*). It is this value that governs the hash table's **add**, **findMember**, and **detach** operations.

initIterator virtual ContainerIterator& initIterator() const;
Creates and returns an iterator for the hash table. See **Container::initIterator** for more details.

isA virtual classType isA() const;
Returns *hashTableClass*, the **HashTable** class ID.

nameOf virtual char *nameOf() const;
Returns "HashTable", the **HashTable** class ID string.

Friends

HashTableIterator is a friend of **HashTable**

HashTableIterator

hashtbl.h



HashTableIterator is an instance class providing iterator functions for **HashTable** objects. Since hash values are stored in an array, hash table iterators use the array iterator mechanism. See **ContainerIterator** for a detailed discussion of iterators.

Member functions

- constructor** HashTableIterator(const Array& toIterate);
See **ContainerIterator** constructor
- current** virtual operator Object& current();
See **ContainerIterator::current**
- operator int** virtual operator int();
Conversion operator to test for end of iterator position.
- operator ++** virtual Object& operator ++ (int);
virtual Object& operator ++ ();

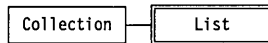
See **ContainerIterator::operator ++**

restart virtual void restart()

See **ContainerIterator::restart**

List

list.h



The instance class **List**, derived from **Collection**, implements a linear, linked list. Lists are unordered collections in which objects are linked in one direction only to form a chain. You can usually add objects only at the start of a list but any object can be removed from a list. You can traverse a list (from head to tail) with an iterator to access the objects sequentially. **List** has an internal private class **ListElement** providing memory management and other functions for the pairs of pointers (to object and to next element) that constitute the elements of a **List** object. (For more on list theory, see Sedgwick's *Algorithms* and Knuth's *The Art of Computer Programming*, Volume 1, 2.2).

Member functions

add void add(Object& toAdd);

Adds the given object at the head of the list. The added object becomes the new head.

constructor List();

Creates an empty list.

detach virtual void detach(Object& toDetach, DeleteType dt = NoDelete);

Removes the given object from the list. Whether the object itself is destroyed or not depends on the *dt* argument, as explained in **TShouldDelete::ownsElements**.

flush virtual void flush(DeleteType dt = DefDelete);

Removes all the objects from the list without destroying it. The value of *dt* determines whether the objects themselves are destroyed. By default (*dt* = *DefDelete*), the ownership status of the list determines the fate of its elements, as explained in **TShouldDelete::ownsElements**. You can set *dt* to *Delete* to force destruction of the flushed objects regardless of

ownership. If *dt* is set to *NoDelete*, the flushed objects will survive regardless of ownership.

See also: **TShouldDelete::ownsElements**

hashValue virtual hashValueType hashValue() const;

Returns the hash value of this list. See **HashTable::hashValue** for more details.

initIterator virtual ContainerIterator& initIterator() const;

See **Container::initIterator**

isA virtual classType isA() const;

Returns *listClass* the **List** class ID.

nameOf virtual char *nameOf() const;

Returns "List", the **List** class ID string.

peekHead Object& peekHead() const;

Returns the object at the head of the list.

Friends

ListIterator is a friend of **List** and **ListElement**.

ListIterator

list.h



ListIterator is an instance class derived from **ContainerIterator** providing iterator functions for **List** objects. See **ContainerIterator** for a discussion of iterators.

Member functions

constructor ListIterator(const List& toIterate);

Creates an iterator for the given list. The starting and current elements are set to the first element of the list. See **ContainerIterator** constructor for details.

current virtual Object& current();

See **ContainerIterator::current**

operator ++ virtual Object& operator ++ (int);
virtual Object& operator ++ ();

See **ContainerIterator::operator ++**

operator int virtual operator int();

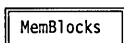
Conversion operator to test for end of iterator position.

restart virtual void restart();

See **ContainerIterator::restart**

MemBlocks

memmgr.h



The classes **MemBlocks** and **MemStack** in memmgr.h offer specialized memory management not only for the container classes but for other applications. Detailed knowledge of their operations is not needed for normal container applications. If you are planning your own advanced memory management schemes, you should first study memmgr.h and MEMMGR.CPP.

MemBlocks is a noncontainer, instance class, providing *fixed-block* memory allocations. Large, dynamic lists and trees need to allocate and free their node blocks as quickly as possible. **MemBlocks** offers more efficient memory management than the standard heap manager for this kind of operation. The **MemBlock** constructor takes two arguments: block size and number of blocks. These determine the size of the internal blocks that are allocated as needed using the normal run-time library allocation functions. A free list of blocks is maintained and the internal blocks are not released until the **MemBlock** object is destroyed. The following example illustrates the use of **MemBlocks** with a simplified **Node** class:

```
class Node
{
    Node *next;
    Object *obj;
    static MemBlocks memBlocks;
```


MemBlocks

```
void *operator new( size_t sz ) { return memBlocks.allocate ( sz); }
void operator delete( void * blk ) { memBlocks.free ( blk ); }
...
};
```

CAUTION: If you derive a class from a class that does its own memory management as in the **Node** example above, then *either* the derived class must be the same size as the base class *or* you must override the **new** and **delete** operators.

See also: **MemStack** class.

allocate void allocate(size_t sz, unsigned blks = 100);

Allocates *blks* blocks each of size *sz*

free void free(void * ptr);

Frees the memory blocks at *ptr*.

MemStack

memmgr.h

MemStack

MemStack is a noncontainer, instance class, providing fast mark-and-release style memory management. Although used internally by various container classes, **MemStack** is also available for general use. Memory allocations and deallocations are extremely fast since they “popped” and “pushed” on a stack of available blocks. Marking and releasing blocks is handled by objects of a helper marker class. When a marker is created it records the current location in the memory stack; when a marker is destroyed, the stack is returned to its original state, freeing any allocations made since the marker was created. For example:

```
MemStack symbols;

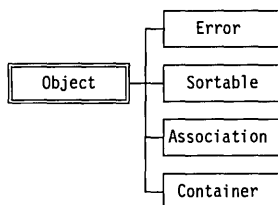
void handleLocals()
{
    Marker locals( symbols ); // marks current state of symbols
    Sym *symbol1 = new(symbols)Sym; // add a Sym to the table
    Sym *symbol2 = new(symbols)Sym; // and another
}
```

When the function exits, the **Marker** destructor releases the memory allocated by the `new(symbolS)` calls made in **handleLocal** and restores the memory stack.

See also: **MemBlocks**

Object

object.h



Object is an abstract class providing the primordial base for the whole **Object**-based container hierarchy (with the exception of the iterator classes). The member functions provide the basic essentials for all derived classes and the objects they contain. **Object** has four immediate children: **Error**, **Sortable**, **Association**, and **Container**.

Data member

ZERO `static Object *ZERO;`

A static pointer to the unique instance of class **Error**. *ZERO* is used to define **NOOBJECT**.

See also: **Error** class

Member functions

constructors

```
Object();
Object( Object& obj );
```

Creates or copies an object.

firstThat `virtual Object& firstThat(condFuncType testFuncPtr, void *paramList) const;`

Returns **this* if the object satisfies the condition specified by the **BOOLEAN** *testFunc* function, otherwise **NOOBJECT** is returned. You can

pass arbitrary arguments via the *paramList* argument. Note that **firstThat**, **lastThat**, and **forEach** work for *all* **Object**-derived objects, both container and non-container objects, whether they are in containers or not. With container objects, you can get iteration through the contained objects. When used with objects outside containers, the three functions act only on the calling object, so **firstThat** and **lastThat** are equivalent. *condFuncType* is defined in *clstypes.h* as

```
#typedef int ( *condFuncType )( const class Object&, void *);
```

firstThat calls (*testFuncPtr)(*this, paramList). If 1 is returned, **firstThat** returns (Object &) *this, otherwise NOOBJECT is returned.

See also: **Container::firstThat**

forEach virtual void forEach(iterFuncType actionFuncPtr, void *args);

forEach executes the given action function on *this. The *args* argument lets you pass arbitrary data to the action function.

See also: **firstThat**

hashValue virtual hashValueType hashValue() const = 0;

A pure virtual function to be defined by derived classes to return the hash value of an object. See **HashTable::hashValue** for more details.

isA virtual classType isA() const = 0;

Pure virtual function for derived classes to return a class ID.

isAssociation virtual int isAssociation() const;

Returns 1 if the calling object is part of an **Association** object, otherwise returns 0. Must be overridden in classes providing associations.

See also: **Association** class.

isEqual virtual int isEqual(const Object& testObject) const = 0;

Pure virtual function to be defined in derived classes to test for equality between *testObject* and the calling object (assumed to be of the same type). **isEqual** is really for internal use by the operator **==** which first applies **isA** to see if the compared objects are of the same type. If they are, **==** then uses **isEqual**.

See also: **operator ==**

isSortable virtual int isSortable() const;

Returns 1 if the calling object can be sorted; that is, if the class **Sortable** is an ancestor. Otherwise returns 0. **Object::isSortable** returns 0. Sortable classes must override **isSortable** to return true.

See also: **Sortable** class

lastThat virtual Object& lastThat(condFuncType testFuncPtr, void *paramList) const;

Returns **this* if the object satisfies the condition specified by the **BOOLEAN** *testFuncPtr* function, otherwise **NOOBJECT** is returned. You can pass arbitrary arguments via the *paramList* argument. Note that **firstThat**, **lastThat**, and **forEach** work for *all* **Object**-derived objects, both container and non-container objects, whether they are in containers or not. With container objects, you get iteration through the contained objects. When used with objects outside containers, the three functions act only on the calling object, so **firstThat** and **lastThat** are equivalent.

See also: **firstThat**, **Container::lastThat**

nameOf virtual char *nameOf() const = 0;

Pure virtual function to be defined by derived classes to return their object ID string.

new void *operator new(size_t size);

Overrides the C++ operator **new**. Allocates *size* bytes for an object. Returns **ZERO** if the allocation fails, otherwise returns a pointer to the new object.

printOn virtual void printOn(ostream& outputStream) const = 0;

Pure virtual function to be defined in derived classes to provide formatted output of objects on the given output stream. **printOn** is really for internal use by the overloaded operator **<<**.

See also: **operator <<**

ptrToRef static Object ptrToRef(Object *p);

Returns **ZERO* if *p* is 0, else returns **p*

Friends

operator << ostream& operator <<(ostream& outputStream, const Object& anObject);

Uses **printOn** to send a formatted representation of *anObject* to the given output stream. The stream is returned, allowing the usual chaining of the **<<** operator.

operator << is a friend of **Object**.

Related functions

The following overloaded operators are related to **Object** but are not member functions:

operator == `int operator ==(const Object& test1, const Object& test2);`

Returns 1 if the two objects are equal, otherwise returns 0. Equal means that **isA** and **isEqual** each return the same values for the two objects.

Note that for sortable objects (derived from the class **Sortable**) there are also overloaded nonmember operators **<**, **>**, **<=**, and **>=**.

See also: **Object::isA**, **Object::isEqual**, **operator !=**, **Sortable** class.

operator != `int operator !=(const Object& test1, const Object& test2);`

Returns 1 if the two objects are unequal, otherwise returns 0. Unequal means that either **isA** or **isEqual** each return the different values for the two objects.

See also: **Object::isA**, **Object::isEqual**, **operator ==**

PriorityQueue

priorityq.h



The instance class **Priority Queue**, derived from **Container**, implements the traditional priority queue data structure. The objects in a priority queue must be *sortable* (see **Sortable** class for details). A priority queue is either a GIFO (greatest-in-first-out) or SIFO (smallest-in-first-out) container widely used in scheduling algorithms. The difference really depends on your ordering definition. In explaining this implementation, we'll assume a GIFO. You can picture sortable objects being added at the right, but each extraction from the left gives the "greatest" object in the queue. (For applications where you need to extract the smallest item, you need to adjust your definition of "less than.") A detailed discussion of priority queues can be found in Knuth's *The Art of Computer Programming*, Volume 3, 5.2.3.

The member function **put** adds objects to the queue; **peekLeft** lets you examine the largest element in the queue; **get** removes and returns the largest element; you can also detach this item with **detachLeft** without

“getting” it. **PriorityQueue** is implemented internally using a private **Btree** object called *tree*.

Member functions

- detachLeft** void detachLeft(Container::DeleteType dt = Container::DefDelete);
Removes the smallest object from the priority queue. Whether this object is destroyed or not depends on the value of *dt* as explained in **TShouldDelete::ownsElements**.
- flush** void flush(Container::DeleteType dt = Container::DefDelete);
Flushes (empties) the priority queue. The fate of the removes objects depends on the value of *dt* as explained in **TShouldDelete::ownsElements**.
- get** Object& get();
Detaches the smallest object from the priority queue and returns it. The detached object is not itself destroyed.
- getItemsInContainer** countType getItemsInContainer() const ;
Returns the number of items in the priority queue.
- hashValue** virtual hashValueType hashValue() const;
Returns the hash value of the priority queue. See **HashTable::hashValue** for more details.
- hasMember** int hasMember(const Object& obj) const;
Returns 1 if *obj* belongs to the priority queue, otherwise returns 0.
- initIterator** virtual void ContainerIterator& initIterator() const;
Creates and returns an iterator for this queue.
See also: **ContainerIterator**
- isA** virtual classType isA() const;
Returns *priorityQueueClass*, the **PriorityQueue** type ID.
- isEmpty** int isEmpty();
Returns 1 if the priority queue is empty, otherwise returns 0.
- nameOf** virtual char *nameOf() const;
Returns “PriorityQueue”, the **PriorityQueue** type ID string.

peekLeft Object& peekLeft();

Returns the smallest object in the priority queue without removing it.

put void put(Object& o);

Add the given object to the priority queue.

Queue

queue.h



The instance class **Queue**, derived from **Deque**, implements the traditional queue data structure. A queue is a FIFO (first-in-first-out) container where objects are inserted at the left (head) and removed from the right (tail). For a detailed discussion of queues, see Knuth's *The Art of Computer Programming*, Volume 1, 2.2.1.

The member functions **put** and **get** insert and remove objects. **Queue** is implemented as a restricted-access version of **Deque**.

Example

Source

```

#include <queue.h>
#include <strng.h>
#include <assoc.h>

main()
{
    Queue q;
    String *s1 = new String("a string");
    String *s2 = new String("another string");
    Association *a1 = new Association(*s1,*s2);

    // Populate the queue
    q.put(*s1);
    q.put(*s2);
    q.put(*a1);

    // Print to cout as a Container
    cout << "As a container:\n" << q << endl;

    // Empty the queue to cout
    cout << "As a queue:\n";
    while (!q.isEmpty())
    {

```

```

        cout << q << endl;
    }
    cout << endl;

    // Queue should be empty
    cout << "Should be empty:\n" << q;
}

```

Output

```

As a container:
Queue { Association { a string, another a string }
,
    another string,
    a string }

As a queue:
a string
another string
Association { a string, another string }

Should be empty:
Queue { }

```

Member functions

get Object& get();

Removes the object from the end (tail) of the queue. By default the removed object will not be destroyed. If the queue is empty, `NOOBJECT` is returned. Otherwise the removed object is returned.

See also: **TShouldDelete** class

isA virtual classType isA() const;

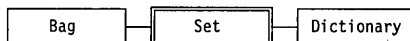
Returns *queueClass*, the **Queue** type ID.

put void put(Object& o);

Add an object to (the tail of) a queue.

Set

set.h



The instance class **Set** is a collection that allows only one instance of any object. This restriction calls for a specialized **add** member function to trap

any duplicates. Apart from this difference, the **Set** and **Bag** classes are essentially the same.

Member functions

add `virtual void add(Object& objectToAdd);`

Adds the given object to the set only if it is not already a member. If *objectToAdd* is found in the set, **add** does nothing.

See also: **Collection::hasMember**

constructor `Set(sizeType setSize = DEFAULT_SET_SIZE);`

Creates a set with the given size by calling the base **Bag** constructor.

See also: **Bag::Bag**

isA `virtual classType isA() const;`

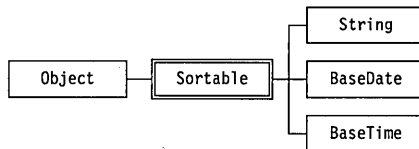
Returns *setClass*, the **Set** class ID.

nameOf `virtual char *nameOf() const;`

Returns "Set", the **Set** class ID string.

Sortable

sortable.h



Sortable is an abstract class derived from **Object**. You can use it to build classes of sortable objects. Objects are said to be sortable when they can be placed in an order based on some useful and consistent definition of "less than", "equal", and "greater than." Any two of these conditions will suffice, in fact, since the remaining condition can be constructed with logical operators. **Sortable** uses the two primitives "less than" and "equal" via the pure virtual functions (pure virtual functions) **isLessThan** and **isEqual**. Both of these member functions are applicable only to objects of the same type (see operators **==** and **<** for more details). The **isEqual** member function is a pure virtual function inherited from **Object** (since unordered objects also need a test for equality), whereas **isLessThan** is a

new pure virtual function for **Sortable**. Your derived classes must define these two member functions to provide an appropriate *ordering* of their objects.

Once **isLessThan** and **isEqual** are defined, you can use the overloaded operators **==**, **!=**, **<**, **<=**, **>**, **>=** in the obvious way (see Related Functions section below). The **<** operator tests the objects' types first with **isA** and returns 0 if the objects are of different types. Then if the objects are of the same type, the **isLessThan** member is called, returning 0 or 1. If your application calls for the ordering of objects of different types, you would have to define your own comparison operators.

The elements stored in *ordered* containers must clearly be sortable. For example, when adding elements to a **SortedArray** object, the **add** member function must compare the "size" of the incoming object against that of the existing elements. Similarly, **Btree** objects make use of magnitude for storage and access methods. Note, however, that an *unordered* container can hold either unsortable or sortable objects.

The type of sortable objects available differs between the **Object**-based containers and the template-based containers. In the **Object**-based hierarchy you must use objects ultimately derived from **Sortable**, whereas the template containers let you store any object or predefined data type for which **==** and **<** is defined. If you want to store **ints** in an **Object**-based container, for example, you must invent a suitable class:

```
class Integer : public Sortable
{
    int data;
    ...
public:
    virtual char *nameOf() const { return "Integer"; }
    virtual classType isA() const { return integerClass; }
    virtual int isLessThan( const Object& i ) const
        { return data < ((Integer&)i).data; }
    ...
}
```

The **Object**-based container library already provides three useful instance classes derived from **Sortable**: **String**, **Date**, and **Time** with the natural ordering you would expect. Remember, though, that you are free to define your own orderings in derived classes to suit your application. You must make sure that your comparisons are logically consistent. For instance, **>** must be transitive: $A > B$ and $B > C$ must imply $A > C$.

Member functions

hashValue virtual valueType hashValue() const = 0;

A pure virtual function to be defined by derived classes to return the hash value of a sortable object. See **HashTable::hashValue** for more details.

isA virtual classType isA() const = 0;

Pure virtual function to be defined in derived classes to return their class ID.

isEqual virtual int isEqual(const Object& testObject) const = 0;

Pure virtual function to be defined in derived classes to test for equality. Equality means that the calling object is the same type as *testObject* and that their values (as defined by this member function) are equal. Returns 1 for equality, otherwise 0.

isLessThan virtual int isLessThan(const Object& testObject) const = 0;

Pure virtual function to be defined in derived classes to test for “less than.” Returns 1 for “less than”, otherwise 0.

isSortable virtual int isSortable() const;

Returns 1 for all objects derived from **Sortable** (overrides **Object::isSortable**).

nameOf virtual char *nameOf() const = 0;

Pure virtual function to be defined by derived classes to return their object ID string.

printOn virtual void printOn(ostream& outputStream) const = 0;

operator << is a friend of Object. See page 281.

Pure virtual function to be defined in derived classes to output the object on the given stream. **printOn** is for internal use by the overloaded operator <<.

Related functions

The following overloaded operators are related to **Sortable** but are not member functions:

operator < int operator <(const Sortable& test1, const Sortable& test2);

Returns 1 if the two objects are of the same type *X*, and *test1* is “less than” *test2* (as defined by **X::isLessThan**). Otherwise returns 0.

See also: **Sortable::isLessThan**, **Sortable::isA**

operator <= int operator <=(const Sortable& test1, const Sortable& test2);

As for **operator <**, but also tests true for equality.

operator > int operator >(const Sortable& test1, const Sortable& test2);

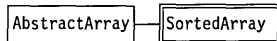
Returns 1 if the two objects are of the same type *X*, and *test1* is not “less than” and not “equal” to *test2* (as defined by **X::isLessThan** and **X::isEqual**). Otherwise returns 0.

operator >= int operator >=(const Sortable& test1, const Sortable& test2);

As for **operator >**, but also tests true for equality. Note that **>=** returns **!(test1<(test2))**, so it returns 1 if *test1* and *test2* are of different types.

SortedArray

sortarry.h



The instance class **SortedArray**, derived from **AbstractArray**, defines an array that maintains its elements in ascending order (according to the ordering defined for the elements). That is, the element at index *n* is less than or equal to the element at index *n* + 1. Note that the operator **<=**, used when adding new elements to the array, must be defined for comparing any objects in the array. This will be the case for objects ultimately derived from **Sortable** (see the Related Functions section of the **Sortable** class reference) as well as for the standard C integral types.

Array and **SortedArray** are identical in many areas (they have the same base, **AbstractArray**). One difference is that **SortedArray::detach** “squeezes” the array to maintain ascending order, while **Array::detach** leaves “holes” in the array.

Stack

stack.h



The instance class **Stack**, derived from **Container**, is one of the *sequence* classes like **Queue** and **Deque**. A stack is a LIFO (last-in-first-out) linear list for which all insertions (pushes) and removals (pops) take place at one end (the top or head) of the list (see D. E Knuth's *The Art of Computer Programming*, Volume 1, 2.2). In addition to the traditional **pop** and **push** member functions, **Stack** provides **top**, a member function for examining the object at the top of the stack without affecting the stack's contents. **top** must be used with care since it returns a reference to an object that may be owned by the stack. Destroying the object returned by **top** can disrupt the internal mechanism for storing the stack. The correct way to dispose of the top element is to use **pop** followed by **delete**. **Stack** is implemented internally as a **List** via a private data member *theStack* of type **List**.

See also: **Stacks** templates and classes

Example

Source

```
#include <stack.h>
#include <string.h>
#include <assoc.h>

main()
{
    Stack s;
    String *s1 = new String("a string");
    String *s2 = new String("another string");
    Association *a1 = new Association(*s1,*s2);

    s.push(*s1);
    s.push(*s2);
    s.push(*a1);

    // Print the Stack
    cout << "As a Container:\n" << s << endl;

    // Empty the stack to cout
    cout << "As a Stack:\n";
    while (!s.isEmpty())
    {
        Object& obj = s.pop();
        cout << obj << endl;
        delete &obj;
    }
}
```

Output

```
As a Container:
Stack { Association { a string, another string }
    another string,
```

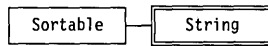
```

        a string )
As a Stack:
    Association { a string, another string }
    another string
    a string

```

Member functions

- flush** virtual void flush(DeleteType dt = DefDelete);
 Flushes (empties) the stack. The fate of the removed objects depends on the argument *dt*. See **TShouldDelete** for details.
- getItemsInContainer** virtual countType getItemsInContainer() const;
 Returns the number of items in the stack.
- initIterator** virtual ContainerIterator& initIterator() const;
 Creates and returns a stack iterator for the stack.
 See also: **ContainerIterator** class
- isA** virtual classType isA() const;
 Returns *stackClass*, the **Stack** type ID.
- isEmpty** virtual int isEmpty() const;
 Returns 1 if the stack is empty, otherwise returns 0.
- nameOf** virtual char *nameOf() const;
 Returns "Stack", the **Stack** type ID string.
- pop** Object& pop();
 Removes the object from the top of the stack and returns the object. The fate of the popped object is determined by ownership as explained in **TShouldDelete**.
- push** void push(Object& toPush);
 Adds (pushes) the object *toPush* to the top of the stack.
- top** Object& top();
 Returns but does not remove the object at the top of the stack.



String is an instance class, derived from **Sortable**, to implement null-terminated, character strings. **String** objects are ordered in the usual lexicographic way using **strcmp** from the standard C `string.h`. Note that the **String** class include file is spelled `strng.h`. See **Sortable** for a discussion on ordered classes.

Member functions

- constructor** `String(const char *aPtr = " ");`
 Constructs a **String** object from the given C string.
- constructor** `String(const String& sourceString);`
 Copy constructor.
- hashValue** `virtual hashValueType hashValue() const;`
 Returns the hash value of this string. See **HashTable::hashValue** for more details.
- isA** `virtual classType isA() const;`
 Returns *stringClass*, the **Stack** type ID.
- isEqual** `virtual int isEqual(const Object& testString) const;`
 Returns 1 if the calling string is equal to *testString*, otherwise returns 0. You can also use the overloaded operators **==** and **!=** as explained in the Related functions section of the **Object** class.
- isLessThan** `virtual int isLessThan(const Object& testString) const;`
 Returns 1 if the calling string lexically precedes *testString*, otherwise returns 0. You can also use the overloaded operators **<**, **<=**, **>**, and **>=**, as explained in the Related functions section of the **Storable** class.
- nameOf** `virtual char *nameOf() const;`
 Returns the **Stack** type ID string.
- printOn** `virtual void printOn(ostream& outputString) const;`

operator << is a friend of Object. See page 281.

Prints this string on the given stream. **printOn** is really for internal use by the overloaded operator <<.

operator = String& operator =(const String& sourceString);

Overloads the assignment operator for string objects.

operator char * operator const char *() const;

Returns a pointer to this string.

Example

Source

```
// File TSTRNG.CPP:    Test the String class
#include <string.h>
void identify(String&);
main()
{
    char s1[21], s2[21];

    cout << "Enter a string: ";           // Read a string
    cin >> s1;
    String str1(s1);
    identify(str1);

    cout << "Enter another string: ";    // Read another
    cin >> s2;
    String str2(s2);
    identify(str2);

    // Do some relational tests:
    cout << "Equal: " << str1.isEqual(str2) << endl
         << "Less than: " << str1.isLessThan(str2) << endl;

    // String assignment:
    str2 = str1;
    cout << "After assignment:\n" << "Equal: "
         << str1.isEqual(str2);
}

void identify(String& str)
{
    // Echo a String's value and type
    cout << "Value: " << str
         << ", Object type: " << str.nameOf() << endl << endl;
}
```

Output

```
Enter a string: hello
Value: hello, Object type: String
Enter another string: goodbye
```


Value: goodbye, Object type: String

Equal: 0

Less than: 0

After assignment:

Equal: 1



Time is a sortable instance class derived from **BaseTime**. **Time** adds a **printOn** member. You can override this in derived classes to cope with international formatting variations.

Member functions

constructor

`Time();`

Calls the **BaseTime** constructor to create a **Time** object with the current time.

See also: **BaseTime** constructor

constructor

`Time(const Time& T);`

Copy constructor.

constructor

`Time(unsigned char H, unsigned char M = 0, unsigned char S = 0, unsigned char D = 0);`

Creates a **Time** object with the given hour, minutes, seconds, and hundredths of seconds.

isA

`virtual classType isA() const;`

Returns *timeClass*, the **Time** class ID.

nameOf

`virtual char *nameOf() const;`

Returns "Time", the **Time** class ID string.

printOn

`virtual void printOn(ostream& outputStream) const;`

operator << is a friend of Object. See page 281.

Sends a formatted **Time** object to the given output stream. The default format is `hh:mm:ss:dd a/pm` with nonmilitary hours. **printOn** is for internal use by the overloaded operator `<<`.

Timer

Timer is an instance class implementing a stop watch. You can use **Timer** objects to time program execution by calling the member functions **start** and **stop** within your program, and then using **time** to return the elapsed time. The **reset** member function resets the elapsed time to zero. Successive starts and stops will accumulate elapsed time until a reset.

Member functions

constructor

```
Timer();
```

Creates a **Timer** object.

reset

```
void reset();
```

Clears the elapsed time accumulated from previous start/stop sequences.

resolution

```
static double resolution();
```

Determines the timer resolution for all timer objects. This value is hardware and OS dependent. For example:

```
if( elapsedTime < timer.resolution() )
    cout << "Measured time not meaningful." << endl;
```

start

```
void start();
```

Ignored if the timer is running, otherwise starts the timer. The elapsed times from any previous start/stop sequences are accumulated until **reset** is called.

status

```
int status();
```

Returns 1 if the timer is running, otherwise 0.

stop

```
void stop();
```

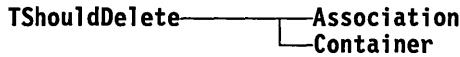
Stops the timer. The accumulated elapsed time is preserved until a **reset** call.

time

```
double time();
```

Returns the elapsed time. The precision is given by the value returned by the member function **resolution**.

Figure 6.3: Class hierarchies in CLASSLIB



TShouldDelete maintains the ownership state of a container. The fate of objects that are removed from a container can be made to depend on whether the container owns its elements or not. Similarly, when a container is destroyed, ownership can dictate the fate of contained objects that are still in scope. As a virtual base class for **Container** and **Association**, **TShouldDelete** provides ownership control for all containers and associations. The member function **ownsElements** can be used either to report or to change the ownership status of a container. **delObj** is used to determine if objects in containers or associations should be deleted or not.

Member functions

constructor

```
TShouldDelete( DeleteType dt = Delete );
```

Creates a **TShouldDelete** object. By default, containers and associations own their elements. *DeleteType* is an enumeration declared within the class as follows:

```
enum DeleteType { NoDelete, DefDelete, Delete };
```

ownsElements

```
int ownsElements();
void ownsElements( int del );
```

The first form returns 1 if the container owns its elements, otherwise it returns 0. The second form changes the ownership status as follows: if *del* is 0, ownership is turned off; otherwise ownership is turned on.

delObj

```
int delObj( DeleteType dt );
```

Tests the state of ownership and returns 1 if the contained objects should be deleted or 0 if the contained elements should not be deleted. The factors determining this are (i) the current ownership state and (ii) the value of *dt*, as shown in the following table.

delObj returns 1 if (*dt* is *Delete*) or (*dt* is *DefDelete* and the container currently owns its elements). Thus a *dt* of *NoDelete* returns 0 (don't delete) regardless of ownership; a *dt* of *Delete* return 1 (do delete) regardless of

ownership; and a *dt* of *DefDelete* returns 1 (do delete) if the elements are owned, but a 0 (don't delete) if the objects are not owned.

ownsElements	delObj	
	no	yes
NoDelete	no	no
DefDelete	no	yes
Delete	yes	yes

Converting from Microsoft C

If you're an experienced C or C++ programmer, but the Borland C++ programming environment is new to you, then you should read this appendix before you do anything else. We appreciate that you want to be up and running fast with a new piece of software, and we know that you want to spend as little time as possible reading the manual. However, the time you spend reading this chapter will probably save you a lot of time later. Please read on.

Environment and tools

The Borland C++ IDE (integrated development environment) is roughly the equivalent of the Programmer's Workbench, although naturally we think you'll find the IDE much easier to use. Chapter 3 in the *User's Guide* provides a complete reference to the IDE. If you're interested in building Windows applications, see Chapter 8 in the *Programmer's Guide*.

You can find out more about configuration and project files in Chapters 2 and 3 in the User's Guide.

The IDE loads its settings from two files: TCCONFIG.TC, the default configuration file, and a project file (.PRJ). TCCONFIG.TC contains general environmental information. The current project file contains information more specific to the application you're building.

A project is the IDE's equivalent of a makefile. It includes the list of files to be built, as well as settings for the IDE options that

control the compilation and linkage of that program. If you don't specify a project file when you start the IDE, a nameless project is opened and set with default compiler and linker options, but no file name list.

Unlike Microsoft C, however, Borland C++ does not automatically create and run a makefile based on settings and file names that you give it in the project. If you want to use the IDE to set up a project, but use MAKE to do the actual build, then you can use the PRJ2MAK utility to convert a project file to a makefile.

The following sections describe the significant differences between Borland C++'s MAKE, Project Manager, linker (TLINK), and command-line compiler (BCC) and Microsoft C's NMAKE, LINK, and CL.

Paths for .h and .LIB files

Microsoft C works with two environment variables, LIB and INCLUDE. The Microsoft linker uses the LIB variable to discover the location of the run-time libraries; similarly, INCLUDE is used to find standard header files. Borland C++ does not use environment variables to store the path for the library or include files. Instead, you can easily set these paths in the IDE using the environment options. If you are working with the command-line compiler or the linker, you can use command-line options or configuration files.

When you install Borland C++, you are asked to set paths for include files and library files. Those paths are then the default paths in the IDE. The include and library files paths are also written to the default command-line compiler configuration file TURBOC.CFG. The library path is written to the default stand-alone linker configuration file TLINK.CFG.

Remember that even if you haven't opened a project, Borland C++ will store the paths in its default project file.

In the IDE, reset default search paths for libraries and header files with the Options | Directories command. The settings in the Directories dialog box become a part of the current project.

For the command-line compiler, you can reset the search path for include and library files with the **-I** and **-L** options, respectively. These options can also be reset in the configuration file for the command-line compiler, TURBOC.CFG.

The linker can use the **/L** option to change search paths for libraries and initialization code (like COS.OBJ, the startup code for the small memory model). For instance, this option

```
/LC:\BORLANDC\LIB;C:\WINAPPS\LIB
```

tells the linker to look in the two paths named for library and initialization files.

You can also create a TLINK.CFG file. TLINK.CFG is a regular text file that contains a list of valid TLINK options.

*Borland licenses the
Resource Compiler from
Microsoft.*

For the Resource Compiler, the **-x** option tells it to ignore the INCLUDE variable. In addition, you can specify an additional search path with the **-i** option (**-i** all by itself does not imply **-x**).

When the Resource Compiler is invoked from the command line, it looks for windows.h on the path specified by the INCLUDE environment variable, if there is one. If that INCLUDE variable is set to some other path than the location of the windows.h supplied by Borland C++, your module might not be compiled correctly. (This does not occur in the IDE, because the IDE passes the correct information to the Resource Compiler.)

For instance, if you have been using Microsoft C, then you probably have an INCLUDE environment variable set to the path of the Microsoft C header files. If you have also been using the Microsoft Windows Software Development Kit, then the version of windows.h included with the SDK is probably also in the INCLUDE directory.

When you're building a Borland C++ application, the Resource Compiler should include the windows.h shipped with Borland C++. If you have a defined INCLUDE environment variable, then you should tell the Resource Compiler to ignore it with the **-x** option.

MAKE

New!

The version of MAKE supplied with Borland C++ 3.0 contains many new features, some of which are designed to increase compatibility with Microsoft's NMAKE. The new command-line switch **-N** turns on full NMAKE compatibility. See Chapter 2 in the *Tools and Utilities Guide* for more information on MAKE's options. The following list summarizes the differences between MAKE and NMAKE.

- NMAKE supports response files but MAKE doesn't.
- In NMAKE, you must surround strings to be compared with quotes. MAKE doesn't have this requirement; as long as the

string to be compared doesn't contain spaces, you can compare them without quotes.

- NMAKE predefines several implicit rules; MAKE doesn't. However, the BUILTINS.MAK file contains several implicit rules that you can use without specifying them in the makefile.

Command-line compiler

The following table lists comparable BCC and CL command-line compiler options. Some of the CPP (standalone preprocessor) options are listed. In many multi-pass compilers, a separate pass performs the work of the preprocessor, and the results of the pass can be examined. Since Borland C++ uses an integrated single-pass compiler, we provide the standalone utility CPP to supply the first-pass functionality found in other compilers.

Note that most CL options that take arguments allow for a space between the option and the argument. BCC options that take arguments are usually immediately followed by the argument or list.

Table 7.1: CL and BCC options compared

Microsoft C CL option	Borland C++ BCC option	What it does
N/A	@filename	Gives the command-line compiler a response file name.
N/A	+filename	Tell the command-line compiler to use the alternate configuration file <i>filename</i> .
N/A	-AK	Use only Kernighan and Ritchie keywords.
N/A	-AU	Use only UNIX keywords.
(See /Zpn)	-a	Align word.
(See /Zpn)	-a-	Align byte (default).
/Aw /Gw	-WD	Creates an .OBJ for Windows to be linked as a .DLL with all functions exportable.
/Aw /GW	-WDE	Creates an .OBJ for Windows to be linked as a .DLL with explicit export functions.
/Ax	-mx	Use memory model <i>x</i> . For BCC, following <i>t</i> , <i>s</i> , or <i>m</i> with ! tells compiler to assume DS != SS.
/Bn	N/A	Use alternate preprocessor <i>CnL</i> .
N/A	-B	Compile and call the assembler to process inline assembly code.
N/A	-b	Make enums word-sized by default.
N/A	-b-	Make enums signed or unsigned.
/C	-C	Nested comments on.
/c	-c	Compile to .OBJ but do not link.
/Did	-Dname	Define <i>name</i> to the string consisting of the null character.
/Did=value	-Dname=string	Defines <i>name</i> to <i>string</i> .
N/A	-d	Merge duplicate strings on.
N/A	-d-	Merge duplicate strings off (default).
N/A	-Efilename	Use <i>filename</i> as the assembler to use.

Table 7.1: CL and BCC options compared (continued)

/E	CPP -P	Preprocess source to standard output, include line numbers.
/EP	CPP -P-	Preprocess source to standard output, without line numbers.
N/A	-f-	Don't do floating point.
N/A	-ff	Fast floating point (default).
N/A	-ff-	Strict ANSI floating point.
N/A	-f87	Use 8087 hardware instructions.
N/A	-f287	Use 80287 hardware instructions.
/F hexnum		Sets stack size to <i>hexnum</i> bytes (hexnum must be hexadecimal).
(By default)	-Fc	Generates COMDEFs.
N/A	-Fm	Enables the -Fc , -Ff , and -Fs options.
(By default)	-Fs	Make DS == SS for all memory models.
/Fa [listfile]	-S	Create assembly listing. Name for list file defaults to <i>Source.ASM</i> .
/Fbbound-exe	N/A	Creates a bound executable file.
/Fc [listfile]	-s	Produces a combined source and assembly code listing. Name for list file defaults to <i>Source.COD</i> .
/Fe exefile	-eexefile	<i>exefile</i> names executable file.
/FI [listfile]	N/A	Creates object code list. Name for list file defaults to <i>Source.COD</i> .
/Fm [mapfile]	-M	Creates map file. Name defaults to <i>Source.MAP</i> , where source is the first source file specified.
/Fo objfile	-oobjfile	<i>objfile</i> names object file.
/FPa	N/A	Generate floating-point calls; select alternate math library.
/FPc	(default)	Emulate floating point (default for Borland C++); coprocessor used if present at run time).
/FPc87	N/A	Selects 80x87 library (80x87 coprocessor must be present at run time).
/FPi	-f	Inlines 80x87 instructions; selects emulator library (coprocessor used if present at run time).
/FPi87	-f87 or -f287	Inlines 80x87 instructions; chooses coprocessor library (coprocessor must be present at run time).
/Fr [browsefile]	N/A	Generates standard PWB Source Browser database.
/FR [browsefile]	N/A	Generates extended PWB Source Browser database.
/Fs [listfile]	N/A	Produce source list file. Source list file name defaults to <i>Source.LST</i> .
/Fx [xreffile]	N/A	<i>xreffile</i> specifies a name for the MASM cross-reference file.
G0	-1	Generate 80186 instructions.
G1	-1-	Generate 8088/8086 instructions.
G2	-2	Generate 80286 protected-mode compatible instructions.
/Gc	-p	Use Pascal calling convention. For CL, this is Pascal or FORTRAN, but currently same calling convention.
/Gd	-p-	Standard C calling conventions (default).
/Ge	-N	Check for stack overflow. (Default for CL, but not for BCC).
/Gi	N/A	Compile incrementally (for use with quick compile option /qc).
/Gm	N/A	Store strings in CONST segment.
/Gr	N/A	Enables _fastcall to call conventions for functions (if possible, passing value in registers).
/Gs	-N-	Turn off checking for stack overflow. (Off by default for BCC.)
/Gt [number]	-Ff[=size]	Creates far variables automatically; <i>size</i> or <i>number</i> is threshold.
/Gw	-W	Creates correct prolog/epilog for Windows program (for Borland C++, this creates an application with all functions exportable).
/GW	-WE	Generates prolog/epilog for explicit functions (marked with _export) in Windows program.
N/A	-H	Causes the compiler to generate and use precompiled headers.
N/A	-H-	Turns off generation and use of precompiled headers (default).

Table 7.1: CL and BCC options compared (continued)

N/A	-Hu	Tells the compiler to use but not generate precompiled headers.
N/A	-H=filename	Sets the name of the file for precompiled headers.
By default	-h	Use fast huge pointer arithmetic.
/H number	-inumber	Restricts length of external names to <i>number</i> .
/HELP	BCC	Calls QuickHelp. For help on BCC, simply invoke without options.
N/A	-in	Make significant identifier length to be <i>n</i> .
/I directory	-Ipath	Directories for include files. For CL, adds <i>directory</i> to the beginning of include file search directory list. See page 300.
N/A	-jn	Errors: Stop after <i>n</i> messages.
/J	-K	Changes default for char . from signed to unsigned. For Borland C++, -K- returns to signed.
N/A	-k	Standard stack frame on (default).
N/A	-Lpath	Directories for libraries.
/Lc and /Lr	/Td	Tells linker to create a real mode executable.
/Li [number]	N/A	Use incremental linker, instead of standard linker. <i>Number</i> specifies byte boundary for padding near functions.
/Lp	N/A	Create protected mode executable (OS/2).
/Lr		See /Lc .
/link options	-loptions	Pass <i>options</i> to linker when invoked.
N/A	-l-option	Suppress option <i>option</i> for the linker.
N/A	-M	Instruct the linker to create a map file.
/MAoption	-Toption	Pass to assembler when invoked.
/MD	N/A	Creates a DLL for OS/2.
/ML	N/A	Statically links a library to a DLL (OS/2).
/MT	N/A	Provides support for multithread programs for OS/2.
N/A	-npath	Set the output directory.
/NDdatasetg	-zRname	Sets the data segment name. For BCC, this option changes the name of the uninitialized data segment class to <i>name</i> . By default, the uninitialized data segments are assigned to class BSS.
/NMmodule	N/A	Sets the module name to <i>module</i> .
/nologo	N/A	Don't print sign-on banner.
/NTsegname	-zCname	Sets code segment name. This option changes the name of the code segment to <i>name</i> . By default, the code segment is named <code>_TEXT</code> , except for the medium, large and huge models, where the name is <code>filename_TEXT</code> . (<i>filename</i> here is the source file name.)
N/A	-O	Optimize jumps.
N/A	-O-	No optimization (default).
/Oa	-Oa	Assume optimistic aliasing.
/Oc	-Oc	Local common subexpressions.
/Od	-Od	Disable all optimizations.
/Oe	-Oe	Enabled global register allocation.
/Oi	-Oi	Generate inline intrinsic functions.
/Ol	-Ol	Enable loop optimizations.
/On	N/A	Disable unsafe optimizations.
/Op	-ff-	Strict ANSI floating point.
/Os	-Os	Optimize for size (default).
/Ot	-Ot	Optimize for speed.
/Ow	N/A	Somewhat optimistic aliasing.
/Ox	-Ox	Optimize for maximum speed.
N/A	-P	Perform a C++ compile regardless of source file extension.
N/A	-Pext	Perform a C++ compile and set the default extension to <i>ext</i> .

Table 7.1: CL and BCC options compared (continued)

N/A	-P-	Perform a C++ or C compile depending on source file extension (default).
N/A	-P-ext	Perform a C++ or C compile depending on extension; set default extension to <i>ext</i> .
N/A	-p-	Use C calling convention (default).
/P	CPP -ofilename	Preprocesses source file and sends output to <i>filename</i> (CPP), or to <i>Source.I</i> (CL).
N/A	-Qe	Instructs the compiler to use all available EMS memory (default).
N/A	-Qe-	Instructs the compiler to not use any EMS memory.
N/A	-Qx	Instructs the compiler to use all available extended memory.
N/A	-Qx=nnnn	Instructs the compiler to reserve <i>nnnn</i> Kb of extended memory for other programs, and to use the rest itself.
N/A	-Qx-	Instructs the compiler to not use any extended memory
N/A	-r	Use register variables on (default).
N/A	-r-	Suppresses the use of register variables.
N/A	-rd	Only allow declared register variables to be kept in registers.
/qc	N/A	Invokes quick compile (default for Borland C++).
/Sx option	N/A	Set options for source listing. Where <i>x</i> is l, p, s, or t.
N/A	-T-	Remove all previous assembler options.
/Ta asm_srcfile	N/A	Specifies that <i>asm_srcfile</i> be treated as an assembler source file.
/Tc c_srcfile	N/A	Specifies that <i>c_srcfile</i> be treated as a c source file.
N/A	-u	Generate underscores (default).
N/A	-u-	Disable underscores.
/u	N/A	Undefines all predefined identifiers.
/U Ident	-UIdent	Undefine any previous definitions of <i>Ident</i> .
N/A	-V	Smart C++ virtual tables.
N/A	-Vs	Local C++ virtual tables.
N/A	-V0, -V1	External and Public C++ virtual tables.
N/A	-Vf	Far C++ virtual tables.
N/A	-vi, -vi-	Controls expansion of inline functions.
/V string	N/A	Copies <i>string</i> to object file (for version control).
N/A	-w	Display warnings on.
N/A	-wxxx	Enable <i>xxx</i> warning message.
N/A	-w-xxx	Disable <i>xxx</i> warning message.
/w	-w-	Display warnings off.
N/A	-WS	Creates an .OBJ for Windows that uses smart callbacks.
/W n	(See -w)	Set warning level 0, 1, 2, 3, or 4.
/WX	-g1	Makes all warnings fatal. No object files are generated if warning occurs. (The -g option takes the form -gn , where <i>n</i> is the limit to number of warnings.)
N/A	-X	Disable compiler autodependency output.
/X	N/A	Ignore INCLUDE environment variable list of include search paths.
N/A	-Y	Enable overlay code generation.
N/A	-Yo	Overlay the compiled files.
N/A	-Z	Enable register usage optimization.
N/A	-zAname	Code class.
N/A	-zBname	BSS class.
N/A	-zDname	BSS segment.
N/A	-zEname	Far segment.
N/A	-zFname	Far class.
N/A	-zGname	BSS group.

Table 7.1: CL and BCC options compared (continued)

N/A	-zHname	Far group.
N/A	-zPname	Code group.
N/A	-zSname	Data group.
N/A	-zTname	Data class.
N/A	-zX*	Use default segment, class, or group name for X.
/Za	-A	Enforces ANSI compatibility. Use only ANSI keywords. No vendor-specific extension allowed.
/Zc	N/A	Ignores case for functions declared as _pascal .
/Zd	/y	Generates line numbers for symbolic debugger.
/Ze	-A-, -AT	Enable vendor-specific extensions.
/Zg	N/A	Generates function prototypes; writes to standard output.
/Zi	/V	For Microsoft, generates debugger information for CodeView. For Borland C++, generates information for IDE debugger and Turbo Debugger.
/ZI	N/A	Library search records not written to object file.
/Zpn	(See -a, -a-)	Packs structure members on the <i>n</i> byte boundary. <i>n</i> can be 1, 2, or 4.
/Zr	N/A	Generates checks for null pointers and far pointers that are out of range.
/Zs sourcefiles	N/A	Syntax check only.

Command-line options and libraries

The C0Fx.OBJ modules are provided for compatibility with source files intended for compilers from other vendors. The C0Fx.OBJ modules substitute for the C0x.OBJ modules; they are to be linked with DOS applications only, not Windows applications or DLLs. These initialization modules are written to alter the memory model such that the stack segment is inside the data segment. The appropriate C0Fx.OBJ module will be used automatically if you use either the **-Fs** or the **-Fm** command-line compiler option.

The **-Fc** (generate COMDEFs), **-Ff** (create far variables), **-Fs** (assume DS == SS in all models), and **-Fm** (enable all **-Fx** options) command-line compiler options are provided for compatibility. These options are documented in full in Chapter 5 in the *User's Guide*.

Linker

The Borland C++ linker, TLINK, is invoked automatically from the command-line compiler unless the **-c** compiler option is used. Options such as memory model and target (Windows or DOS), are passed from the compiler to TLINK; TLINK links the appropriate libraries based on the compile options.

TLINK can be used to build both DOS and Windows programs. See Chapter 4 in the *Tools and Utilities Guide* for material on module definition file statements.

The following table compares TLINK and LINK options. Note that Borland C++ TLINK options are case-sensitive, while Microsoft TLINK options are not.

Table 7.2: LINK and TLINK options compared

Microsoft C 6.0 Link option	Borland C++ TLINK option	What it does
N/A	/3	Enable 32-bit processing.
/A:size	/A=nnnn	Specify segment alignment for NewExe (Windows) images.
/BA	N/A	BATCH. Suppresses prompts for library or object files not found.
N/A	/C	Treat EXPORTS and IMPORTS section of module definition file as case sensitive.
/CO	/v	Include full symbolic debug information.
/CP:bytes	N/A	Sets the program's maximum memory allocation to <i>bytes</i> .
N/A	/d	Warn if duplicate symbols in libraries.
/DOSSEG	(See comment)	For assembly programs, forces a certain ordering of segments in executable. To enable DOSSEG for an assembly program, include DOSSEG in the source code.
/DS	N/A	For assembly programs, tells linker to load data starting at high end of DS instead of low end.
/E	N/A	Packs the executable by removing repeated series of bytes.
/F	By default	For LINK, tells linker to optimize far calls to procedures in same segment as caller. (Used with MS /PACKCODE option.) TLINK optimizes far calls automatically.
/HE	/?	Provides help on command-line options.
/HI	N/A	For real-mode assembly programs, places executable as high in memory as possible.
N/A	/i	Initialize all segments.
/INC	N/A	Prepares for ILINK.
/INF	N/A	Tells LINK to display link information while in process.
N/A	/Lpaths	Specify library search paths.
/LI	/l	Include source line numbers and associated addresses in map file.
/M	/m	Create map file with public global symbols.
/NOD[:filename]	/n	Don't use default libraries.
/NOE	/e	Ignore Extended Dictionary.
/NOF	N/A	Turns off far call translation (see LINK /F option).
/NOI	/c	Treat case as significant in symbols.
/NOL	N/A	Causes LINK to suppress banner (logo).
/NON	N/A	Arrange segments in executable in the same order as they are arranged by /DOSSEG.
/NOP	/P-	Turn off code packing.
N/A	/b	Overlay following modules or libraries. Microsoft LINK uses parentheses around files to be overlaid. (Note that the overlay scheme is different between products.)

Table 7.2: LINK and TLINK options compared (continued)

/O: <i>number</i>	N/A	Set interrupt <i>number</i> for passing control to overlays (other than the default 63).
/PACKC[:<i>number</i>]	/P=<i>n</i>	Pack code segments. <i>number</i> or <i>n</i> specifies maximum size of groups formed by /PACKC or /P.
/PACKD[:<i>number</i>]	N/A	Pack data segments. <i>number</i> specifies maximum size of groups formed by /PACKD.
/PADC:<i>padsiz</i>	N/A	Tells LINK to pad code module for ILINK.
/PADD:<i>padsiz</i>	N/A	Tells LINK to pad data segments by <i>padsiz</i> bytes.
/PAU	N/A	Pauses linking.
/PM:<i>type</i>	N/A	Sets window type for Presentation Manager.
/Q	N/A	Produces Quick library.
N/A	/s	Create detailed map of segments.
/SE:<i>number</i>	N/A	Sets maximum number of segments allowed.
/ST:<i>number</i>	N/A	Sets stack size.
/T	/t	Produce .COM files.
N/A	/Td	Create target DOS executable.
N/A	/Tdc	Create target DOS .COM file.
N/A	/Tde	Create target DOS .EXE file.
N/A	/Tw	Create target Windows executable (.DLL or .EXE).
N/A	/Twe	Create target Windows application (.EXE).
N/A	/Twd	Create target Windows DLL (.DLL).
/W	N/A	Warn fixups.
N/A	/x	Don't create map file.
N/A	/ye	Use expanded memory for swapping.
N/A	/yx	Use extended memory for swapping.

Source-level compatibility

The following sections tell you how to make sure that your code is compatible with Borland C++'s compiler and linker.

`__MSC` macro

The Borland C++ libraries contain many functions to increase compatibility with applications originally written in Microsoft C. If you define the macro `__MSC` before you include the `dos.h` header file, the `DOSERROR` structure will be defined to match Microsoft's format.

Header files

Some nonstandard header files can be included by one of two names, as follows.

Original name	Alias
alloc.h	malloc.h
dir.h	direct.h
mem.h	memory.h

If you are defining data in header files in your program, you should use the **-Fc** command-line compiler option or [Options | Compiler | Advanced code generation | Generate COMDEFs IDE option](#) to generate COMDEFs. Otherwise you will get linker errors. Chapter 5 of the *User's Guide* provides a complete reference to the command-line compiler options.

Memory models

Although the same names are used for the standard memory models, there are fairly significant differences for the large data models in the standard configuration.

In Microsoft C, all large data models have a default NEAR data segment to which DS is maintained. Data is allocated in this data segment if the data size falls below a certain threshold, or in a far data segment otherwise. You can set the threshold value with the **/Gtn** option, where *n* is a byte value. The default threshold is 32,767. If **/Gt** is given but *n* is not specified, the default is 256.

In all other memory models under Microsoft C, both a near and a far heap are maintained.

In Borland C++, the large and compact models (but not huge) have a default NEAR data segment to which DS is maintained. All static data is allocated to this segment by default, limiting the total static data in the program to 64K, but making all external data references near. In the huge model all data is far.

In Microsoft's version of the huge memory model, a default data segment for the entire program is maintained which limits total near data to 64K. No limit is imposed on array sizes since all extern arrays are treated as huge (**_huge**).

In Borland C++'s huge memory model, each module has its own data segment. The data segment is loaded on function entry. All data defined in a module is referenced as near data and all extern data references are far. The huge model is limited to 64K of near data in each module.

Keywords

Borland C++ supports the same set of keywords as Microsoft C 5.1 with the exception of **fortran**.

Borland C++ supports the same set of keywords as Microsoft C 6.0 with the exception of:

- **_based**, **_self**, and **_segment**, because Borland C++ does not support based pointers
- **_segment**; Borland C++'s keyword **_seg** is the equivalent of **_segment**
- **_emit**; Borland C++ uses the pseudofunction **__emit__**, because this style allows addresses of variables to be given as arguments, and allows multiple bytes to be output; **_emit**, by contrast, works like an assembly DB, allowing one immediate byte to be output
- **_fortran**; use the **_pascal** calling convention instead

Borland C++ provides **_cs**, **_ds**, **_es**, and **_ss** pointer types. See the section "Mixed model programming: Addressing modifiers" in Chapter "9" for more information.

Floating-point return values

In Microsoft C, **_cdecl** causes float and double values to be returned in the **__fac** (floating point accumulator) global variable. Long doubles are returned on the NDP stack. **_fastcall** causes floating point types to be returned on the NDP stack. **_pascal** causes the calling program to allocate space on the stack and pass address to function. The function stores the return value and returns the address.






In Borland C++, floating point values are returned on the NDP stack.

Structures returned by value

In a Microsoft C-compiled function declared with **_cdecl**, the function returns a pointer to a static location. This static location is created on a per-function basis. For a function declared with **_pascal**, the calling program allocates space on the stack for the return value. The calling program passes the address for the return value in a hidden argument to the function.

Borland C++ returns 1-byte structures in AL, 2-byte structures in AX and 4-byte structures in AX and DX. For 3-byte structures and structures larger than 4 bytes, the compiler passes a hidden argument (a far pointer) to the function that tells the function where to return the structure.

Conversion hints

-  Write *portable* code. Portable code is compatible with many different compilers and machines. Whenever possible, use only functions from the ANSI standard library (for example, use **time** instead of **gettime**). The portability bars in the *Library Reference* will tell you if a function is ANSI standard.
-  If you must use a function that's not in the ANSI standard library, use a Unix-compatible function, if possible (for example, use **chmod** instead of **_chmod**, or **signal** instead of **ctrlbrk**). Again, the portability bars in the *Library Reference* will tell you if a function is available on Unix machines.
-  Avoid the use of bit fields and code that depends on word size, structure alignment, or memory model. For example, Borland C++ defines **ints** to be 16 bits wide, but a 32-bit C++ compiler would define 32-bit wide **ints**.
-  Insert the preprocessor statement **#define _ _MSC** in each module before `dos.h` is included.
-  If you were using the link option **/STACK:n** in your Microsoft application, initialize the global variable **_stklen** with the appropriate stack size.

Building a Windows application

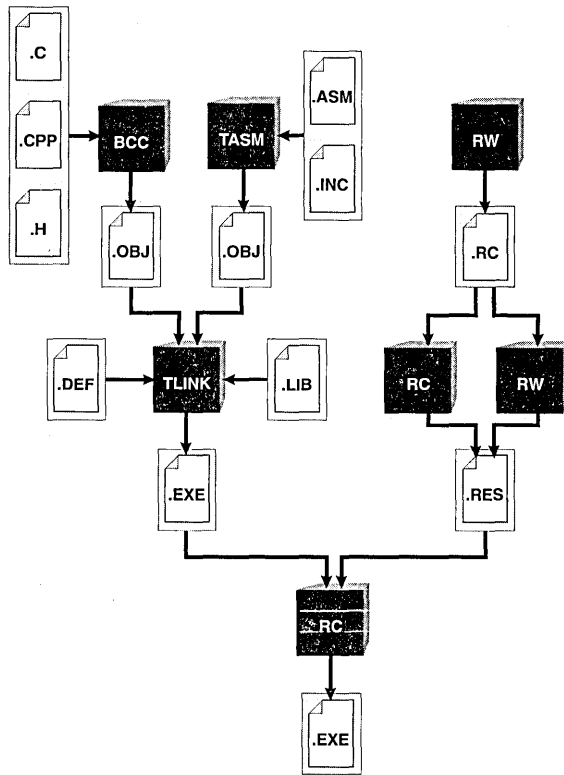
We don't explain the intricacies of designing Windows applications, nor teach you how to program under Windows—these topics go beyond the scope of this chapter or book.

This chapter explains how to use Borland C++ to build Windows applications or dynamic link libraries (DLLs). Compiling and linking a module for Windows is basically the same as it is for DOS. The compiler first generates an object file which differs from a DOS compilation primarily in the special Windows prolog and epilog code that wraps each function. The prolog and epilog code varies depending on which Windows compilation options are used; these options are described later.

To create a Windows module for the memory model you are compiling under, the linker links the object files with the appropriate Borland C++ startup code, various libraries, and the module definition file.

Finally, either the IDE, the makefile, or the programmer invokes the Resource Compiler to bind the resources to the module. Figure 8.1 illustrates the entire process.

Figure 8.1
 Compiling and linking
 a Windows program



The next section, "Compiling and linking with the IDE," gives you a quick example of how to compile, link, and run a Windows program in the Borland C++ IDE. If you normally compile and link from the command line or from a makefile, then you should read "Compiling and linking from the command line," starting on page 318.

Compiling and linking within the IDE

You can find complete descriptions of the various IDE commands and options in Chapter 2 of the User's Guide.

By way of example, you'll be producing a simple Windows application called WHELLO, which creates a window and writes a text message to that window. WHELLO.EXE is produced by compiling and linking the following three files:

- WHELLO.CPP, the C++ source file
 - WHELLO.RC, the resource file
 - WHELLO.DEF, the module definition file
-

Understanding resource files

Windows applications typically use *resources*, which can be icons, menus, dialog boxes, fonts, cursors, bitmaps, or user-defined resources. These resources are defined in a file called a resource file. For this application, the resource file is WHELLO.RC.

.RC resource files are source files, also called resource script files. Before an .RC file can be added to an executable, the .RC file must first be compiled by the Resource Compiler into a binary format; compilation creates a .RES file. For instance, compiling WHELLO.RC with the Resource Compiler creates WHELLO.RES. The Resource Compiler is also used to bind .RES resource files to an executable file.

To build a final Windows application, complete with resources, you need to invoke the Resource Compiler in order to bind the .RES file to the .EXE file. The Resource Compiler does three things:

1. It compiles .RC files to .RES files.
2. It binds the .RES file to the compiled module (.EXE or .DLL).
3. It marks the .EXE or .DLL as Windows-compatible.

Understanding module definition files

Module definition files are described in detail in Chapter 4 in the Tools and Utilities Guide.

The module definition file WHELLO.DEF provides information to the linker about the contents and system requirements of a Windows application. Because TLINK and the IDE linker have other ways of finding out the information contained in the module definition, module definition files are not required for Borland C++'s linker to create a Windows application, although one is included here for the sake of example.

Compiling and linking WHELLO

Here's how you turn these three files into a Windows application:

You can also open the WHELLO.PRJ project file, and skip the process of adding files to the project.

1. Choose Project | Open Project. In the Project Name box, type WHELLO.PRJ. Press *Enter* or click OK to open a new project with the name WHELLO.
2. Choose Project | Add item and type `whello.*` in the Name box, so that you'll get a list of all the WHELLO files. Press *Enter* or click OK.
3. Add the three files WHELLO.CPP, WHELLO.RC, and WHELLO.DEF for the application. Close the dialog box after you've added the three files.
4. Choose Options | Application to open the Application Options dialog box and select Windows App if it's not already selected. The information pane at the top of the dialog box changes. Each of the buttons at the bottom of the dialog box sets several other options in the IDE.
5. Choose Compile | Build all to build the project.
6. Exit the IDE by pressing *Alt-X* or choosing File | Quit.
7. From the DOS command line, type
`win whello`
DOS will load Windows, which will itself run the WHELLO application.

That's all there is to building and running a Windows application with Borland C++. You can generalize this process into the following checklist:

1. Create a project.
2. Add the source files, resource files, import libraries (if necessary), and the module definition file (if necessary) to the project.
3. Set up the compilation and link environment with the Application Options dialog box, or with a combination of other settings and options.
4. Build the project.
5. Run the application under Windows.

Using the project manager

Specifying an .RC file is similar to specifying a source file in a project. The Project Manager will invoke the Resource Compiler once to compile it to a .RES file, and a second time to bind the .RES to the module and to mark the module as Windows-compatible.

Specifying a .RES file is similar to specifying an object file. The Project Manager will invoke the Resource Compiler only to bind it to the module and to mark the module as Windows-compatible.

For example, if you enter HELLO.CPP, HELLO.RC, and HELLO.DEF into a project, the Borland C++ Project Manager will

- create HELLO.OBJ by compiling HELLO.CPP with the C++ compiler
- create HELLO.RES by compiling HELLO.RC with the Resource Compiler
- create HELLO.EXE by linking HELLO.OBJ with its appropriate libraries, using information contained in HELLO.DEF
- create the final HELLO.EXE by using the Resource Compiler to bind the resources contained in HELLO.RES to HELLO.EXE

Setting compile and link options

The bulk of the setup in this example is accomplished by the Application Options dialog box. The command buttons in this dialog box check or set various other options in other dialog boxes. Borland C++ makes it easy for you to change the settings that control compilation and linking of your programs, so you'll want to familiarize yourself with the following dialog boxes (all described in full in Chapter 3 in the *User's Guide*):

- The Code Generation Options dialog box sets such things as the memory model, tells the compiler to use precompiled headers, and more. Choose Options | Compiler | Code Generation to see this dialog box.
- The Entry/Exit Code Generation dialog box sets Borland C++ compiler options for prolog and epilog code generation, and export options. Choose Options | Compiler | Entry/Exit Code and browse through the contents of this dialog box.
- The Make dialog box (Options | Make). The Generate Import Library options allow you to create an import library for a DLL. An import library makes it possible to declare all the functions in a DLL as imports to another module without using a module definition file (see Chapter 1 in the *Tools and Utilities Guide*).
- The Linker Settings dialog box (Options | Linker | Settings) sets options for the type of output you want from the linker—such as a standard DOS .EXE, an overlaid DOS .EXE, a Windows .EXE, or a Windows DLL—as well as a number of other linker options.

WinMain

You must supply the **WinMain** function as the main entry point for a Windows application;

The following parameters are passed to **WinMain**:

The HANDLE and LPSTR types are defined in windows.h.

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,  
                  LPSTR lpCmdLine, int nCmdShow)
```

- *hInstance* is the instance handle of the application. Each instance of a Windows application has a unique instance handle that's used as an argument to several Windows functions and can be used to distinguish between multiple instances of a given application.
- *hPrevInstance* is the handle of the previous instance of this application. *hPrevInstance* is NULL if this is the first instance.
- *lpCmdLine* is a far pointer to a null-terminated command-line string. This value can be specified when invoking the application from the program manager or from a call to **WinExec**.
- *nCmdShow* is an integer that specifies how to display the application's window.

The return value from **WinMain** is not currently used by Windows. However, it can be useful during debugging since Turbo Debugger for Windows can display this value when your program terminates.

Compiling and linking from the command line

If you know how to compile and link a C++ or C program for DOS, then you already know almost all you need to do the same thing for Windows. You'll need three files to compile and link the example application:

- WHELLO.CPP, the C++ source code
- WHELLO.DEF, the module definition file
- WHELLO.RC, the resource file

Compiling from the command line

To compile and link WHELLO.CPP for a Windows application, type

```
BCC -W whello.cpp
```

Given this command line, Borland C++ compiles WHELLO.CPP into WHELLO.OBJ, then links in the correct libraries and startup code automatically. To suppress the link phase, add the **-c** option to the command line. To include debugging information, add the **-v** option.

You can find detailed descriptions of all the command-line options in Chapter 5 in the User's Guide.

The **-W** option tells the command-line compiler that you want a Windows application. There are other Windows options (of the form **-Wxxx**) that give the compiler more specific instructions about the compilation and code generation of a Windows application (for instance, **-WD** to create a DLL).

Once the WHELLO application is compiled and linked, the only thing left to do is add the resources. First, compile the WHELLO.RC file with the command

```
rc -r whello.rc
```

This produces a WHELLO.RES file (**-r** instructs the Resource Compiler to *not* add the result to the executable of the same name). Now, invoke the Resource Compiler again to add the binary resource file to the executable:

```
rc whello.res whello.exe
```

Actually, the Resource Compiler makes it easier than we've shown here, because it can compile an .RC file into a .RES file and then add it to the executable all in one step. Furthermore, if the executable file has the same first name as the resource file, then you don't need to specify the executable file on the command line at all. So, the previous two commands can be rewritten like this:

```
rc whello
```

To load Windows and run the application, type

```
win whello
```

Linking from the command line

To link WHELLO.OBJ with the correct libraries and startup code, invoke TLINK with the following command-line:

```
TLINK /Tw /v /c /LC:\BORLANDC\LIB c0ws whello, whello, , import mathws  
cws, whello
```

The TLINK command line is composed of options and five file names or groups of file names; each file or group of files is separated by a comma.

The **/Tw** option means to link for (target) Windows, **/v** tells TLINK to include debugging information, and **/c** forces case to be significant in public and external symbols. **/L** followed by a path name, tells TLINK where to look for library files and for the startup .OBJ code.

The object files to link are listed next in the command line. C0WS.OBJ is the initialization module for the small memory model, and WHELLO.OBJ is the program module for this application. The .OBJ extension is assumed for both these files.

For more details on how TLINK knows whether you want an .EXE or .DLL, see the section "Linker options" on page 329.

The next file on the command line, WHELLO, is the name you want TLINK to give the executable file. The .EXE extension is assumed when you create a Windows application, and the .DLL extension is assumed when you create a DLL.

The next file on the command line is the name you want to give the map file. If no name is given, as in this example, TLINK gives the map file the name of the executable and adds the .MAP extension. After you run this command, you'll notice the file WHELLO.MAP in the examples directory.

The library files to link are listed after the map file. CWINS.LIB is the small memory model run-time library for Windows, CS.LIB is the regular run-time library, and IMPORT.LIB is the library that provides access to the built-in Windows functions. The .LIB extension is assumed for all library files.

The last file name on the TLINK command line is the module definition file, WHELLO.DEF (the .DEF extension is assumed). Module definition files are described briefly on page 315, and in detail in Chapter 4 in the *Tools and Utilities Guide*.

Using a makefile

Since you probably won't want to type in the full command lines for the command line compiler and TLINK every time you want to build a Windows application, it's a good idea to create a makefile for your application.

The makefile for the WHELLO application is WHELLO.MAK. Note that for this example, the libraries are in C:\BORLANDC\LIB, and the include files are in C:\BORLANDC\INCLUDE. The following section explains each rule in the makefile.

To run MAKE on this makefile, type

```
make -f whello.mak
```

The first rule tells MAKE how to make the final executable from WHELLO.EXE and a WHELLO.RES, and how to make the intermediate executable from the object file and the module definition file. (See the alternate makefile at the end of this section for a more generalized approach to building a Windows application.)

```
whello.exe: whello.obj whello.def whello.res
    tlink /Tw /v /n /c C:\BORLANDC\LIB\c0ws whello,\
        whello,\
        ,\
        C:\BORLANDC\LIB\cwins C:\BORLANDC\LIB\cs
    C:\BORLANDC\LIB\import,\
        whello
    rc whello.res
```

The next rule tells MAKE how to make required .OBJ files from .CPP files of the same name. The options are: make a Windows application (**-W**), compile only (**-c**), use the small memory model (**-ms**), and include debugging info (**-v**).

```
.cpp.obj:
    BCC -c -ms -v -W $<
```

This last rule tells MAKE how to make required .RES files (final resource files) from .RC files of the same name.

```
.rc.res:
    rc -r -iC:\BORLANDC\INCLUDE $<
```

The **-r** option tells the Resource Compiler to compile the resources only (instead of also adding them to the executable of

the same name). The `-i` options specifies the directory in which to search for include files.

Another makefile for Windows

The following makefile is a more general-purpose makefile than the one shown previously. It can be easily modified by redefining the macros `OBJS`, `INCPATH`, and `FLAGS`. `TLINK` is not invoked in a separate rule; instead, `BCC` invokes `TLINK` automatically.

```
OBJS = whello.obj
INCPATH = C:\BORLANDC\INCLUDE
FLAGS = -W -v -I$(INCPATH)

test.exe: $(OBJS) whello.def whello.res
    BCC $(FLAGS) -ewhello.exe @&&!
$(OBJS)
!
    rc whello.res

.c.obj:
    BCC -c $(FLAGS) {$< }

.cpp.obj:
    BCC -c $(FLAGS) {$< }

.rc.res:
    rc -r -i$(INCPATH) $<
```

Prologs and epilogs

The need for prologs and epilogs is not new to Windows; they must be generated for code intended for DOS as well. However, if the program is intended for Windows, the compiler generates a different prolog and epilog than it would for DOS.

When you compile a module for Windows, the compiler needs to know which kind of prolog and epilog to create for each of a module's functions. Settings in the IDE and options for the command-line compiler control the creation of the prolog and epilog. The prolog and epilog perform several functions, including ensuring that the correct data segment is active during callback functions, and marking near and far stack frames for the Windows stack-crawling mechanism.

The prolog and epilog code is automatically generated by the compiler, though various compiler options or IDE options dictate the exact instructions contained in the code.

The following list describes the effects of the different prolog/epilog code generation options and their corresponding command-line compiler options. To set these options in the IDE, choose Options | Compiler | Entry/Exit Code.

Windows All Functions Exportable (-W)

This option creates a Windows application object module with all far functions exportable.

This is the most general kind of Windows application module, although not necessarily the most efficient. The compiler generates a prolog and epilog for every far function that makes the function exportable. This does not mean that all far functions actually will be exported, it only means that the function can be exported. In order to actually export one of these functions, you must either use the **_export** keyword or add an entry for the function name in the EXPORTS section of the module definition file.

*See page 52 for description and usage of the **_export** keyword.*

Windows Explicit Functions Exported (-WE)

This option creates an object module with only those functions marked as **_export** exportable.

Since, in any given application module, many of the functions won't be exported, it is not necessary for the compiler to include the special prolog and epilog for exportable functions unless a particular function is known to be exported. The **_export** keyword in a function definition tells the compiler to use the special prolog and epilog required for exported functions. All functions not flagged with **_export** receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

Note that the Windows Explicit Functions Exported option *only* works in conjunction with the **_export** keyword. This option does not export those functions listed in the EXPORTS section of a module definition file. In fact, you can't use this option and provide the names of the exported functions in the EXPORTS section. If you do, the compiler will generate prolog and epilog code that is incompatible with exported functions; incorrect behavior will result when these functions are called.

Windows Smart Callbacks (-WS)

This option creates an object module with functions using smart callbacks.

This form of prolog and epilog assumes that DS == SS; in other words, that the default data segment is the same as the stack

segment. This eliminates the need for the special Windows code (called a *thunk*) created for exported functions. Using smart callbacks can improve performance because calls to functions in the module don't have to be redirected through the thunks.

Exported functions here don't need the **_export** keyword or to be listed in the EXPORTS section of the module definition file, because the linker doesn't need to create an export entry for them in the executable.

When you use functions compiled and linked with smart callbacks, you don't need to precede them with a call to **MakeProclnstance** (which rewrites the function's prolog in such a way that it uses a smart callback).

There are no smart callbacks for DLLs since DLLs assume DS != SS.

Because of the assumption that DS == SS, you can only use this option for applications, not DLLs. Furthermore, you must not explicitly change DS in your program (a very unsafe practice under Windows in any circumstance).

Windows DLL All Functions Exportable (-WD)

This option creates a DLL object module with all functions exportable. This prolog and epilog code is used for functions that will reside in a DLL. It also supports the exporting of these functions. This is similar to the corresponding non-DLL option.

Windows DLL Explicit Functions Exported (-WDE)

This prolog and epilog code is also used for functions that will reside in a DLL. However, any functions that will be exported must explicitly specify **_export** in the function definition. This is similar to the corresponding non-DLL option.

The **_export** keyword

*Note that exported functions must be declared far; you can use the **FAR** type, defined in windows.h.*

The keyword **_export** in a function definition tells the compiler to compile the function as exportable and tells the linker to export the function. In a function declaration, **_export** immediately precedes the function name; for example,

```
LONG FAR PASCAL _export MainWindowProc( HWND hWnd, unsigned iMessage, WORD wParam, LONG lParam )
```

You can also use **_export** with a C++ class definition; see page 336.

Prologs, epilogs, and exports: a summary

There are two steps to exporting a function. First, the compiler must create the correct prolog and epilog for the function; if so, the function is called exportable. Second, the linker must create an entry for every export function in the header section of the executable. All of this occurs so that the correct data segment can be bound to the function at run time.

If a function is flagged with the `_export` keyword and any of the Windows compiler options are used, it will be compiled as exportable and linked as an export.

If a function is *not* flagged with the `_export` keyword, then Borland C++ will take one of the following actions:

- If you compile with the `-W` or `-WD` option (or with the IDE equivalent of either option), the function will be compiled as exportable.

If the function is listed in the EXPORTS section of the module definition file, then the function will be linked as an export. If it is not listed in the module definition file, or if no module definition file is linked, then it won't be linked as an export.

- If you compile with the `-WE` or `-WDE` option (or with the IDE equivalent of either option), the function will *not* be compiled as exportable. Including this function in the EXPORTS section of the module definition will cause it be exported, but, because the prolog is incorrect, the program will run incorrectly. You may get the Windows error message, "Unrecoverable Application Error."

Table 8.1 summarizes the effect of the combination of the Windows compiler options and the `_export` keyword:

Table 8.1: Compiler options and the `_export` keyword

Function flagged with <code>_export</code> and <code>far</code> ?	Yes	Yes	Yes	Yes	No	No	No	No
Function listed in EXPORTS?	Yes	Yes	No	No	Yes	Yes	No	No
And the compiler option is:	<code>-W</code> or <code>-WD</code>	<code>-WE</code> or <code>-WDE</code>	<code>-W</code> or <code>-WD</code>	<code>-WE</code> or <code>-WDE</code>	<code>-W</code> or <code>-WD</code>	<code>-WE</code> or <code>-WDE</code>	<code>-W</code> or <code>-WD</code>	<code>-WE</code> or <code>-WDE</code>

Table 8.1: Compiler options and the `_export` keyword (continued)

Will function be exportable?	Yes	Yes	Yes	Yes	Yes	No	Yes	No
Will function be exported?	Yes	Yes	Yes	Yes	Yes	Yes**	No***	No

** The function will be exported in some sense, but, because the prolog and epilog won't be correct, the function won't work as expected.

*** This combination also makes little sense. It's inefficient to compile all functions as exportable if you don't actually export some of them.

Memory models

See the section "Linking .OBJ and .LIB files for DLLs" on page 331 for more information.

You can use the small, medium, compact, or large memory models with any kind of Windows executable, including DLLs. Windows doesn't support the tiny or huge memory models.

Module definition files

The module definition file is not strictly necessary to produce a Windows executable under Borland C++. If no module definition file is specified, the following defaults are assumed.

```

CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE MULTIPLE (for
              applications) or PRELOAD MOVEABLE
              SINGLE (for DLLs)
HEAPSIZE      4096
STACKSIZE     5120
    
```

To replace the `EXETYPE` statement, the Borland C++ linker can discover what kind of executable you want to produce by checking settings in the IDE or options on the command line.

You can include an import library to substitute for the `IMPORTS` section of the module definition.

You can use the `_export` keyword in the definitions of export functions in your C and C++ source code to remove the need for an `EXPORTS` section. Note, however, that if `_export` is used to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need to have a module definition file.

A quick example

Here's the module definition from the WHELLO example:

```
NAME            WHELLO
DESCRIPTION     'C++ Windows Hello World'
EXETYPE        WINDOWS
CODE           MOVEABLE
DATA          MOVEABLE MULTIPLE
HEAPSIZE      1024
STACKSIZE     5120
EXPORTS       MainWindowProc
```

Let's take this file apart, statement by statement:

- ❑ **NAME** specifies a name for an application. If you want to build a DLL instead of an application, you would use the **LIBRARY** statement instead. Every module definition file should have either a **NAME** statement or a **LIBRARY** statement, but never both. The name specified must be the same name as the executable file.
- ❑ **DESCRIPTION** lets you specify a string that describes your application or library.
- ❑ **EXETYPE** can be either **WINDOWS** or **OS2**. Only **WINDOWS** is supported in this version of Borland C++.
- ❑ **CODE** defines the default attributes of code segments. The **MOVEABLE** option means that the code segment can be moved in memory at run-time.
- ❑ **DATA** defines the default attributes of data segments. **MOVEABLE** means that it can be moved in memory at run-time. **WINDOWS** lets you run more than one instance of an application at the same time. In support of that, the **MULTIPLE** options ensures that each instance of the application has its own data segment.
- ❑ **HEAPSIZE** specifies the size of the application's local heap.
- ❑ **STACKSIZE** specifies the size of the application's local stack. You can't use the **STACKSIZE** statement to create a stack for a DLL.
- ❑ **EXPORTS** lists those functions in the **WHELLO** application that will be called by other applications or by Windows. Functions

that are intended to be called by other modules are called callbacks, callback functions, or export functions.

- To help you avoid the necessity of creating and maintaining long EXPORTS sections, Borland C++ provides the **_export** keyword. Functions flagged with **_export** will be identified by the linker and entered into an export table for the module. If the Smart Callbacks option is used at compile time (**/WS** on the BCC command-line, or Options | Compiler | Entry/Exit Code), then callback functions do *not* need to be listed either in the EXPORTS statement or flagged with the **_export** keyword. Borland C++ compiles them in such a way so that they can be callback functions.

This application doesn't have an IMPORTS statement, because the only functions it calls from other modules are those from the Windows API; those functions are imported via the automatic inclusion of the IMPORT.LIB import library. When an application needs to call other external functions, these functions must be listed in the IMPORTS statement, or included via an import library (see page 333 for a discussion of import libraries).

This application doesn't include a STUB statement. Borland C++ uses a built-in stub for Windows applications. The built-in stub simply checks to see if the application was loaded under Windows, and, if not, terminates the application with a message that Windows is required. If you want to write and include a custom stub, specify the name of that stub with the STUB statement.

Linking for Windows

TLINK is discussed in detail in Chapter 4, "TLINK: The Turbo linker" in the Tools and Utilities Guide

In general, Borland C++ needs to take object files compiled with the .correct Windows options and then link them with the proper Windows initialization code, run-time and math libraries, and a module definition file. Settings in the Linker Settings dialog box in the IDE do this for you automatically; if you use TLINK, you must specify all the options and files.

Linking in the IDE

With the Linker Settings dialog box in the IDE, you can set link options for a Windows application or DLL. Options in the IDE override settings in the module definition file. This means if you check the Windows EXE box instead of the Windows DLL box, and the module definition file has a LIBRARY statement instead of a NAME statement, the file will be linked as a Windows application, not a DLL.



The linker uses the C0Wx.OBJ initialization file for applications and the C0Dx.OBJ initialization file for DLLs, where *x* depends on the memory model set in the Code Generation dialog box. For both Windows options, the linker uses the current project object files and libraries, IMPORT.LIB, MATHWx.LIB, and CWx.LIB. Borland C++ allows you to override the default setting for a memory model.

Linking with TLINK

For a list of TLINK messages (errors and warnings), see Appendix A in the Tools and Utilities Guide.

Linker options

The syntax of the TLINK command line is:

TLINK *objfiles, exe file, mapfile, libfiles, deffile*

There are three options that you can pass to TLINK to control its linkage of Windows executables and DLLs.

- Use the **/Tw** option to create a Windows .EXE or .DLL according to the settings in the module definition file. If you have a NAME statement in the module definition file, TLINK will link it as a Windows executable; if you have a LIBRARY statement in the .DEF file, the files will be linked as a DLL.
If no module definition file is specified on the TLINK command line, this option causes the files to be linked as a Windows .EXE. You don't need this option if you are using a module definition file in which the EXETYPE statement specifies WINDOWS.
- Use the **/Twe** option to specify a Windows executable. This overrides settings in the module definition file. For instance, even if you have a LIBRARY statement in the include .DEF file, TLINK will link the files as an .EXE.
- Use the **/Twd** option to specify a Windows DLL. This overrides settings in the module definition file.

When you're linking a Windows executable, do *not* use the **b** option to overlay files, or the **/t** or **/Tdc** option to make a .COM file.

Linking .OBJ and .LIB files

The list of object files must begin with the file C0Wx.OBJ or C0Dx.OBJ (for DLLs), followed by the names of the other object files to link. User libraries and IMPORT.LIB can be included anywhere on the list, although, by convention, they are usually listed before the standard libraries. The other required libraries must be in this order:

***Important!** Do not link in EMU.LIB or FP87.LIB for a Windows application. Borland C++ takes care of the floating-point math automatically.*

- MATHWx.LIB
- CWx.LIB

To create a Windows application executable, you might use this response file, named WINRESP:

```
/Tw /c \BORLANDC\LIB\C0WS winapp1 winapp2
winapp
winapp
\BORLANDC\LIB\IMPORT \BORLANDC\LIB\MATHWS \BORLANDC\LIB\CWS
winapp.def
```

where

- The **/Tw** option tells TLINK to generate a Windows application or DLL. If a module definition file were not included in the link, TLINK would create a Windows application. If the module definition file is included and it contains instructions to create a DLL, then TLINK will create a DLL.
- The **/c** option tells TLINK to be sensitive to case during linking.
- BORLANDC\LIB\C0WS is the standard Windows initialization file and WINAPP1 and WINAPP2 are the module's object files.
- WINAPP is the name of the target Windows executable.
- TLINK will name the map file WINAPP.MAP.
- BORLANDC\LIB\IMPORT is the library that provides access to the built-in Windows functions, BORLANDC\LIB\MATHWS is the small memory model floating point math library for Windows and BORLANDC\LIB\CWS is the small memory model run-time library for Windows.
- WINAPP.DEF is the Windows module definition file for the object files named.

To use this response file on the TLINK command line, type

TLINK @winresp



After linking the application or DLL, you *must* invoke the Resource Compiler to add resources to the image. The Windows 3.x Resource Compiler also marks the image as Windows 3.x compatible. Even if you have no resources, you need to run the Resource Compiler.

Linking .OBJ and .LIB files for DLLs

You need to link different .OBJ and .LIB files for a DLL than for a Windows application. If the linker is invoked either from the IDE or from the command-line compiler BCC, the correct .OBJ and .LIB files will be linked in automatically. If you invoke TLINK explicitly, then you need to know which files to link in for a DLL. The following table summarizes the memory models, startup files, and libraries:

Table 8.2
Startup and library files for
DLLs

Model	Startup file	Library
Small	C0DS.OBJ	CWC.LIB
Compact	C0DC.OBJ	CWC.LIB
Medium	C0DM.OBJ	CWL.LIB
Large	C0DL.OBJ	CWL.LIB

The compact memory model library is used for both small and compact because it creates far data pointers and near code pointers. The large memory model library is used for both medium and large because it creates far data pointers as well as far code pointers. DLLs can only have far pointers to data; near pointers are not allowed.

When you add an .RC file to a project, the Project Manager automatically assigns the default translator to be the Resource Compiler. In addition, the default output name is *file.RES* (not *file.OBJ*). Finally, "Exclude from Link" is selected because TLINK should not link the resulting .RES file.

During a make, the Project Manager recompiles the .RC file if it is newer than the .RES file, in the same way that it recompiles HELLO.C if it is newer than HELLO.OBJ. No autodependencies are checked because that information is not available.

During a make, the Project Manager runs the Resource Compiler after any relink because the Resource Compiler also marks the image as Windows 3.x compatible. Even if you have no resources, you need to run the Resource Compiler.

Dynamic link libraries

A dynamic link library (DLL) is a library of functions that a Windows module can call to accomplish a task. If you've written a Windows application, then you've already used DLLs. The files KERNEL.EXE, USER.EXE, and GDI.EXE are actually DLLs, not applications (as the .EXE extension implies). The references to the API functions that you call from these modules are resolved at run time (dynamic linking), instead of at link time (static linking).

Compiling and linking a DLL within the IDE

To compile and link a DLL from within the IDE, follow these steps:

1. Create the DLL source files. Optionally, create the resource file and the module definition file.
2. Choose Project | Open Project to start a new project.
3. Choose Project | Add Item, and add the source and resource files for the DLL.
4. If you have created a module definition file for the DLL, add it to the project. (Note that Borland C++ can link without one. To link without a module definition file for the DLL, you must have flagged every function to be exported in the DLL with the keyword **_export**. In addition, choose Options | Compiler | Entry/Exit Code | Windows DLL Explicit Functions Exportable.)
5. Choose Options | Application | Windows DLL.
6. Choose Compile | Build all.

*The **_export** keyword should immediately precede the function name.*

Compiling and linking a DLL from the command line

To compile and link a DLL composed of the source file LIBXAMP.CPP, type

```
BCC -WD libxamp.cpp
```

The command-line compiler takes care of linking in the correct startup code and libraries. The **-WD** option tells the compiler to build a Windows DLL with all functions exportable. To compile and link with explicit functions exportable, you would use the **-WDE** option and use the **_export** keyword for export functions.

To link a DLL with the command-line linker TLINK, you might use this command line

```
TLINK /Twd /v /c /LC:\BORLANDC\LIB c0ds libxamp, libxamp, , import  
mathwc cwc,libxamp
```

See page 331 for an explanation of the library and object files needed to link a DLL.

The **/Twd** option indicates a Windows DLL, **/v** tells TLINK to include debugging information, and **/c** forces case to be significant in public and external symbols. The **/L** option specifies a library and startup file search path.

Module definition files

A module definition file is not strictly necessary to link either a DLL or a Windows application.

See Chapter 4 in the Tools and Utilities Guide for information on default module definition file replacement settings.

There are two ways to tell the linker about export functions:

- ▣ To link with a module definition file, create an EXPORTS section in the module definition file that lists all the functions that will be used by other applications. (IMPDEF can help you do this, see Chapter 1 in the *Tools and Utilities Guide*.)
- ▣ To link without a module definition file, you must flag every function to be exported in the DLL with the keyword **_export**. In addition, when you build or link the DLL, you must choose Options | Compiler | Entry/Exit Code | Windows DLL Explicit Functions Exportable (or **-WDE** on the command line).

A function must be exported from a DLL before it can be imported to another DLL or application.

Import libraries

If a Windows application module or another DLL uses functions from a DLL, you have two ways to tell the linker about them:

- ▣ You can add an IMPORTS section to the module definition file and list every function from DLLs that the module will use.
- ▣ Or you can include the import library for the DLLs when you link the module. (A utility called IMPLIB creates an import library for one or more DLLs; see Chapter 1 in the *Tools and Utilities Guide* for details.)

Creating DLLs

The following sections provide information on the specifics of writing a DLL.

LibMain and WEP

You must supply the **LibMain** function as the main entry point for a Windows DLL.

Windows calls **LibMain** once, when the library is first loaded. **LibMain** performs initialization for the DLL. This initialization depends almost entirely on the function of the particular DLL, but might include the following tasks:

- Unlocking the data segment with **UnlockData**, if it has been declared as MOVEABLE
- Setting up global variables for the DLL, if it uses any



The DLL startup code `C0Dx.OBJ` initializes the local heap automatically; you do not need to include code in **LibMain** to do this.

The following parameters are passed to **LibMain**:

HANDLE, WORD, and LPSTR are defined in windows.h.

```
int FAR PASCAL LibMain (HANDLE hInstance, WORD wDataSeg,  
                        WORD cbHeapSize, LPSTR lpCmdLine)
```

- *hInstance* is the instance handle of the DLL.
- *wDataSeg* is the value of the data segment (DS) register.
- *cbHeapSize* is the size of the local heap specified in the module definition file for the DLL.
- *lpCmdLine* is a far pointer to the command line specified when the DLL was loaded. This is almost always null since DLLs are typically loaded automatically with no parameters. It is possible, however, to supply a command line to a DLL when it is loaded explicitly.

The return value for **LibMain** is either 1 (successful initialization) or 0 (failure in initialization). If 0, Windows will unload the DLL from memory.

The exit point of a DLL is the function **WEP** (which stands for Windows Exit Procedure). This function is not necessary in a DLL (since the Borland C++ run-time libraries provide a default) but can be supplied by the writer of a DLL to perform any cleanup of the DLL before it is unloaded from memory. Windows will call **WEP** just prior to unloading the DLL.

Under Borland C++, **WEP** does not need to be exported. Borland C++ defines its own **WEP** that calls your **WEP**, and then performs system cleanup. This is the prototype for **WEP**:

```
int FAR PASCAL WEP (int nParameter)
```

■ *nParameter* is either `WEP_SYSTEMEXIT` or `WEP_FREE_DLL`.

The former means that all of Windows is shutting down and the latter indicates that just this DLL is being unloaded.

WEP should return 1 to indicate success. Windows currently doesn't do anything with this return value.

Pointers and memory

Functions in a DLL are not linked directly into a Windows application; they are called at run time. This means that calls to DLL functions will be far calls, because the DLL will have a different code segment than the application. The data used by called DLL functions will need to be far as well.

Let's say you have a Windows application called APP1, a DLL defined by `LSOURCE1.C`, and a header file for that DLL called `lsource1.h`. Function **f1**, which operates on a string, is called by the application.

If you want the function to work correctly regardless of the memory model the DLL will be compiled under, you need to explicitly make the function and its data far. In the header file, the function prototype would take this form:

```
extern int _export FAR f(char FAR *dstring);
```

In the DLL, the implementation of the function would take this form:

```
int FAR f1(char far *dstring)
{
  :
}
```

For the function to be used by the application, the function would also need to be compiled as exportable and then exported. To accomplish this, you can either compile the DLL with all functions exportable (**-WD**) and list **f1** in the `EXPORTS` section of the module definition file, or you can flag the function with the **_export** keyword, like so:

```
int FAR _export f1(char far *dstring)
{
  :
}
```

Before an application could use `f1`, it would have to be imported into the application, either by listing `f1` in the `IMPORTS` section of a module definition file, or by linking with an import library for the DLL. See Chapter 1, "Import library tools" in the *Tools and Utilities Guide* for more information about import libraries.

If you compile the DLL under the large model (far data, far code), then you don't need to explicitly define the function or its data far in the DLL. In the header file, the prototype would still take this form

```
extern int FAR f(char FAR *dstring);
```

because the prototype would need to be correct for a module compiled with a smaller memory model. But in the DLL, the function could be defined like this:

```
int _export f1(char *dstring)
{
  :
}
```

Static data in DLLs

Through a DLL's functions, all applications using the DLL have access to that DLL's global data. A particular function will use the same data, regardless of the application that called it. If you want a DLL's global data to be protected for use by a single application, you would need to write that protection yourself. The DLL itself does not have a mechanism for making global data available to a single application. If you need data to be private for a given caller of a DLL, you will need to dynamically allocate the data and manage the access to that data manually. Static data in a DLL is global to all callers of a DLL.

C++ classes and pointers

A C++ class used only inside a DLL doesn't need to be declared **far**. The class requires special handling if it will be used from another DLL or a Windows application.

All the members of a shared class must be far. Do this by declaring the class members as **far** or compiling the DLL under the large memory model. The classes also must be exported, which can be accomplished two ways:

- Include the names of all the class members in the `EXPORTS` section of the module definition file, then compile the DLL with the Options | Compiler | Entry/Exit code | Windows DLL All Functions Exportable(-**WD**) option.
- Mark the entire class with the `_export` keyword and compile the DLL with the Options | Compiler | Entry/Exit code | Windows DLL Explicit Functions Exported(-**WDE**) option.

C++ classes use virtual table pointers and include a hidden **this** pointer. Both pointers must be far pointers as well. There are two basic ways to accomplish this.

One way is to simply compile the DLL modules and the application using the DLL with the Far Virtual Tables option (Options | Compiler | C++ Options in the IDE or **-vf** from the command line). This causes all virtual table pointers and **this** parameters to be full 32-bit pointers. The advantage of this approach is that it does not require any source code changes; however, all classes, shared or not, suffer the overhead of 32-bit pointers.

Note that a huge class can only inherit from other huge classes.

A more efficient approach is to declare the shared classes **huge** instead of **far** which tells the compiler to use full 32-bit pointers for those classes only. Here is an example of a huge class declaration:

```
class huge DLLclass
{
:
};
```

For a class that is defined in a DLL to be usable from a Windows application, its non-inline member functions and static data members must be made available by making them exported names. You can do this by adding their public (mangled) names to the EXPORTS section of the DLL module definition file, but this can be rather tedious.

There's an easier alternative: Declare the classes to be exported as **_export**. Whenever a class is declared as **_export**, Borland C++ treats it as huge (with 32-bit pointers), and automatically exports all its non-inline member functions and static data members. If you declare a class as **_export**, you can't also declare it as **far** or **huge** (**_export** implies **huge**, which implies **far**).

If you declare the class in an include file that is included both by the DLL source files and by the source files of the application using the DLL, such a class should be declared **_export** when compiling the DLL, and merely **huge** when compiling the application. To do this, you can use the **__DLL__** macro, which is defined by the compiler when it's building a DLL. The following code could be a part of an include file that defines a shared class:

```
#ifndef __DLL__
# define EXPORT _export
```

```
#else
# define EXPORT huge
#endif

class EXPORT DLLclass
{
:
};
```

Note that the compiler encodes (in the mangled name) the information that a given class member is a member of a huge class. This ensures that any mismatches are caught by the linker when a program is using huge and non-huge classes.

DOS memory management

This chapter covers

- What to do when you receive “Out of memory” errors.
- What memory models are: how to choose one, and why you would (or wouldn’t) want to use a particular memory model.
- How **Overlays** work, and how to use them.

See Chapter 8, “Building a Windows application,” in this book for information on choosing a memory model for Windows modules.

Running out of memory

Borland C++ does not generate any intermediate data structures to disk when it is compiling (Borland C++ writes only .OBJ files to disk); instead it uses RAM for intermediate data structures between passes. Because of this, you might encounter the message “Out of memory” if there is not enough memory available for the compiler.

The solution to this problem is to make your functions smaller, or to split up the file that has large functions.

Memory models

Borland C++ gives you six memory models, each suited for different program and code sizes. Each memory model uses memory differently. What do you need to know to use memory

See page 346 for a summary of each memory model.

models? To answer that question, we have to take a look at the computer system you're working on. Its central processing unit (CPU) is a microprocessor belonging to the Intel iAPx86 family; an 80286, 80386, or 80486. For now, we'll just refer to it as an 8086.

The 8086 registers

These are some of the registers found in the 8086 processor. There are other registers—but they can't be accessed directly, so they're not shown here.

Figure 9.1
8086 registers

General-purpose registers	
AX	accumulator (math operations)
	AH AL
BX	base (indexing)
	BH BL
CX	count (indexing)
	CH CL
DX	data (holding data)
	DH DL

Segment address registers	
CS	code segment pointer
DS	data segment pointer
SS	stack segment pointer
ES	extra segment pointer

Special-purpose registers	
SP	stack pointer
BP	base pointer
SI	source index
DI	destination index

General-purpose registers

The general-purpose registers are the ones used most often to hold and manipulate data. Each has some special functions that only it can do. For example,

- Some math operations can only be done using AX.
- BX can be used as an index register.

- CX is used by LOOP and some string instructions.
- DX is implicitly used for some math operations.

But there are many operations that all these registers can do; in many cases, you can freely exchange one for another.

Segment registers The segment registers hold the starting address of each of the four segments. As described in the next section, the 16-bit value in a segment register is shifted left 4 bits (multiplied by 16) to get the true 20-bit address of that segment.

Special-purpose registers

The 8086 also has some special-purpose registers:

- The SI and DI registers can do many of the things the general-purpose registers can, plus they are used as index registers. They're also used by Borland C++ for register variables.
- The SP register points to the current top-of-stack and is an offset into the stack segment.
- The BP register is a secondary stack pointer, usually used to index into the stack in order to retrieve arguments or automatic variables.

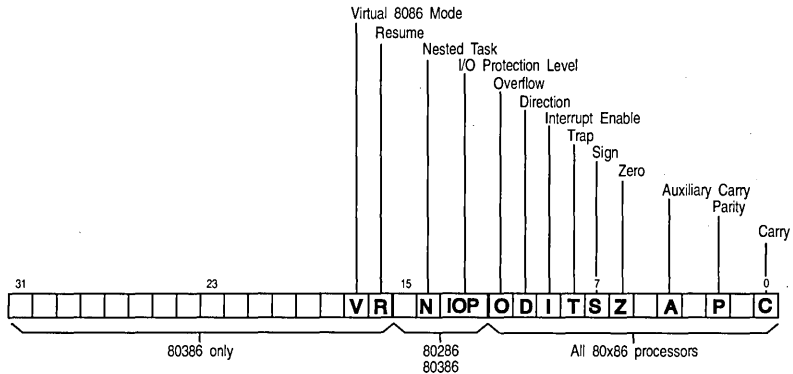
Borland C++ functions use the base pointer (BP) register as a base address for arguments and automatic variables. Parameters have positive offsets from BP, which vary depending on the memory model. BP points to the saved previous BP value if there is a stack frame. Functions that have no arguments will not use or save BP if the Standard Stack Frame option is *Off*.

Automatic variables are given negative offsets from BP. The offsets depend on how much space has already been assigned to local variables.

The flags register The 16-bit flags register contains all pertinent information about the state of the 8086 and the results of recent instructions.

For example, if you wanted to know whether a subtraction produced a zero result, you would check the *zero flag* (the Z bit in the flags register) immediately after the instruction; if it were set, you would know the result was zero. Other flags, such as the *carry* and *overflow flags*, similarly report the results of arithmetic and logical operations.

Figure 9.2
Flags register of the 8086



Other flags control modes of operation of the 8086. The *direction flag* controls the direction in which the string instructions move, and the *interrupt flag* controls whether external hardware, such as a keyboard or modem, is allowed to halt the current code temporarily so that urgent needs can be serviced. The *trap flag* is used only by software that debugs other software.

The flags register isn't usually modified or read directly. Instead, the flags register is generally controlled through special assembler instructions (such as **CLD**, **STI**, and **CMC**) and through arithmetic and logical instructions that modify certain flags. Likewise, the contents of certain bits of the flags register affect the operation of instructions such as **JZ**, **RCR**, and **MOVS**. The flags register is not really used as a storage location, but rather holds the status and control data for the 8086.

Memory segmentation

The Intel 8086 microprocessor has a *segmented memory architecture*. It has a total address space of 1 MB, but it is designed to directly address only 64K of memory at a time. A 64K chunk of memory is known as a segment; hence the phrase, "segmented memory architecture."

- The 8086 keeps track of four different segments: *code*, *data*, *stack*, and *extra*. The code segment is where the machine instructions are; the data segment, where information is; the stack is, of course, the stack; and the extra segment is also used for extra data.
- The 8086 has four 16-bit segment registers (one for each segment) named CS, DS, SS, and ES; these point to the code, data, stack, and extra segments, respectively.

- ▣ A segment can be located anywhere in memory. In DOS real mode, at least, almost anywhere. For reasons that will become clear as you read on, a segment must start on an address that's evenly divisible by 16 (in decimal).

Address calculation

This whole section is applicable only to real mode under DOS. You can safely ignore it for Windows development.

A complete address on the 8086 is composed of two 16-bit values: the segment address and the offset. Suppose the data segment address—the value in the DS register—is 2F84 (base 16), and you want to calculate the actual address of some data that has an offset of 0532 (base 16) from the start of the data segment; how is that done?

Address calculation is done as follows: Shift the value of the segment register 4 bits to the left (equivalent to one hex digit), then add in the offset.

The resulting 20-bit value is the actual address of the data, as illustrated here:

DS register (shifted):	0010 1111 1000 0100 0000	=	2F840
Offset:	0000 0101 0011 0010	=	00532
Address:	0010 1111 1101 0111 0010	=	2FD72

A chunk of 16 bytes is known as a paragraph, so you could say that a segment always starts on a paragraph boundary.

The starting address of a segment is always a 20-bit number, but a segment register only holds 16 bits—so the bottom 4 bits are always assumed to be all zeros. This means—as we said—that segments can only start every 16 bytes through memory, at an address where the last 4 bits (or last hex digit) are zero. So, if the DS register is holding a value of 2F84, then the data segment actually starts at address 2F840.

The standard notation for an address takes the form *segment:offset*; for example, the previous address would be written as 2F84:0532. Note that since offsets can overlap, a given segment:offset pair is not unique; the following addresses all refer to the same memory location:

```
0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003
```

Segments can overlap (but don't have to). For example, all four segments could start at the same address, which means that your

entire program would take up no more than 64K—but that’s all the space you’d have for your code, your data, and your stack.

Pointers

Although you can declare a pointer or function to be a specific type regardless of the model used, by default the type of memory model you choose determines the default type of pointers used for code and data. Pointers come in four flavors: *near* (16 bits), *far* (32 bits), *huge* (also 32 bits), and *segment* (16 bits).

Near pointers A near pointer (16-bits) relies on one of the segment registers to finish calculating its address; for example, a pointer to a function would add its 16-bit value to the left-shifted contents of the code segment (CS) register. In a similar fashion, a near data pointer contains an offset to the data segment (DS) register. Near pointers are easy to manipulate, since any arithmetic (such as addition) can be done without worrying about the segment.

Far pointers A far pointer (32-bits) contains not only the offset within the segment, but also the segment address (as another 16-bit value), which is then left-shifted and added to the offset. By using far pointers, you can have multiple code segments; that, in turn, allows you to have programs larger than 64K. You can also address more than 64K of data.

When you use far pointers for data, you need to be aware of some potential problems in pointer manipulation. As explained in the section on address calculation, you can have many different segment:offset pairs refer to the same address. For example, the far pointers 0000:0120, 0010:0020, and 0012:0000 all resolve to the same 20-bit address. However, if you had three different far pointer variables—*a*, *b*, and *c*—containing those three values respectively, then all the following expressions would be *false*:

```
if (a == b) . . .
if (b == c) . . .
if (a == c) . . .
```

A related problem occurs when you want to compare far pointers using the `>`, `>=`, `<`, and `<=` operators. In those cases, only the offset (as an **unsigned**) is used for comparison purposes; given that *a*, *b*, and *c* still have the values previously listed, the following expressions would all be *true*:

```
if (a > b) . . .
if (b > c) . . .
if (a > c) . . .
```

The equals (==) and not-equal (!=) operators use the 32-bit value as an **unsigned long** (not as the full memory address). The comparison operators (<=, >=, <, and >) use just the offset.

The == and != operators need all 32 bits, so the computer can compare to the NULL pointer (0000:0000). If you used only the offset value for equality checking, any pointer with 0000 offset would be equal to the NULL pointer, which is not what you want.

Important! If you add values to a far pointer, only the offset is changed. If you add enough to cause the offset to exceed FFFF (its maximum possible value), the pointer just wraps around back to the beginning of the segment. For example, if you add 1 to 5031:FFFF, the result would be 5031:0000 (not 6031:0000). Likewise, if you subtract 1 from 5031:0000, you would get 5031:FFFF (not 5030:000F).

If you want to do pointer comparisons, it's safest to use either near pointers—which all use the same segment address—or huge pointers, described next.

Huge pointers Huge pointers are also 32 bits long. Like far pointers, they contain both a segment address and an offset. Unlike far pointers, they are *normalized* to avoid the problems associated with far pointers.

What is a normalized pointer? It is a 32-bit pointer which has as much of its value in the segment address as possible. Since a segment can start every 16 bytes (10 in base 16), this means that the offset will only have a value from 0 to 15 (0 to F in base 16).

To normalize a pointer, convert it to its 20-bit address, then use the right 4 bits for your offset and the left 16 bits for your segment address. For example, given the pointer 2F84:0532, you would convert that to the absolute address 2FD72, which you would then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

0000:0123	0012:0003
0040:0056	0045:0006
500D:9407	594D:0007
7418:D03F	811B:000F

There are three reasons why it is important to always keep huge pointers normalized.

1. For any given memory address there is only one possible huge address—segment:offset pair. That means that the `==` and `!=` operators return correct answers for any huge pointers.
2. In addition, the `>`, `>=`, `<`, and `<=` operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results of these comparisons will be correct also.
3. Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size. This ensures that, for example, if you have a huge array of **structs** that's larger than 64K, indexing into the array and selecting a **struct** field will always work with structs of any size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

The six memory models



Use this model when memory is at an absolute premium.

This is a good size for average applications.

Best for large programs without much data in memory.

Borland C++ gives you six memory models: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. (See Chapter 8, “Building a Windows application,” in this book for information on choosing a memory model for Windows modules.) Here’s a brief summary of each:

Tiny. As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and stack. Near pointers are always used. Tiny model programs can be converted to .COM format by linking with the `/t` option.

Small. The code and data segments are different and don’t overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used.

Medium. Far pointers are used for code, but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1 MB.

Best if code is small but needs to address a lot of data.

Large and huge are needed only for very large applications.

Compact. The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1 MB range.

Large. Far pointers are used for both code and data, giving both a 1 MB range.

Huge. Far pointers are used for both code and data. Borland C++ normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

Figures 9.3 through 9.8 show how memory in the 8086 is apportioned for the Borland C++ memory models. To select these memory models, you can either use menu selections from the IDE, or you can type options invoking the command-line compiler version of Borland C++.

Figure 9.3
Tiny model memory segmentation

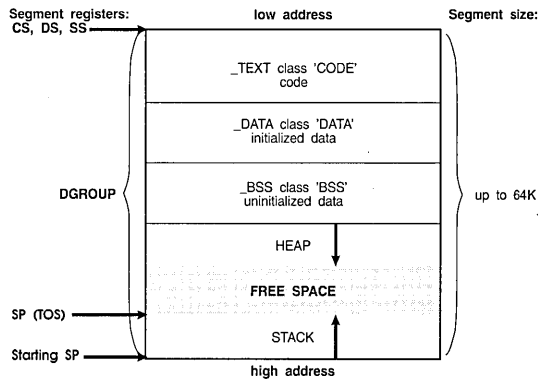


Figure 9.4
Small model memory segmentation

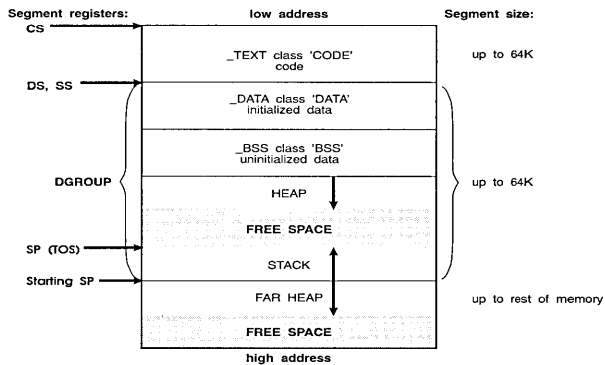
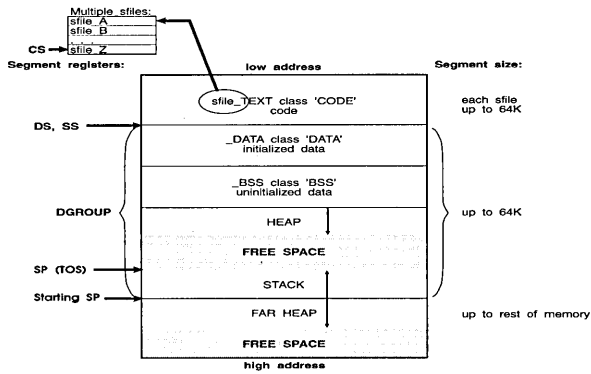


Figure 9.5
Medium model memory
segmentation



CS points to only one sfile at a time

Figure 9.6
Compact model memory
segmentation

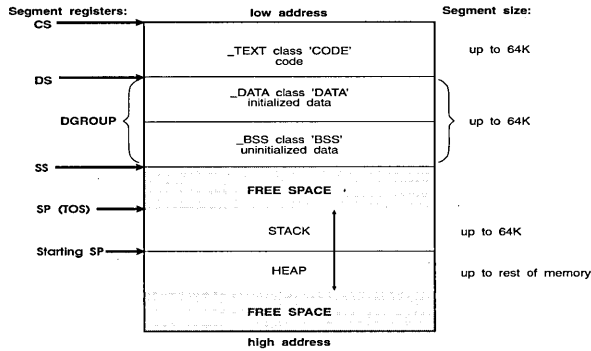


Figure 9.7
Large model memory
segmentation

CS points to only one sfile at a time

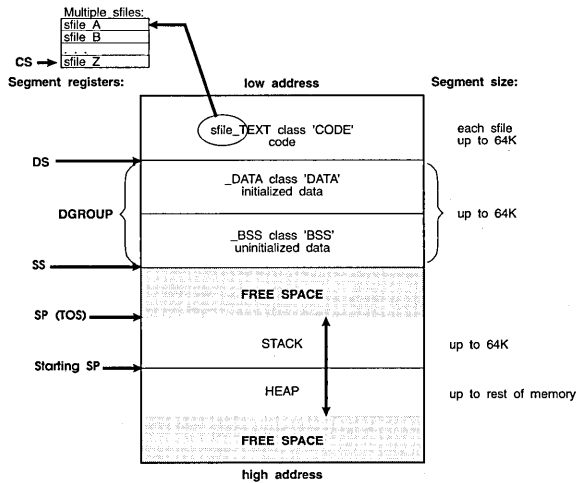


Figure 9.8
Huge model memory
segmentation

CS and DS point to only one
sfile at a time

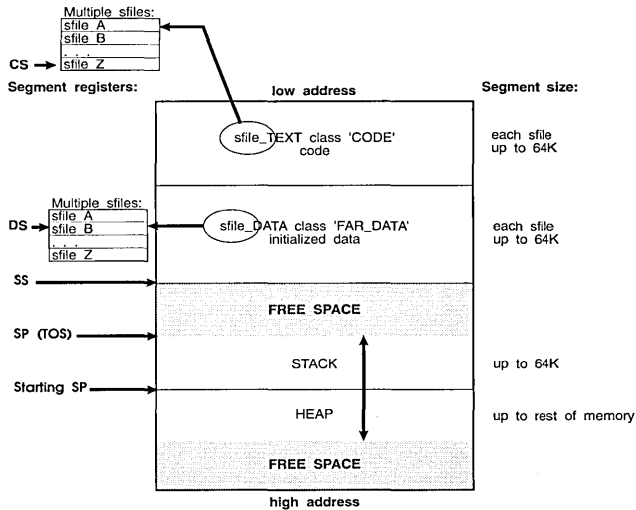


Table 9.1 summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (16 MB); these groups correspond to the rows and columns in Table 9.1.

Table 9.1
Memory models

The models tiny, small, and compact are small code models because, by default, code pointers are near; likewise, compact, large, and huge are large data models because, by default, data pointers are far.

Data size	Code size	
	64K	16 MB
64K	Tiny (data, code overlap; total size = 64K)	Medium (small data, large code)
16 MB	Compact (large data, small code)	Huge (same as large but static data > 64K)

Important!

When you compile a module (a given source file with some number of routines in it), the resulting code for that module cannot be greater than 64K, since it must all fit inside of one code segment. This is true even if you're using one of the larger code

models (medium, large, or huge). If your module is too big to fit into one (64K) code segment, you must break it up into different source code files, compile each file separately, then link them together. Similarly, even though the huge model permits static data to total more than 64K, it still must be less than 64K in *each* module.

Mixed-model programming: Addressing modifiers

Borland C++ introduces eight new keywords not found in standard ANSI C (**near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_ss**, and **_seg**) that can be used as modifiers to pointers (and in some cases, to functions), with certain limitations and warnings.

In Borland C++, you can modify the declarations of pointers, objects, and functions with the keywords **near**, **far**, or **huge**. We explained **near**, **far**, and **huge** data pointers earlier in this chapter. You can declare far objects using the **far** keyword. **near** functions are invoked with near calls and exit with near returns. Similarly, **far** functions are called **far** and do far returns. **huge** functions are like **far** functions, except that **huge** functions set DS to a new value, while **far** functions do not.

There are also four special **near** data pointers: **_cs**, **_ds**, **_es**, and **_ss**. These are 16-bit pointers that are specifically associated with the corresponding segment register. For example, if you were to declare a pointer to be

```
char _ss *p;
```

then *p* would contain a 16-bit offset into the stack segment.

Functions and pointers within a given program default to near or far, depending on the memory model you select. If the function or pointer is near, it is automatically associated with either the CS or DS register.

The next table shows just how this works. Note that the size of the pointer corresponds to whether it is working within a 64K memory limit (near, within a segment) or inside the general 1 MB memory space (far, has its own segment address).

Table 9.2
Pointer results

Memory model	Function pointers	Data pointers
Tiny	near, _cs	near, _ds
Small	near, _cs	near, _ds
Medium	far	near, _ds
Compact	near, _cs	far
Large	far	far
Huge	far	far

Segment pointers

Use `_seg` in segment pointer type declarators. The resulting pointers are 16-bit segment pointers. The syntax for `_seg` is:

```
datatype _seg *identifier;
```

For example,

```
int _seg *name;
```

Any indirection through *identifier* has an assumed offset of 0. In arithmetic involving segment pointers the following rules hold true:

1. You can't use the `++`, `--`, `+=`, or `-=` operators with segment pointers.
2. You cannot subtract one segment pointer from another.
3. When adding a near pointer to a segment pointer, the result is a far pointer that is formed by using the segment from the segment pointer and the offset from the near pointer. Therefore, the two pointers must either point to the same type, or one must be a pointer to void. There is no multiplication of the offset regardless of the type pointed to.
4. When a segment pointer is used in an indirection expression, it is also implicitly converted to a far pointer.
5. When adding or subtracting an integer operand to or from a segment pointer, the result is a far pointer, with the segment taken from the segment pointer and the offset found by multiplying the size of the object pointed to by the integer operand. The arithmetic is performed as if the integer were added to or subtracted from the far pointer.
6. Segment pointers can be assigned, initialized, passed into and out of functions, compared and so forth. (Segment pointers are compared as if their values were **unsigned** integers.) In other

words, other than the above restrictions, they are treated exactly like any other pointer.

Declaring far objects

You can declare far objects in Borland C++. For example,

```
int far x = 5;
int far z;
extern int far y = 4;
static long j;
```

The command-line compiler options **-zE**, **-zF**, and **-zH** (which can also be set using **#pragma option**) affect the far segment name, class, and group, respectively. When you change them with **#pragma option**, you can change them at any time and they apply to any ensuing far object declarations. Thus you could use the following sequence to create a far object in a specific segment:

```
#pragma option -zEmysegment -zHmygroup -zFmyclass
int far x;
#pragma option -zE* -zH* -zF*
```

This will put *x* in segment MYSEGMENT 'MYCLASS' in the group 'MYGROUP', then reset all of the far object items to the default values. Note that by using these options, several far objects can be forced into a single segment:

```
#pragma option -zEcombined -zFmyclass
int far x;
double far y;
#pragma option -zE* -zF*
```

Both *x* and *y* will appear in the segment COMBINED 'MYCLASS' with no group.

Declaring functions to be near or far

On occasion, you'll want (or need) to override the default function type of your memory model shown in Table 9.1 (page 349).

For example, suppose you're using the large memory model, but you have a recursive (self-calling) function in your program, like this:

```
double power(double x,int exp)
{
    if (exp <= 0)
        return(1);
```

```

        else
            return(x * power(x, exp-1));
    }

```

Every time **power** calls itself, it has to do a far call, which uses more stack space and clock cycles. By declaring **power** as **near**, you eliminate some of the overhead by forcing all calls to that function to be near:

```
double near power(double x,int exp)
```

This guarantees that **power** is callable only within the code segment in which it was compiled, and that all calls to it are near calls.

This means that if you are using a large code model (medium, large, or huge), you can only call **power** from within the module where it is defined. Other modules have their own code segment and thus cannot call **near** functions in different modules. Furthermore, a near function must be either defined or declared before the first time it is used, or the compiler won't know it needs to generate a near call.

Conversely, declaring a function to be far means that a far return is generated. In the small code models, the far function must be declared or defined before its first use to ensure it is invoked with a far call.

Look back at the **power** example. It is wise to also declare **power** as static, since it should only be called from within the current module. That way, being a static, its name will not be available to any functions outside the module.

Declaring pointers to be near, far, or huge

You've seen why you might want to declare functions to be of a different model than the rest of the program. Why might you want to do the same thing for pointers? For the same reasons given in the preceding section: either to avoid unnecessary overhead (declaring **near** when the default would be **far**) or to reference something outside of the default segment (declaring **far** or **huge** when the default would be **near**).

There are, of course, potential pitfalls in declaring functions and pointers to be of nondefault types. For example, say you have the following small model program:

```

void myputs(s)
char *s;
{
    int i;
    for (i = 0; s[i] != 0; i++) putc(s[i]);
}

main()
{
    char near *mystr;

    mystr = "Hello, world\n";
    myputs(mystr);
}

```

This program works fine. In fact, the **near** declaration on *mystr* is redundant, since all pointers, both code and data, will be near.

But what if you recompile this program using the compact (or large or huge) memory model? The pointer *mystr* in **main** is still near (it's still a 16-bit pointer). However, the pointer *s* in **myputs** is now far, since that's the default. This means that **myputs** will pull two words out of the stack in an effort to create a far pointer, and the address it ends up with will certainly not be that of *mystr*.

How do you avoid this problem? The solution is to define **myputs** in modern C style, like this:

```

void myputs(char *s)
{
    /* body of myputs */
}

```

*If you're going to explicitly declare pointers to be of type **far** or **near**, be sure to use function prototypes for any functions that might use them.*

Now when Borland C++ compiles your program, it knows that **myputs** expects a pointer to **char**; and since you're compiling under the large model, it knows that the pointer must be **far**. Because of that, Borland C++ will push the data segment (DS) register onto the stack along with the 16-bit value of *mystr*, forming a far pointer.

How about the reverse case: Arguments to **myputs** declared as **far** and compiling with a small data model? Again, without the function prototype, you will have problems, since **main** will push both the offset and the segment address onto the stack, but **myputs** will only expect the offset. With the prototype-style function definitions, though, **main** will only push the offset onto the stack.

Pointing to a given
segment:offset address

How do you make a far pointer point to a given memory location (a specific segment:offset address)? You can use the macro **MK_FP**, which takes a segment and an offset and returns a far pointer. For example,

```
MK_FP(segment_value, offset_value)
```

Given a **far** pointer, *fp*, you can get the segment component with **FP_SEG(fp)** and the offset component with **FP_OFF(fp)**. For more information about these three Borland C++ library routines, refer to the *Library Reference*.

Using library files

Borland C++ offers a version of the standard library routines for each of the six memory models. Borland C++ is smart enough to link in the appropriate libraries in the proper order, depending on which model you've selected. However, if you're using the Borland C++ linker, TLINK, directly (as a standalone linker), you need to specify which libraries to use. See Chapter 4, "TLINK: The Turbo linker" in the *Tools and Utilities Guide* for details on how to do so.

Linking mixed modules

What if you compiled one module using the small memory model, and another module using the large model, then wanted to link them together? What would happen?

The files would link together fine, but the problems you would encounter would be similar to those described in the earlier section, "Declaring functions to be near or far." If a function in the small module called a function in the large module, it would do so with a near call, which would probably be disastrous. Furthermore, you could face the same problems with pointers as described in the earlier section, "Declaring pointers to be near, far, or huge," since a function in the small module would expect to pass and receive **near** pointers, while a function in the large module would expect **far** pointers.

The solution, again, is to use function prototypes. Suppose that you put **myputs** into its own module and compile it with the large memory model. Then create a header file called **myputs.h** (or

some other name with a .h extension), which would have the following function prototype in it:

```
void far myputs(char far *s);
```

Now, if you put **main** into its own module (called MYMAIN.C), set things up like this:

```
#include <stdio.h>
#include "myputs.h"

main()
{
    char near *mystr;

    mystr = "Hello, world\n";
    myputs(mystr);
}
```

When you compile this program, Borland C++ reads in the function prototype from myputs.h and sees that it is a **far** function that expects a **far** pointer. Because of that, it will generate the proper calling code, even if it's compiled using the small memory model.

What if, on top of all this, you need to link in library routines? Your best bet is to use one of the large model libraries and declare everything to be **far**. To do this, make a copy of each header file you would normally include (such as stdio.h), and rename the copy to something appropriate (such as fstdio.h).

Then edit each function prototype in the copy so that it is explicitly **far**, like this:

```
int far cdecl printf(char far * format, ...);
```

That way, not only will **far** calls be made to the routines, but the pointers passed will be **far** pointers as well. Modify your program so that it includes the new header file:

```
#include <fstdio.h>

main()
{
    char near *mystr;
    mystr = "Hello, world\n";
    printf(mystr);
}
```

Compile your program with the command-line compiler BCC then link it with TLINK, specifying a large model library, such as

CL.LIB. Mixing models is tricky, but it can be done; just be prepared for some difficult bugs if you do things wrong.

Overlays (VROOMM) for DOS

Overlays are only used in DOS; you can mark the code segments of a Windows application as discardable to decrease memory consumption. See Chapter 4, "TLINK: The Turbo linker" in the Tools and Utilities Guide.

Overlays are parts of a program's code that share a common memory area. Only the parts of the program that are required for a given function reside in memory at the same time.

Overlays can significantly reduce a program's total run-time memory requirements. With overlays, you can execute programs that are much larger than the total available memory, since only parts of the program reside in memory at any given time.

How overlays work

Borland C++'s overlay manager (called VROOMM for Virtual Run-time Object-Oriented Memory Manager) is highly sophisticated; it does much of the work for you. In a conventional overlay system, modules are grouped together into a base and a set of overlay units. Routines in a given overlay unit may call other routines in the same unit and routines in the base, but not routines in other units. The overlay units are overlaid against each other; that is, only one overlay unit may be in memory at a time, and they each occupy the same physical memory. The total amount of memory needed to run the program is the size of the base plus the size of the largest overlay.

This conventional scheme is quite inflexible. It requires complete understanding of the possible calling dependencies in the program, and requires you to have the overlays grouped accordingly. It may be impossible to break your program into overlays if you can't split it into separable calling dependencies.

VROOMM's scheme is quite different. It provides *dynamic segment swapping*. The basic swapping unit is the segment. A segment can be one or more modules. More importantly, any segment can call *any other segment*.

Memory is divided into an area for the base plus a swap area. Whenever a function is called in a segment that is neither in the base nor in the swap area, the segment containing the called function is brought into the swap area, possibly displacing other segments. This is a powerful approach—it is like software virtual

memory. You no longer have to break your code into static, distinct, overlay units. You just let it run!

What happens when a segment needs to be brought into the swap area? If there is room for the segment, execution just continues. If there is not, then one or more segments in the swap area must be thrown out to make room. How to decide which segment to throw out? The actual algorithm is quite sophisticated. A simplified version: If there is an inactive segment, choose it for removal. Inactive segments are those without executing functions. Otherwise, pick an active segment and toss it out. Keep tossing out segments until there is enough room available. This technique is called *dynamic swapping*.

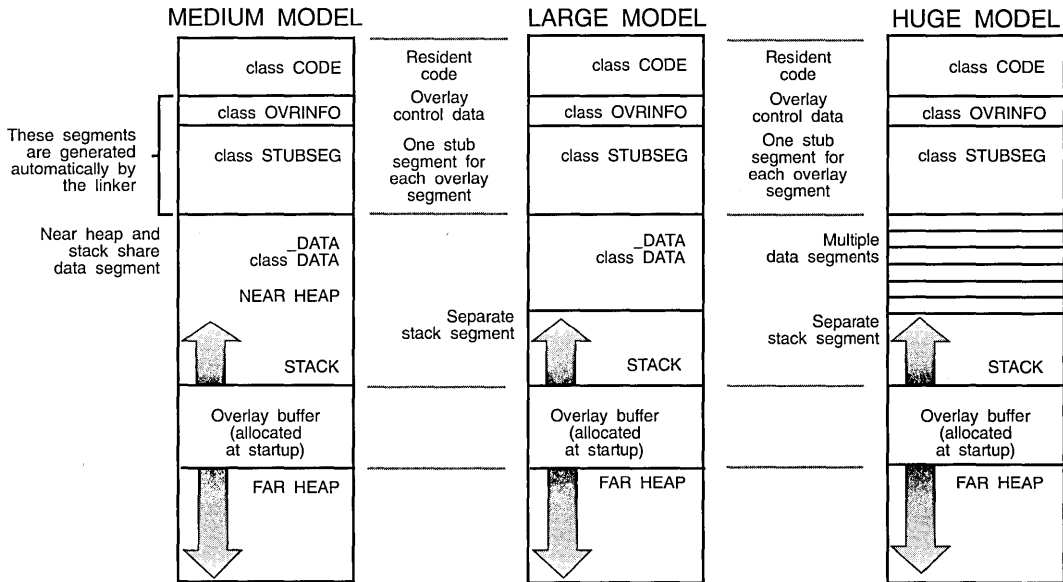
The more memory you provide for the swap area, the better the program performs. The swap area acts like a cache; the bigger the cache, the faster the program runs. The best setting for the size of the swap area is the size of the program's *working set*.

Once an overlay is loaded into memory, it is placed in the overlay buffer, which resides in memory between the stack segment and the far heap. By default, the size of the overlay buffer is estimated and set at startup, but you can change it using the global variable `_ovrbuffer` (see page 362). If enough memory isn't available, an error message will be displayed by DOS ("Program too big to fit in memory") or by the C startup code ("Not enough memory to run program").

One very important option of the overlay manager is the ability to swap the modules to expanded or extended memory when they are discarded from the overlay buffer. Next time the module is needed, the overlay manager can copy it from where the module was swapped to instead of reading from the file. This makes it much faster.

When using overlays, memory is used as shown in the next figure.

Figure 9.9: Memory maps for overlays



Getting the best out of Borland C++ overlays

See page 362 for more information on setting the size of the overlay buffer.

To get the best out of Borland C++ overlays,

- Minimize resident code (resident run-time library, interrupt handlers, and device drivers is a good starting point).
- Set overlay buffer size to be a comfortable working set (start with 128K and adjust up and down to see the speed/size tradeoff).
- Think versatility and variety: Take advantage of the overlay system to provide support for special cases, interactive help, and other end-user benefits you could not consider before.

Requirements

In order to create overlays, you'll need to remember a few simple rules,

- The smallest part of a program that can be made into an overlay is a segment.
- Overlaid applications must use the medium, large, or huge programming models; the tiny, small, and compact models are not supported.

- Normal segment merging rules govern overlaid segments. That is, several .OBJ modules can contribute to the same overlaid segment.

The link-time generation of overlays is completely separated from the run-time overlay management; the linker does *not* automatically include code to manage the overlays. In fact, from the linker's point of view, the overlay manager is just another piece of code that gets linked in. The only assumption the linker makes is that the overlay manager takes over an interrupt vector (typically INT 3FH) through which all dynamic loading is controlled.

Using overlays

To overlay a program, all of its modules must be compiled with the `-Y` compiler option enabled. To make a particular module into an overlay, it needs to be compiled with the `-Yo` option. (`-Yo` automatically enables `-Y`.)

The `-Yo` option applies to all modules and libraries that follow it on the command line; you can disable it with `-Yo-`. These are the only command line options that are allowed to follow file names. For example, to overlay the module OVL.C but not the library GRAPHICS.LIB, either of the following command lines could be used:

```
BCC -ml -Yo ovl.c -Yo- graphics.lib
```

or

```
BCC -ml graphics.lib -Yo ovl.c
```

If TLINK is invoked explicitly to link the .EXE file, the `b` linker option must be specified on the linker command line or response file. See Chapter 4, "TLINK: The Turbo linker," in the *Tools and Utilities Guide* for details on how to use the `b` option.

Overlay example

Suppose that you want to overlay a program consisting of three modules: MAIN.C, O1.C, and O2.C. Only the modules O1.C and O2.C should be made into overlays. (MAIN.C contains time-critical routines and interrupt handlers, so it should stay resident.) Let's assume that the program uses the large memory model.

The following command accomplishes the task:

```
BCC -ml -Y main.c -Yo o1.c o2.c
```

The result will be an executable file MAIN.EXE, containing two overlays.

Overlaying in the IDE To overlay modules in the IDE, you must take the following steps:

1. Select Options | Application | DOS Overlay
2. In the project manager, use project item Local options to specify each module that needs to go into an overlay.

Selecting Options | Application | DOS Overlay will also select the following options automatically for you:

- Options | Compiler | Entry | Exit Code | DOS overlay
- Options Linker | Settings | Overlaid DOS Exe
- Project | Local Options | Overlay this module
- Options | Compiler | Code generation | Medium
- Options | Compiler | Code generation | Assume SS Equals DS | Default for memory model
- Options | Linker | Libraries | Graphics library



If you are building an .EXE file containing overlays, compile all modules after selecting DOS Overlay from the Options | Application dialog box.



No module going into an overlay should ever change the default Code Class name. The IDE lets you change the set of modules residing in overlays without having to worry about recompiling. This can only be accomplished (with current .OBJ information) if overlays keep default code class names.

Overlaid programs

This section discusses issues vital to well-behaved overlaid applications.

The far call requirement

Use a large code model (medium, large, or huge) when you want to compile an overlay module. At any call to an overlaid function in another module, you *must* guarantee that all currently active functions are far.

You *must* compile all overlaid modules with the `-Y` option, which makes the compiler generate code that can be overlaid.

Important! Failing to observe the far call requirement in an overlaid program will cause unpredictable and possibly catastrophic results when the program is executed.

Buffer size The default overlay buffer size is twice the size of the largest overlay. This is adequate for some applications. But imagine that a particular function of a program is implemented through many modules, each of which is overlaid. If the total size of those modules is larger than the overlay buffer, a substantial amount of swapping will occur if the modules make frequent calls to each other.

The solution is to increase the size of the overlay buffer so that enough memory is available at any given time to contain all overlays that make frequent calls to each other. You can do this by setting the `_ovrbuffer` global variable to the required size in paragraphs. For example, to set the overlay buffer to 128K, include the following statement in your code:

```
unsigned _ovrbuffer = 0x2000;
```

There is no general formula for determining the ideal overlay buffer size. Borland's Turbo Profiler can help provide a suitable value.

What not to overlay Don't overlay modules that contain interrupt handlers, or small and time-critical routines. Due to the non-reentrant nature of the DOS operating system, modules that may be called by interrupt functions should not be overlaid.

Borland C++'s overlay manager fully supports passing overlaid functions as arguments, assigning and initializing function pointer variables with addresses of overlaid functions, and calling overlaid routines via function pointers.

Debugging overlays Most debuggers have very limited overlay debugging capabilities, if any at all. Not so with Borland C++'s integrated debugger and Turbo Debugger, the standalone debugger. Both debuggers fully support single-stepping and breakpoints in overlays in a manner completely transparent to you. By using overlays, you can easily engineer and debug huge applications—all from inside the IDE or by using Turbo Debugger.

External routines in overlays

Like normal C functions, **external** assembly language routines must observe certain programming rules to work correctly with the overlay manager.

If an assembly language routine makes calls to *any* overlaid functions, the assembly language routine *must* be declared FAR, and it *must* set up a stack frame using the BP register. For example, assuming that *OtherFunc* is an overlaid function in another module, and that the assembly language routine *ExternFunc* calls it, then *ExternFunc* must be FAR and set up a stack frame, as shown:

```
ExternFunc    PROC    FAR
              push    bp                ;Save BP
              mov     bp,sp            ;Set up stack frame
              sub     sp,LocalSize     ;Allocate local variables
              :
              call   OtherFunc        ;Call another overlaid module
              :
              mov     sp,bp            ;Dispose local variables
              pop     bp                ;Restore BP
              RET                     ;Return
ExternFunc    ENDP
```

where *LocalSize* is the size of the local variables. If *LocalSize* is zero, you can omit the two lines to allocate and dispose local variables, but you must not omit setting up the BP stack frame even if you have no arguments or variables on the stack.

These requirements are the same if *ExternFunc* makes *indirect* references to overlaid functions. For example, if *OtherFunc* makes calls to overlaid functions, but is not itself overlaid, *ExternFunc* must be FAR and still has to set up a stack frame.

In the case where an assembly language routine doesn't make any direct or indirect references to overlaid functions, there are no special requirements; the assembly language routine can be declared NEAR. It does not have to set up a stack frame.

Overlaid assembly language routines should *not* create variables in the code segment, since any modifications made to an overlaid code segment are lost when the overlay is disposed. Likewise, pointers to objects based in an overlaid code segment cannot be expected to remain valid across calls to other overlays, since the overlay manager freely moves around and disposes overlaid code segments.

Swapping

If you have expanded or extended memory available, you can tell the overlay manager to use it for swapping. If you do so, when the overlay manager has to discard a module from the overlay buffer (because it should load a new module and the buffer is full), it can store the discarded module in this memory. Any later loading of this module is reduced to in-memory transfer, which is significantly faster than reading from a disk file.

In both cases there are two possibilities: The overlay manager can either detect the presence of expanded or extended memory and can take it over by itself, or it can use an already detected and allocated portion of memory. For extended memory, the detection of the memory use is not always successful because of the many different cache and RAM disk programs that can take over extended memory without any mark. To avoid this problem, you can tell the overlay manager the starting address of the extended memory and how much of it is safe to use.

Expanded memory

The **_OvrInitEms** function initializes expanded memory swapping. Here's its prototype:

_OvrInitEms and **_OvrInitExt**
are defined in *dos.h*.

```
extern int far _OvrInitEms
(
    unsigned emsHandle,
    unsigned emsFirst,
    unsigned emsPages
);
```

If the *emsHandle* parameter is zero, the overlay manager checks for the presence of expanded memory and allocates the amount (if it can) that can contain all of the overlays minus the size of the overlay buffer. Otherwise, *emsHandle* should be a legal EMS handle, *emsFirst* is the first usable EMS page, and *emsPages* is the number of pages usable by the overlay manager. This function returns 0 if expanded memory is available.

Extended memory

The **_OvrInitExt** function initializes extended memory swapping. Here's its prototype:

```
extern int far _OvrInitExt
(
    unsigned long extStart,
```

```
        unsigned long extLength
    );
```

If the *extStart* parameter is zero, the overlay manager checks for extended memory. If it can, the overlay manager uses the amount of free memory that can contain all of the overlays minus the size of the overlay buffer. Otherwise, *extStart* is the start of the usable extended memory, with *extLength* bytes usable by the overlay manager. If *extlength* is zero, the overlay manager will use all available extended memory above *extStart*. This function returns 0 if extended memory is available. **_OvrInitExt** is defined in `dos.h`.

Important!

The use of extended memory is not standardized. Though the overlay manager tries every known method to find out the amount of extended memory which is already used, use this function carefully. For example, if you have a 2 MB hard disk cache program installed (that uses extended memory), you could use the following call to let the overlay manager use the remaining extended memory:

```
if (_OvrInitExt (1024L * (2048 + 1024), 0L))
    puts ("No extended memory available for overlay swapping");
```


Math

This chapter covers the floating-point options and explains how to use complex math.

Floating-point options

There are two types of numbers you work with in C: integer (**int**, **short**, **long**, and so on) and floating point (**float**, **double**, and **long double**). Your computer's processor is set up to easily handle integer values, but it takes more time and effort to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087, the 80287, and the 80387. We refer to this entire family of math coprocessors as the 80x87, or "the coprocessor."

If you have an 80486 processor, the numeric coprocessor is probably already built in.

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly. If you use floating point a lot, you'll probably want a coprocessor. The CPU in your computer interfaces to the 80x87 via special hardware lines.

Emulating the 80x87 chip

The default Borland C++ code generation option is *emulation* (the `-f` command-line compiler option). This option is for programs that may or may not have floating point, and for machines that may or may not have an 80x87 math coprocessor.

With the emulation option, the compiler will generate code as if the 80x87 were present, but will also link in the emulation library (EMU.LIB). When the program runs, it will use the 80x87 if it is present; if no coprocessor is present at run time, it uses special software that *emulates* the 80x87.

Using 80x87 code

If your program is *only* going to run on machines with an 80x87 math coprocessor, you can save a small amount in your .EXE file size by omitting the 80x87 autodetection and emulation logic. Simply choose the 80x87 floating-point code generation option (the `-f87` command-line compiler option). Borland C++ will then link your programs with FP87.LIB instead of EMU.LIB.

No floating-point code

If there is no floating-point code in your program, you can save a small amount of link time by choosing None for the floating-point code generation option (the `-f-` command-line compiler option). Then Borland C++ will not link with EMU.LIB, FP87.LIB, or MATHx.LIB.

Fast floating-point option

Borland C++ has a fast floating-point option (the `-ff` command-line compiler option). It can be turned off with `-ff-` on the command line. Its purpose is to allow certain optimizations that are technically contrary to correct C semantics. For example,

```
double x;  
x = (float)(3.5*x);
```

To execute this correctly, *x* is multiplied by 3.5 to give a **double** that is truncated to **float** precision, then stored as a **double** in *x*. Under the fast floating-point option, the **long double** product is converted directly to a **double**. Since very few programs depend

on the loss of precision in passing to a narrower floating-point type, fast floating point is the default.

The 87 environment variable

If you build your program with 80x87 emulation, which is the default, your program will automatically check to see if an 80x87 is available, and will use it if it is.

There are some situations in which you might want to override this default autodetection behavior. For example, your own runtime system might have an 80x87, but you need to verify that your program will work as intended on systems without a coprocessor. Or your program may need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that a nonexistent 80x87 is available, or vice versa).

Borland C++ provides an option for overriding the start-up code's default autodetection logic; this option is the 87 environment variable.

You set the 87 environment variable at the DOS prompt with the SET command, like this:

```
C> SET 87=N
```

or like this:

```
C> SET 87=Y
```

Don't include spaces to either side of the =. Setting the 87 environment variable to N (for No) tells the start-up code that you do not want to use the 80x87, even though it might be present in the system.

Setting the 87 environment variable to Y (for Yes) means that the coprocessor is there, and you want the program to use it. Let the programmer beware!! If you set 87 = Y when, in fact, there is no 80x87 available on that system, your system will hang.

If the 87 environment variable has been defined (to any value) but you want to undefine it, enter the following at the DOS prompt:

```
C> SET 87=
```

Press *Enter* immediately after typing the equal sign.

Registers and the 80x87

There are a couple of points concerning registers that you should be aware of when using floating point.

1. In 80x87 emulation mode, register wraparound and certain other 80x87 peculiarities are not supported.
2. If you are mixing floating point with inline assembly, you may need to take special care when using 80x87 registers. You might need to pop and save the 80x87 registers before calling functions that use the coprocessor, unless you are sure that enough free registers exist.

Disabling floating-point exceptions

By default, Borland C++ programs abort if a floating-point overflow or divide by zero error occurs. You can mask these floating-point exceptions by a call to `_control87` in `main`, before any floating-point operations are performed. For example,

```
#include <float.h>
main() {
    _control87(MCW_EM,MCW_EM);
    ...
}
```

You can determine whether a floating-point exception occurred after the fact by calling `_status87` or `_clear87`. See the entries for these functions using 2 for details.

Certain math errors can also occur in library functions; for instance, if you try to take the square root of a negative number. The default behavior is to print an error message to the screen, and to return a NAN (an IEEE not-a-number). Use of the NAN will likely cause a floating-point exception later, which will abort the program if unmasked. If you don't want the message to be printed, insert the following version of `matherr` into your program.

```
#include <math.h>
int cdecl matherr(struct exception *e)
{
    return 1;          /* error has been handled */
}
```

Any other use of `matherr` to intercept math errors is not encouraged, as it is considered obsolete and may not be supported in future versions of Borland C++.

Using complex math

Complex numbers are numbers of the form $x + yi$, where x and y are real numbers, and i is the square root of -1 . Borland C++ has always had a type

```
struct complex
{
    double x, y;
};
```

defined in `math.h`. This type is convenient for holding complex numbers, as they can be considered a pair of real numbers. However, the limitations of C make arithmetic with complex numbers rather cumbersome. With the addition of C++, complex math is much simpler.

*See the description of class **complex** in the Library Reference for more information.*

To use complex numbers in C++, all you have to do is to include `complex.h`. In `complex.h`, all the following have been overloaded to handle complex numbers:

- all of the usual arithmetic operators
- the stream operators, `>>` and `<<`
- the usual math functions, such as **sqrt** and **log**

The complex library is invoked only if the argument is of type **complex**. Thus, to get the complex square root of -1 , use

```
sqrt(complex(-1))
```

and not

```
sqrt(-1)
```

As an example of the use of complex numbers, the following function computes a complex Fourier transform.

```
#include <complex.h>
// calculate the discrete Fourier transform of a[0], ..., a[n-1].
void Fourier(int n, complex a[], complex b[])
{
    int j, k;
    complex i(0,1); // square root of -1
```

```

for (j = 0; j < n; ++j)
{
    b[j] = 0;
    for (k = 0; k < n; ++k)
        b[j] += a[k] * exp(2*M_PI*j*k*i/n);
    b[j] /= sqrt(n);
}
}

```

Using BCD math

Borland C++, along with almost every other computer and compiler, does arithmetic on binary numbers (that is, base 2). This is sometimes confusing to people who are used to decimal (base 10) representations. Many numbers that are exactly representable in base 10, such as 0.01, can only be approximated in base 2.

Binary numbers are preferable for most applications, but in some situations the roundoff error involved in converting between base 2 and 10 is undesirable. The most common case is a financial or accounting application, where the pennies are supposed to add up. Consider the following program to add up 100 pennies and subtract a dollar:

```

#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n", x);

```

The correct answer is 0.0, but the computed answer is a small number close to 0.0. The computation magnifies the tiny roundoff error that occurs when converting 0.01 to base 2. Changing the type of *x* to **double** or **long double** reduces the error, but does not eliminate it.

To solve this problem, Borland C++ offers the C++ type **bcd**, which is declared in `bcd.h`. With **bcd**, the number 0.01 is represented exactly, and the **bcd** variable *x* will give an exact penny count.

```

#include <bcd.h>
int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;

```

```
x -= 1.0;
cout << "100*.01 - 1 = " << x << "\n";
```

Here are some facts to keep in mind about **bcd**.

- ▣ **bcd** does not eliminate all roundoff error: A computation like 1.0/3.0 will still have roundoff error.
- ▣ The usual math functions, such as **sqrt** and **log**, have been overloaded for **bcd** arguments.
- ▣ BCD numbers have about 17 decimal digits precision, and a range of about 1×10^{-125} to 1×10^{125} .

Converting BCD numbers

bcd is a defined type distinct from **float**, **double**, or **long double**; decimal arithmetic is only performed when at least one operand is of the type **bcd**.

Important!

The **bcd** member function **real** is available for converting a **bcd** number back to one the usual base 2 formats (**float**, **double**, or **long double**), though the conversion is not done automatically. **real** does the necessary conversion to **long double**, which can then be converted to other types using the usual C conversions. For example,

```
bcd a = 12.1;
```

can be printed using any of the following four lines of code:

```
double x = a; printf("a = %g", x);
printf("a = %Lg", real(a));
printf("a = %g", (double)real(a));
cout << "a = " << a;
```

Note that since **printf** does not do argument checking, the format specifier must have the *L* if the **long double** value **real(a)** is passed.

Number of decimal digits

You can specify how many decimal digits after the decimal point are to be carried in a conversion from a binary type to a **bcd**. The number of places is an optional second argument to the constructor **bcd**. For example, to convert \$1000.00/7 to a **bcd** variable rounded to the nearest penny, use

```
bcd a = bcd(1000.00/7, 2)
```

where 2 indicates two digits following the decimal point. Thus,

1000.00/7	=	142.85714...
bcd(1000.00/7, 2)	=	142.860
bcd(1000.00/7, 1)	=	142.900
bcd(1000.00/7, 0)	=	143.000
bcd(1000.00/7, -1)	=	140.000
bcd(1000.00/7, -2)	=	100.000

This method of rounding is specified by IEEE.

The number is rounded using banker's rounding, which means round to the nearest whole number, with ties being rounded to an even digit. For example,

bcd(12.335, 2)	=	12.34
bcd(12.345, 2)	=	12.34
bcd(12.355, 2)	=	12.36

Video functions

Borland C++ comes with a complete library of graphics functions, so you can produce onscreen charts and diagrams. This chapter briefly discusses video modes and windows, then explains how to program in text mode and in graphics mode.

Borland C++'s video functions are similar to corresponding routines in Turbo Pascal. If you are already familiar with controlling your PC's screen modes or creating and managing windows and viewports, you can skip to page 377.

Some words about video modes

Your PC has some type of video adapter. This can be a Monochrome Display Adapter (MDA) for text-only display, or it can be a graphics adapter, such as a Color/Graphics Adapter (CGA), an Enhanced Graphics Adapter (EGA), a Video Graphics Array adapter (VGA), or a Hercules Monochrome Graphics Adapter. Each adapter can operate in a variety of modes; the mode specifies whether the screen displays 80 or 40 columns (text mode only), the display resolution (graphics mode only), and the display type (color or black and white).

The screen's operating mode is defined when your program calls one of the mode-defining functions **textmode**, **initgraph**, or **setgraphmode**.

- In *text mode*, your PC's screen is divided into cells (80- or 40-columns wide by 25, 43, or 50 lines high). Each cell consists of an attribute and a character. The character is the displayed ASCII character; the attribute specifies *how* the character is displayed (its color, intensity, and so on). Borland C++ provides a full range of routines for manipulating the text screen, for writing text directly to the screen, and for controlling cell attributes.
- In *graphics mode*, your PC's screen is divided into pixels; each pixel displays a single dot onscreen. The number of pixels (the resolution) depends on the type of video adapter connected to your system and the mode that adapter is in. You can use functions from Borland C++'s graphics library to create graphic displays onscreen: You can draw lines and shapes, fill enclosed areas with patterns, and control the color of each pixel.

In text modes, the upper left corner of the screen is position (1,1), with x-coordinates increasing from left to right, and y-coordinates increasing from screen-top to screen-bottom. In graphics modes, the upper left corner is position (0,0), with the x- and y-coordinate values increasing in the same manner.

Some words about windows and viewports

Borland C++ provides functions for creating and managing windows on your screen in text mode (and viewports in graphics mode). If you are not familiar with windows and viewports, you should read this brief overview. Borland C++'s window- and viewport-management functions are explained in "Programming in text mode" and "Programming in graphics mode" later in this chapter.

What is a window?

A window is a rectangular area defined on your PC's video screen when it's in a text mode. When your program writes to the screen, its output is restricted to the active window. The rest of the screen (outside the window) remains untouched.

The default window is a full-screen text window. Your program can change this default full-screen text window to a text window smaller than the full screen (with a call to the **window** function). This function specifies the window's position in terms of screen coordinates.

What is a viewport?

In graphics mode, you can also define a rectangular area on your PC's video screen; this is a viewport. When your graphics program outputs drawings and so on, the viewport acts as the virtual screen. The rest of the screen (outside the viewport) remains untouched. You define a viewport in terms of screen coordinates with a call to the **setviewport** function.

Coordinates

Except for these window- and viewport-defining functions, all coordinates for text-mode and graphics-mode functions are given in window- or viewport-relative terms, not in absolute screen coordinates. The upper left corner of the text-mode window is the coordinate origin, referred to as (1,1); in graphics modes, the viewport coordinate origin is position (0,0).

Programming in text mode

This section briefly summarizes the text mode functions. For more information about these functions, see Chapter 2, "The run-time library," of the Library Reference.

In Borland C++, the direct console I/O package (**cprintf**, **cputs**, and so on) provides high-performance text output, window management, cursor positioning, and attribute control functions. These functions are all part of the standard Borland C++ libraries; they are prototyped in the header file `conio.h`.

The console I/O functions

Borland C++'s text-mode functions work in any of the six possible video text modes. The modes available on your system depend on the type of video adapter and monitor you have. You specify the current text mode with a call to **textmode**. We explain how to use this function later in this chapter and under the **textmode** entry in Chapter 2 in the *Library Reference*.

The text mode functions are divided into five separate groups:

These five text mode function groups are covered in the following sections.

- text output and manipulation
- window and mode control
- attribute control
- state query
- cursor shape

Text output and manipulation

Here's a quick summary of the text output and manipulation functions:

Writing and reading text:

cprintf	Sends formatted output to the screen.
cputs	Sends a string to the screen.
getche	Reads a character and echoes it to the screen.
putch	Sends a single character to the screen.

Manipulating text (and the cursor) onscreen:

clreol	Clears from the cursor to the end of the line.
clrscr	Clears the text window.
delline	Deletes the line where the cursor rests.
gotoxy	Positions the cursor.
insline	Inserts a blank line below the line where the cursor rests.
movetext	Copies text from one area onscreen to another.

Moving blocks of text into and out of memory:

gettext	Copies text from an area onscreen to memory.
puttext	Copies text from memory to an area onscreen.

Your screen-output programs will come up in a full-screen text window by default, so you can immediately write, read, and manipulate text without any preliminary mode-setting. You write text to the screen with the direct console output functions **cprintf**, **cputs**, and **putch**, and echo input with the function **getche**. Text wrapping is controlled by the global variable `_wscroll`. If `_wscroll` is 1, text wraps onto the next line, scrolling as necessary. If `_wscroll` is 0, text wraps onto the same line, and there is no scrolling. `_wscroll` is 1 by default.

Once your text is on the screen, you can erase the active window with **clrscr**, erase part of a line with **clreol**, delete a whole line with **delline**, and insert a blank line with **insline**. The latter three functions operate relative to the cursor position; you move the cursor to a specified location with **gotoxy**. You can also copy a whole block of text from one rectangular location in the window to another with **movetext**.

You can capture a rectangle of onscreen text to memory with **gettext**, and put that text back on the screen (anywhere you want) with **puttext**.

Window and mode control

There are two window- and mode-control functions:

- textmode** Sets the screen to a text mode.
- window** Defines a text-mode window.

You can set your screen to any of several video text modes with **textmode** (depending on your system's monitor and adapter). This initializes the screen as a full-screen text window, in the particular mode specified, and clears any residual images or text.

When your screen is in a text mode, you can output to the full screen, or you can set aside a *portion* of the screen—a window—to which your program's output is confined. To create a text window, you call **window**, specifying the onscreen area it will occupy.

Attribute control

Here's a quick summary of the text-mode attribute control functions:

Setting foreground and background:

- textattr** Sets the foreground and background colors (attributes) at the same time.
- textbackground** Sets the background color (attribute).
- textcolor** Sets the foreground color (attribute).

Modifying intensity:

- highvideo** Sets text to high intensity.
- lowvideo** Sets text to low intensity.
- normvideo** Sets text to original intensity.

The attribute control functions set the current attribute, which is represented by an 8-bit value: The four lowest bits represent the foreground color, the next three bits give the background color, and the high bit is the "blink enable" bit.

Subsequent text is displayed in the current attribute. With the attribute control functions, you can set the background and foreground (character) colors separately (with **textbackground** and **textcolor**) or combine the color specifications in a single call to **textattr**. You can also specify that the character (the foreground) will blink. Most color monitors in color modes will display the true colors. Non-color monitors may convert some or all of the attributes to various monochromatic shades or other visual effects, such as bold, underscore, reverse video, and so on.

You can direct your system to map the high-intensity foreground colors to low-intensity colors with **lowvideo** (which turns off the high-intensity bit for the characters). Or you can map the low-intensity colors to high intensity with **highvideo** (which turns on

the character high-intensity bit). When you're through playing around with the character intensities, you can restore the settings to their original values with **normvideo**.

State query Here's a quick summary of the state-query functions:

gettextinfo	Fills in a text_info structure with information about the current text window.
wherex	Gives the x-coordinate of the cell containing the cursor.
wherey	Gives the y-coordinate of the cell containing the cursor.

Borland C++'s console I/O functions include some designed for *state queries*. With these functions, you can retrieve information about your text-mode window and the current cursor position within the window.

The **gettextinfo** function fills a **text_info** structure (defined in `conio.h`) with several details about the text window, including:

- the current video mode
- the window's position in absolute screen coordinates
- the window's dimensions
- the current foreground and background colors
- the cursor's current position

Sometimes you might need only a few of these details. Instead of retrieving all text window information, **wherex** and **wherey** return just the cursor's (window-relative) position.

Cursor shape The function **_setcursortype** enables you to change the appearance of your cursor. The values are `_NOCURS`, which turns off the cursor; `_SOLIDCURSOR`, which gives you a solid block (large) cursor; and `_NORMALCURSOR`, which gives you the normal underscore cursor.

Text windows

The default text window is full screen; you can change this to a smaller text window with a call to the **window** function. Text windows can contain up to 50 lines and up to 40 or 80 columns.

The coordinate origin (point where the numbers start) of a Borland C++ text window is the upper left corner of the window.

The coordinates of the window's upper left corner are (1,1); the coordinates of the bottom right corner of a full-screen 80-column, 25-line text window are (80,25).

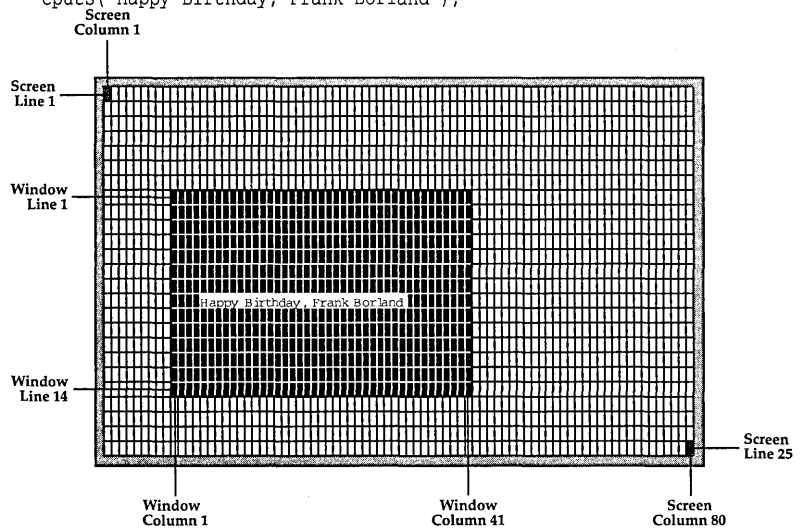
An example Suppose your 100% PC-compatible system is in 80-column text mode, and you want to create a window. The upper left corner of the window will be at screen coordinates (10, 8), and the lower right corner of the window will be at screen coordinates (50, 21). To do this, you call the **window** function, like this:

```
window(10, 8, 50, 21);
```

Now that you've created the text-mode window, you want to move the cursor to the *window* position (5, 8) and write some text in it, so you decide to use **gotoxy** and **cputs**. The following figure illustrates the code.

```
gotoxy(5, 8);  
cputs("Happy Birthday, Frank Borland");
```

Figure 11.1
A window in 80x25 text mode



The *text_modes* type

You can put your monitor into one of seven PC text modes with a call to the **textmode** function. The enumeration type *text_modes*, defined in *conio.h*, enables you to use symbolic names for the *mode* argument to the **textmode** function, instead of "raw" mode numbers. However, with symbolic constants, you must put


```
#include <conio.h>
```

in your source code.

The numeric and symbolic values defined by *text_modes* are as follows:

Symbolic constant	Numeric value	Video text mode
LASTMODE	-1	Previous text mode enabled
BW40	0	Black and white, 40 columns
C40	1	16-color, 40 columns
BW80	2	Black and white, 80 columns
C80	3	16-color, 80 columns
MONO	7	Monochrome, 80 columns
C4350	64	EGA, 80x43; VGA, 80x50 lines

For example, the following calls to **textmode** put your color monitor in the indicated operating mode:

```
textmode(0)    Black and white, 40 column  
textmode(BW80) Black and white, 80 column  
textmode(C40)  16-color, 40 column  
textmode(3)    16-color, 80 column  
textmode(7)    Monochrome, 80 columns  
textmode(C4350) EGA, 80x43; VGA, 80x50 lines
```

Use **settextinfo** to determine the number of rows in the screen after calling **textmode** in the mode C4350.

Text colors

For a detailed description of how cell attributes are laid out, refer to the **textattr** entry in Chapter 2 of the *Library Reference*.

When a character occupies a cell, the color of the character is the *foreground*; the color of the cell's remaining area is the *background*. Color monitors with color video adapters can display up to 16 different colors; monochrome monitors substitute different visual attributes (highlighted, underscored, reverse video, and so on) for the colors.

Symbolic constant	Numeric value	Foreground or background?
BLACK	0	Both
BLUE	1	Both
GREEN	2	Both
CYAN	3	Both
RED	4	Both
MAGENTA	5	Both
BROWN	6	Both
LIGHTGRAY	7	Both
DARKGRAY	8	Foreground only
LIGHTBLUE	9	Foreground only
LIGHTGREEN	10	Foreground only
LIGHTCYAN	11	Foreground only
LIGHTRED	12	Foreground only
LIGHTMAGENTA	13	Foreground only
YELLOW	14	Foreground only
WHITE	15	Foreground only
BLINK	128	Foreground only

The include file `conio.h` defines symbolic names for the different colors. If you use the symbolic constants, you must include `conio.h` in your source code.

Table 11 lists these symbolic constants and their corresponding numeric values. Note that only the first eight colors are available for the foreground and background; the last eight (colors 8 through 15) are available for the foreground (the characters themselves) only.

You can add the symbolic constant `BLINK` (numeric value 128) to a foreground argument if you want the character to blink.

High-performance output

Borland C++'s console I/O package includes a variable called *directvideo*. This variable controls whether your program's console output goes directly to the video RAM (*directvideo* = 1) or goes via BIOS calls (*directvideo* = 0).

The default value is *directvideo* = 1 (console output goes directly to the video RAM). In general, going directly to video RAM gives very high performance (spelled f-a-s-t-e-r o-u-t-p-u-t), but doing so requires your computer to be 100% IBM PC-compatible: Your video hardware must be identical to IBM display adapters. Setting *directvideo* = 0 will work on any machine that is IBM BIOS-compatible, but the console output will be slower.

Programming in graphics mode

In this section, we give a brief summary of the functions you use in graphics mode. For more detailed information about these functions, refer to Chapter 2 of the *Library Reference*.

Borland C++ provides a separate library of over 70 graphics functions, ranging from high-level calls (like **setviewport**, **bar3d**, and **drawpoly**) to bit-oriented functions (like **getimage** and **putimage**). The graphics library supports numerous fill and line styles, and provides several text fonts that you can size, justify, and orient horizontally or vertically.

These functions are in the library file GRAPHICS.LIB, and they are prototyped in the header file graphics.h. In addition to these two files, the graphics package includes graphics device drivers (*.BGI files) and stroked character fonts (*.CHR files); we discuss these additional files in following sections.

In order to use the graphics functions:

- If you're using the IDE, toggle Full Menus to *On*, then check **Options | Linker | Graphics Library**. When you make your program, the linker automatically links in the Borland C++ graphics library.
- If you're using the command-line compiler (BCC.EXE or BCCX.EXE), you have to list GRAPHICS.LIB on the command line. For example, if your program MYPROG.C uses graphics, the BCC command line would be

```
BCC MYPROG GRAPHICS.LIB
```

Important! Because graphics functions use **far** pointers, graphics are not supported in the tiny memory model.

There is only one graphics library, not separate versions for each memory model (in contrast to the standard libraries CS.LIB, CC.LIB, CM.LIB, and so on, which are memory-model specific). Each function in GRAPHICS.LIB is a **far** function, and those graphics functions that take pointers take **far** pointers. For these functions to work correctly, it is important that you **#include** graphics.h in every module that uses graphics.

The graphics library functions

Borland C++'s graphics functions fall into seven categories:

- graphics system control
- drawing and filling
- manipulating screens and viewports
- text output
- color control
- error handling
- state query

Graphics system control

Here's a quick summary of the graphics system control:

closegraph	Shuts down the graphics system.
detectgraph	Checks the hardware and determines which graphics driver to use; recommends a mode.
graphdefaults	Resets all graphics system variables to their default settings.
_graphfreemem	Deallocates graphics memory; hook for defining your own routine.
_graphgetmem	Allocates graphics memory; hook for defining your own routine.
getgraphmode	Returns the current graphics mode.
getmoderange	Returns lowest and highest valid modes for specified driver.
initgraph	Initializes the graphics system and puts the hardware into graphics mode.
installuserdriver	Installs a vendor-added device driver to the BGI device driver table.
installuserfont	Loads a vendor-added stroked font file to the BGI character file table.
registerbgidriver	Registers a linked-in or user-loaded driver file for inclusion at link time.
restorecrtmode	Restores the original (pre- initgraph) screen mode.
setgraphbufsize	Specifies size of the internal graphics buffer.
setgraphmode	Selects the specified graphics mode, clears the screen, and restores all defaults.

Borland C++'s graphics package provides graphics drivers for the following graphics adapters (and true compatibles):

- Color/Graphics Adapter (CGA)

- Multi-Color Graphics Array (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Hercules Graphics Adapter
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter
- IBM 8514 Graphics Adapter

To start the graphics system, you first call the **initgraph** function. **initgraph** loads the graphics driver and puts the system into graphics mode.

You can tell **initgraph** to use a particular graphics driver and mode, or to autodetect the attached video adapter at run time and pick the corresponding driver. If you tell **initgraph** to autodetect, it calls **detectgraph** to select a graphics driver and mode. If you tell **initgraph** to use a particular graphics driver and mode, you must be sure that the hardware is present. If you force **initgraph** to use hardware that is not present, the results will be unpredictable.

Once a graphics driver has been loaded, you can find out the name of the driver by using the **getdrivername** function and how many modes a driver supports with **getmaxmode**. **getgraphmode** will tell you which graphics mode you are currently in. Once you have a mode number, you can find out the name of the mode with **getmodename**. You can change graphics modes with **setgraphmode** and return the video mode to its original state (before graphics was initialized) with **restorecrtmode**. **restorecrtmode** returns the screen to text mode, but it does not close the graphics system (the fonts and drivers are still in memory).

graphdefaults resets the graphics state's settings (viewport size, draw color, fill color and pattern, and so on) to their default values.

installuserdriver and **installuserfont** let you add new device drivers and fonts to your BGI.

Finally, when you're through using graphics, call **closegraph** to shut down the graphics system. **closegraph** unloads the driver from memory and restores the original video mode (via **restorecrtmode**).

A more detailed discussion

The previous discussion provided an overview of how **initgraph** operates. In the following paragraphs, we describe the behavior of **initgraph**, **_graphgetmem**, and **_graphfreemem** in some detail.

Normally, the **initgraph** routine loads a graphics driver by allocating memory for the driver, then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. You do this by first converting the .BGI file to an .OBJ file (using the BGIOBJ utility—see UTIL.DOC, included with your distribution disks), then placing calls to **registerbgidriver** in your source code (before the call to **initgraph**) to *register* the graphics driver(s). When you build your program, you need to link the .OBJ files for the registered drivers.

After determining which graphics driver to use (*via detectgraph*), **initgraph** checks to see if the desired driver has been registered. If so, **initgraph** uses the registered driver directly from memory. Otherwise, **initgraph** allocates memory for the driver and loads the .BGI file from disk.

Note Using **registerbgidriver** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in Chapter 2 of the *Library Reference*.

During run time, the graphics system might need to allocate memory for drivers, fonts, and internal buffers. If this is necessary, it calls **_graphgetmem** to allocate memory, and calls **_graphfreemem** to free it. By default, these routines simply call **malloc** and **free**, respectively.

If you provide your own **_graphgetmem** or **_graphfreemem**, you may get a “duplicate symbols” warning message. Just ignore the warning.

You can override this default behavior by defining your own **_graphgetmem** and **_graphfreemem** functions. By doing this, you can control graphics memory allocation yourself. You must, however, use the same names for your own versions of these memory-allocation routines: They will override the default functions with the same names that are in the standard C libraries.

Drawing and filling

Here’s a quick summary of the drawing and filling functions:

<i>Drawing:</i>	arc	Draws a circular arc.
	circle	Draws a circle.
	drawpoly	Draws the outline of a polygon.
	ellipse	Draws an elliptical arc.

getarccoords	Returns the coordinates of the last call to arc or ellipse .
getaspectratio	Returns the aspect ratio of the current graphics mode.
getlinesettings	Returns the current line style, line pattern, and line thickness.
line	Draws a line from (x_0,y_0) to (x_1,y_1) .
linerel	Draws a line to a point some relative distance from the current position (CP).
lineto	Draws a line from the current position (CP) to (x,y) .
moveto	Moves the current position (CP) to (x,y) .
moverel	Moves the current position (CP) a relative distance.
rectangle	Draws a rectangle.
setaspectratio	Changes the default aspect ratio-correction factor.
setlinestyle	Sets the current line width and style.

Filling:

bar	Draws and fills a bar.
bar3d	Draws and fills a 3-D bar.
fillellipse	Draws and fills an ellipse.
fillpoly	Draws and fills a polygon.
floodfill	Flood-fills a bounded region.
getfillpattern	Returns the user-defined fill pattern.
getfillsettings	Returns information about the current fill pattern and color.
pieslice	Draws and fills a pie slice.
sector	Draws and fills an elliptical pie slice.
setfillpattern	Selects a user-defined fill pattern.
setfillstyle	Sets the fill pattern and fill color.

With Borland C++'s drawing and painting functions, you can draw colored lines, arcs, circles, ellipses, rectangles, pie slices, two- and three-dimensional bars, polygons, and regular or irregular shapes based on combinations of these. You can fill any bounded shape (or any region surrounding such a shape) with one of eleven predefined patterns, or your own user-defined pattern. You can also control the thickness and style of the drawing line, and the location of the current position (CP).

You draw lines and unfilled shapes with the functions **arc**, **circle**, **drawpoly**, **ellipse**, **line**, **linerel**, **lineto**, and **rectangle**. You can fill these shapes with **floodfill**, or combine drawing/filling into one

step with **bar**, **bar3d**, **fillellipse**, **fillpoly**, **pieslice**, and **sector**. You use **setlinestyle** to specify whether the drawing line (and border line for filled shapes) is thick or thin, and whether its style is solid, dotted, and so forth, or some other line pattern you've defined. You can select a predefined fill pattern with **setfillstyle**, and define your own fill pattern with **setfillpattern**. You move the CP to a specified location with **moveto**, and move it a specified displacement with **moverel**.

To find out the current line style and thickness, you call **getline-settings**. For information about the current fill pattern and fill color, you call **getfillsettings**; you can get the user-defined fill pattern with **getfillpattern**.

You can get the aspect ratio (the scaling factor used by the graphics system to make sure circles come out round) with **getaspectratio**, and get coordinates of the last drawn arc or ellipse by calling **getarccoords**. If your circles are not perfectly round, use **setaspectsratio** to correct them.

Manipulating the screen and viewport

Here's a quick summary of the screen-, viewport-, image-, and pixel-manipulation functions:

Screen manipulation:

cleardevice Clears the screen (active page).
setactivepage Sets the active page for graphics output.
setvisualpage Sets the visual graphics page number.

Viewport manipulation:

clearviewport Clears the current viewport.
getviewsettings Returns information about the current viewport.
setviewport Sets the current output viewport for graphics output.

Image manipulation:

getimage Saves a bit image of the specified region to memory.
imagesize Returns the number of bytes required to store a rectangular region of the screen.
putimage Puts a previously saved bit image onto the screen.

Pixel manipulation:

getpixel Gets the pixel color at (x,y) .
putpixel Plots a pixel at (x,y) .

Besides drawing and painting, the graphics library offers several functions for manipulating the screen, viewports, images, and pixels. You can clear the whole screen in one fell swoop with a call to **cleardevice**; this routine erases the entire screen and homes

the CP in the viewport, but leaves all other graphics system settings intact (the line, fill, and text styles; the palette; the viewport settings; and so on).

Depending on your graphics adapter, your system has between one and four screen-page buffers, which are areas in memory where individual whole-screen images are stored dot-by-dot. You can specify the active screen page (where graphics functions place their output) with **setactivepage** and the visual page (the one displayed onscreen) with **setvisualpage**.

Once your screen is in a graphics mode, you can define a viewport (a rectangular "virtual screen") on your screen with a call to **setviewport**. You define the viewport's position in terms of absolute screen coordinates and specify whether clipping is on (active) or off. You clear the viewport with **clearviewport**. To find out the current viewport's absolute screen coordinates and clipping status, call **getviewsettings**.

You can capture a portion of the onscreen image with **getimage**, call **imagesize** to calculate the number of bytes required to store that captured image in memory, then put the stored image back on the screen (anywhere you want) with **putimage**.

The coordinates for all output functions (drawing, filling, text, and so on) are viewport-relative.

You can also manipulate the color of individual pixels with the functions **getpixel** (which returns the color of a given pixel) and **putpixel** (which plots a specified pixel in a given color).

Text output in graphics mode

Here's a quick summary of the graphics-mode text output functions:

gettextsettings	Returns the current text font, direction, size, and justification.
outtext	Sends a string to the screen at the current position (CP).
outtextxy	Sends a string to the screen at the specified position.
registerbgifont	Registers a linked-in or user-loaded font.
settextjustify	Sets text justification values used by outtext and outtextxy .
settextstyle	Sets the current text font, style, and character magnification factor.
setusercharsize	Sets width and height ratios for stroked fonts.

textheight Returns the height of a string in pixels.
textwidth Returns the width of a string in pixels.

The graphics library includes an 8×8 bit-mapped font and several stroked fonts for text output while in graphics mode.

- In a *bit-mapped* font, each character is defined by a matrix of pixels.
- In a *stroked* font, each character is defined by a series of vectors that tell the graphics system how to draw that character.

The advantage of using a stroked font is apparent when you start to draw large characters. Since a stroked font is defined by vectors, it will still retain good resolution and quality when the font is enlarged. On the other hand, when you enlarge a bit-mapped font, the matrix is multiplied by a scaling factor; as the scaling factor becomes larger, the characters' resolution becomes coarser. For small characters, the bit-mapped font should be sufficient, but for larger text you should select a stroked font.

You output graphics text by calling either **outtext** or **outtextxy**, and control the justification of the output text (with respect to the CP) with **settextjustify**. You choose the character font, direction (horizontal or vertical), and size (scale) with **settextstyle**. You can find out the current text settings by calling **gettextsettings**, which returns the current text font, justification, magnification, and direction in a **textsettings** structure. **setusercharsize** allows you to modify the character width and height of stroked fonts.

If clipping is *on*, all text strings output by **outtext** and **outtextxy** will be clipped at the viewport borders. If clipping is *off*, these functions will throw away bit-mapped font output if any part of the text string would go off the screen edge; stroked font output is truncated at the screen edges.

To determine the onscreen size of a given text string, call **textheight** (which measures the string's height in pixels) and **textwidth** (which measures its width in pixels).

The default 8×8 bit-mapped font is built into the graphics package, so it is always available at run time. The stroked fonts are each kept in a separate .CHR file; they can be loaded at run time or converted to .OBJ files (with the BGI OBJ utility) and linked into your .EXE file.

Normally, the **settextstyle** routine loads a font file by allocating memory for the font, then loading the appropriate .CHR file from

disk. As an alternative to this dynamic loading scheme, you can link a character font file (or several of them) directly into your executable program file. You do this by first converting the .CHR file to an .OBJ file (using the BGIOBJ utility—read about it in UTIL.DOC, the online documentation included with your distribution disks), then placing calls to **registerbgifont** in your source code (before the call to **settextstyle**) to *register* the character font(s). When you build your program, you need to link in the .OBJ files for the stroked fonts you register.

Note Using **registerbgifont** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in UTIL.DOC, included with your distribution disks.

Color control Here's a quick summary of the color control functions:

<i>Get color information:</i>	getbkcolor	Returns the current background color.
	getcolor	Returns the current drawing color.
	getdefaultpalette	Returns the palette definition structure.
	getmaxcolor	Returns the maximum color value available in the current graphics mode.
	getpalette	Returns the current palette and its size.
<i>Set one or more colors:</i>	getpalettesize	Returns the size of the palette look-up table.
	setallpalette	Changes all palette colors as specified.
	setbkcolor	Sets the current background color.
	setcolor	Sets the current drawing color.
	setpalette	Changes one palette color as specified by its arguments.

Before summarizing how these color control functions work, we first present a basic description of how colors are actually produced on your graphics screen.

Pixels and palettes The graphics screen consists of an array of pixels; each pixel produces a single (colored) dot onscreen. The pixel's value does not specify the precise color directly; it is an index into a color table called a *palette*. The palette entry corresponding to a given pixel value contains the exact color information for that pixel.

This indirection scheme has a number of implications. Though the hardware might be capable of displaying many colors, only a subset of those colors can be displayed at any given time. The number of colors that can be displayed at any one time is equal to the

number of entries in the palette (the palette's *size*). For example, on an EGA, the hardware can display 64 different colors, but only 16 of them at a time; the EGA palette's *size* is 16.

The *size* of the palette determines the range of values a pixel can assume, from 0 to (*size* - 1). **getmaxcolor** returns the highest valid pixel value (*size* - 1) for the current graphics driver and mode.

When we discuss the Borland C++ graphics functions, we often use the term *color*, such as the current drawing color, fill color and pixel color. In fact, this color is a pixel's value: it's an index into the palette. Only the palette determines the true color on the screen. By manipulating the palette, you can change the actual color displayed on the screen even though the pixel values (drawing color, fill color, and so on) have not changed.

Background and drawing color

The *background color* always corresponds to pixel value 0. When an area is cleared to the background color, that area's pixels are simply set to 0.

The *drawing color* is the value to which pixels are set when lines are drawn. You choose a drawing color with `setcolor(n)`, where *n* is a valid pixel value for the current palette.

Color control on a CGA

Due to graphics hardware differences, how you actually control color differs quite a bit between CGA and EGA, so we'll present them separately. Color control on the AT&T driver, and the lower resolutions of the MCGA driver is similar to CGA.

On the CGA, you can choose to display your graphics in low resolution (320×200), which allows you to use four colors, or high resolution (640×200), in which you can use two colors.

CGA low resolution

In the low resolution modes, you can choose from four predefined four-color palettes. In any of these palettes, you can only set the first palette entry; entries 1, 2, and 3 are fixed. The first palette entry (color 0) is the background color. This background color can be any one of the 16 available colors (see table of CGA background colors below).

You choose which palette you want by the mode you select (CGAC0, CGAC1, CGAC2, CGAC3); these modes use color palette 0 through color palette 3, as detailed in the following table.

The CGA drawing colors and the equivalent constants are defined in graphics.h.

Palette number	Constant assigned to color number (pixel value)		
	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

To assign one of these colors as the CGA drawing color, call **setcolor** with either the color number or the corresponding constant name as an argument; for example, if you are using palette 3 and you want to use cyan as the drawing color:

```
setcolor(1);
```

or

```
setcolor(CGA_CYAN);
```

The available CGA background colors, defined in graphics.h, are listed in the following table:

Numeric value	Symbolic name	Numeric value	Symbolic name
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
3	CYAN	11	LIGHTCYAN
4	RED	12	LIGHTRED
5	MAGENTA	13	LIGHTMAGENTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

The CGA's foreground colors are the same as those listed in this table.

To assign one of these colors to the CGA background color, use **setbkcolor**(*color*), where *color* is one of the entries in the preceding table. For CGA, this color is not a pixel value (palette index); it directly specifies the *actual* color to be put in the first palette entry.

CGA high resolution

In high resolution mode (640×200), the CGA displays two colors: a black background and a colored foreground. Pixels can take on values of either 0 or 1. Because of a quirk in the CGA itself, the foreground color is actually what the hardware thinks of as its

background color; you set it with the **setbkcolor** routine. (Strange but true.)

The colors available for the colored foreground are those listed in the preceding table. The CGA uses this color to display all pixels whose value equals 1.

The modes that behave in this way are CGAHI, MCGAMED, MCGAHI, ATT400MED, and ATT400HI.

CGA palette routines

Because the CGA palette is predetermined, you should not use the **setallpalette** routine on a CGA. Also, you should not use **setpalette**(*index, actual_color*), except for *index* = 0. (This is an alternate way to set the CGA background color to *actual_color*.)

Color control on the
EGA and VGA

On the EGA, the palette contains 16 entries from a total of 64 possible colors, and each entry is user-settable. You can retrieve the current palette with **getpalette**, which fills in a structure with the palette's size (16) and an array of the actual palette entries (the "hardware color numbers" stored in the palette). You can change the palette entries individually with **setpalette**, or all at once with **setallpalette**.

The default EGA palette corresponds to the 16 CGA colors, as given in the previous color table: black is in entry 0, blue in entry 1, ..., white in entry 15. There are constants defined in `graphics.h` that contain the corresponding hardware color values: these are `EGA_BLACK`, `EGA_WHITE`, and so on. You can also get these values with **getpalette**.

The **setbkcolor**(*color*) routine behaves differently on an EGA than on a CGA. On an EGA, **setbkcolor** copies the actual color value that's stored in entry *#color* into entry #0.

As far as colors are concerned, the VGA driver behaves like the EGA driver; it just has higher resolution (and smaller pixels).

Error handling in
graphics mode

Here's a quick summary of the graphics-mode error-handling functions:

grapherrormsg Returns an error message string for the specified error code.

graphresult Returns an error code for the last graphics operation that encountered a problem.

If an error occurs when a graphics library function is called (such as a font requested with **settextstyle** not being found), an internal error code is set. You retrieve the error code for the last graphics operation that reported an error by calling **graphresult**. A call to **grapherrormsg(graphresult())** returns the error strings listed in the following table.

The error return code accumulates, changing only when a graphics function reports an error. The error return code is reset to 0 only when **initgraph** executes successfully, or when you call **graphresult**. Therefore, if you want to know which graphics function returned which error, you should store the value of **graphresult** into a temporary variable and then test it.

Error code	<i>graphics_errors</i> constant	Corresponding error message string
0	grOk	No error
-1	grNoInitGraph	(BGI) graphics not installed (use initgraph)
-2	grNotDetected	Graphics hardware not detected
-3	grFileNotFound	Device driver file not found
-4	grInvalidDriver	Invalid device driver file
-5	grNoLoadMem	Not enough memory to load driver
-6	grNoScanMem	Out of memory in scan fill
-7	grNoFloodMem	Out of memory in flood fill
-8	grFontNotFound	Font file not found
-9	grNoFontMem	Not enough memory to load font
-10	grInvalidMode	Invalid graphics mode for selected driver
-11	grError	Graphics error
-12	grIOerror	Graphics I/O error
-13	grInvalidFont	Invalid font file
-14	grInvalidFontNum	Invalid font number
-15	grInvalidDeviceNum	Invalid device number
-18	grInvalidVersion	Invalid version of file

State query The following table summarizes the graphics mode state query functions:

Table 11.1
Graphics mode state query
functions

Function	Returns
getarccoords	Information about the coordinates of the last call to arc or ellipse .
getaspectratio	Aspect ratio of the graphics screen.
getbkcolor	Current background color.
getcolor	Current drawing color.
getdrivename	Name of current graphics driver.
getfillpattern	User-defined fill pattern.
getfillsettings	Information about the current fill pattern and color.
getgraphmode	Current graphics mode.
getlinesettings	Current line style, line pattern, and line thickness.
getmaxcolor	Current highest valid pixel value.
getmaxmode	Maximum mode number for current driver.
getmaxx	Current <i>x</i> resolution.
getmaxy	Current <i>y</i> resolution.
getmodename	Name of a given driver mode.
getmoderange	Mode range for a given driver.
getpalette	Current palette and its size.
getpixel	Color of the pixel at <i>x,y</i> .
gettextsettings	Current text font, direction, size, and justification.
getviewsettings	Information about the current viewport.
getx	<i>x</i> coordinate of the current position (CP).
gety	<i>y</i> coordinate of the current position (CP).

In each of Borland C++'s graphics functions categories there is at least one state query function. These functions are mentioned under their respective categories and also covered here. Each of the Borland C++ graphics state query functions is named **getsomething** (except in the error-handling category). Some of them take no argument and return a single value representing the requested information; others take a pointer to a structure defined in `graphics.h`, fill that structure with the appropriate information, and return no value.

The state query functions for the graphics system control category are **getgraphmode**, **getmaxmode**, and **getmoderange**: The first returns an integer representing the current graphics driver and mode, the second returns the maximum mode number for a given driver, and the third returns the range of modes supported by a given graphics driver. **getmaxx** and **getmaxy** return the maximum *x* and *y* screen coordinates for the current graphics mode.

The drawing and filling state query functions are **getarccoords**, **getaspectratio**, **getfillpattern**, **getfillsettings**, and **getlinesettings**. **getarccoords** fills a structure with coordinates from the last call to **arc** or **ellipse**; **getaspectratio** tells the current mode's aspect ratio, which the graphics system uses to make circles come out round.

getfillpattern returns the current user-defined fill pattern. **getfillsettings** fills a structure with the current fill pattern and fill color. **getlinesettings** fills a structure with the current line style (solid, dashed, and so on), line width (normal or thick), and line pattern.

In the screen- and viewport-manipulation category, the state query functions are **getviewsettings**, **getx**, **gety**, and **getpixel**. When you have defined a viewport, you can find out its absolute screen coordinates and whether clipping is active by calling **getviewsettings**, which fills a structure with the information. **getx** and **gety** return the (viewport-relative) x- and y-coordinates of the CP. **getpixel** returns the color of a specified pixel.

The graphics mode text-output function category contains one all-inclusive state query function: **gettextsettings**. This function fills a structure with information about the current character font, the direction in which text will be displayed (horizontal or bottom-to-top vertical), the character magnification factor, and the text-string justification (both horizontal and vertical).

Borland C++'s color-control function category includes three state query functions. **getbkcolor** returns the current background color, and **getcolor** returns the current drawing color. **getpalette** fills a structure with the size of the current drawing palette and the palette's contents. **getmaxcolor** returns the highest valid pixel value for the current graphics driver and mode (palette size - 1).

Finally, **getmodename** and **getdrivername** return the name of a given driver mode and the name of the current graphics driver, respectively.

BASM and inline assembly

This chapter tells you how to use the Borland C++ built-in inline assembler (BASM) to include assembly language routines in your C and C++ programs without any need for a separate assembler. Such assembly language routines are called *inline assembly*, because they are compiled right along with your C routines, rather than being assembled separately, then linked together with modules produced by the C compiler.

Of course, Borland C++ also supports traditional mixed-language programming in which your C program calls assembly language routines (or vice-versa) that are separately assembled by TASM (Turbo Assembler). In order to interface C and assembly language, you must know how to write 80x86 assembly language routines and how to define segments, data constants, and so on. You also need to be familiar with *calling conventions* (parameter passing sequences) in C and assembly language, including the **pascal** parameter passing sequence in C. If you are unfamiliar with these concepts, read the Turbo Assembler manuals for more information, especially “Interfacing Turbo Assembler with Borland C++” in the *Turbo Assembler User’s Guide*. Turbo Assembler includes several features that make interfacing with Borland C++ easy and transparent.

Inline assembly language

Borland C++ lets you write assembly language code right inside your C and C++ programs. This is known as *inline assembly*.

By default, **-B** invokes TASM. You can override it with **-Exxx**, where *xxx* is another assembler. See Chapter 5, "The command-line compiler," in the User's Guide for details.

You can use the **-B** compiler option for inline assembly in your C program. If you use this option, the compiler first generates an assembly file, then invokes TASM on that file to produce the .OBJ file.

You can invoke TASM while omitting the **-B** option if you include the `#pragma inline` statement in your source code. This statement enables the **-B** option for you when the compiler encounters it.

BASM If you don't invoke TASM, Borland C++ can assemble your inline assembly instructions using the built-in assembler (BASM). This assembler can do everything TASM can do with the following restrictions:

- It cannot use assembler macros
- It cannot handle 80386 or 80486 instructions
- It does not permit Ideal mode syntax
- It allows only a limited set of assembler directives (see page 404)

Inline syntax Of course, you also need to be familiar with the 80x86 instruction set and architecture. Even though you're not writing complete assembly language routines, you still need to know how the instructions you're using work, how to use them, and how not to use them.

Having done all that, you need only use the keyword **asm** to introduce an inline assembly language instruction. The format is

```
asm opcode operands ; or newline
```

where

- *opcode* is a valid 80x86 instruction (Table 12.1 lists all allowable *opcodes*). For 80286 instructions, use the **-2** command-line compiler option or the 80286 instruction set option (O | C | Advanced Code Generation).
- *operands* contains the operand(s) acceptable to the *opcode*, and can reference C constants, variables, and labels.
- *;* or *newline* is a semicolon or a new line, either of which signals the end of the **asm** statement.

A new **asm** statement can be placed on the same line, following a semicolon, but no **asm** statement can continue to the next line.

To include a number of **asm** statements, surround them with braces:

The initial brace **must** appear on the same line as the **asm** keyword.

```
asm {
    pop ax; pop ds
    iret
}
```

Semicolons are not used to start comments (as they are in TASM). When commenting **asm** statements, use C-style comments, like this:

```
asm mov ax,ds;           /* This comment is OK */
asm {pop ax; pop ds; iret;} /* This is legal too */
asm push ds              ;THIS COMMENT IS INVALID!!
```

The assembly language portion of the statement is copied straight to the output, embedded in the assembly language that Borland C++ is generating from your C or C++ instructions. Any C symbols are replaced with appropriate assembly language equivalents.

Because the inline assembly facility is not a complete assembler, it may not accept some assembly language constructs. If this happens, Borland C++ will issue an error message. You then have two choices. You can simplify your inline assembly language code so that the assembler will accept it, or you can use the **-B** option to invoke TASM. If you do, TASM will catch whatever errors there might be. However, TASM might not identify the location of errors, since the original C source line number is lost.

Each **asm** statement counts as a C statement. For example,

```
myfunc()
{
    int i;
    int x;

    if (i > 0)
        asm mov x,4
    else
        i = 7;
}
```

This construct is a valid C **if** statement. Note that no semicolon was needed after the `mov x,4` instruction. **asm** statements are the only statements in C that depend on the occurrence of a new line. This is not in keeping with the rest of the C language, but this is the convention adopted by several UNIX-based compilers.

An assembly statement can be used as an executable statement inside a function, or as an external declaration outside of a function. Assembly statements located outside any function are placed in the data segment, and assembly statements located inside functions are placed in the code segment.

Opcodes You can include any of the 80x86 instruction opcodes as inline assembly statements. There are four classes of instructions allowed by the Borland C++ compiler:

- normal instructions—the regular 80x86 opcode set
- string instructions—special string-handling codes
- jump instructions—various jump opcodes
- assembly directives—data allocation and definition

Note that all operands are allowed by the compiler, even if they are erroneous or disallowed by the assembler. The exact format of the operands is not enforced by the compiler.

The following is a summary list of the opcode mnemonics that can be used in inline assembler:

Table 12.1
Opcode mnemonics

*If you are using inline assembly in routines that use floating-point emulation (the command-line compiler option -f), the opcodes marked with ** are not supported.*

aaa	fclex	fldenv	fstenv
aad	fcom	fldl2e	fstp
aam	fcomp	fldl2t	fstsw
aas	fcompp	fldlg2	fsub
adc	fdecstp**	fldln2	fsubp
add	fdisi	fldpi	fsubr
and	fdiv	fldz	fsubrp
bound	fdivp	fmul	ftst
call	fdivr	fmulp	fwait
cbw	fdivrp	fnclx	fxam
clc	feni	fndisi	fxch
cld	ffree**	fneni	fextract
cli	fiadd	fninit	fyl2x
cmc	ficom	fnop	fyl2xp1
cmp	ficomp	fnsave	hlt
cwd	fidiv	fnstcw	idiv
daa	fidivr	fnstenv	imul
das	fidl	fnstsw	in
dec	fimul	fpatan	inc
div	fincstp**	fprem	int
enter	finit	fptan	into
f2xm1	fist	frndint	iret
fabs	fistp	frstor	lahf
fadd	fisub	fsave	lds
faddp	fisubr	fscale	lea
fbld	fld	fsqrt	leave
fbstp	fldl	fst	les
fchs	fldcw	fstcw	lsl

Table 12.1: Opcode mnemonics (continued)

mov	popf	sahf	sti
mul	push	sal	sub
neg	pusha	sar	test
nop	pushf	sbb	verr
not	rcl	shl	verw
or	rcr	shr	wait
out	ret	smsw	xchg
pop	rol	stc	xlat
popa	ror	std	xor

When using 80186 instruction mnemonics in your inline assembly statements, you must include the `-1` command-line option. This forces appropriate statements into the assembly language compiler output so that the assembler will expect the mnemonics. If you are using an older assembler, these mnemonics may not be supported.

String instructions

In addition to the listed opcodes, the string instructions given in the following table can be used alone or with repeat prefixes.

Table 12.2
String instructions

cmps	insw	movsb	outsw	stos
cmpsb	lods	movsw	scas	stosb
cmpsw	lodsb	outs	scasb	stosw
ins	lodsw	outsb	scasw	
insb	movs			

Prefixes

The following prefixes can be used:

`lock rep repe repne repnz repz`

Jump instructions

Jump instructions are treated specially. Since a label cannot be included on the instruction itself, jumps must go to C labels (discussed in “Using jump instructions and labels” on page 406). The allowed jump instructions are given in the next table.

Table 12.3
Jump instructions

ja	jge	jnc	jns	loop
jae	jl	jne	jnz	loope
jb	jle	jng	jo	loopne
jbe	jmp	jnge	jp	loopnz
jc	jna	jnl	jpe	loopz
jcxz	jnae	jnle	jpo	
je	jnb	jno	js	
jg	jnbe	jnp	jz	

Assembly directives

The following assembly directives are allowed in Borland C++ inline assembly statements:

db dd dw extrn

Inline assembly
references to data and
functions

You can use C symbols in your **asm** statements; Borland C++ automatically converts them to appropriate assembly language operands and appends underscores onto identifier names. You can use any symbol, including automatic (local) variables, register variables, and function parameters.

In general, you can use a C symbol in any position where an address operand would be legal. Of course, you can use a register variable wherever a register would be a legal operand.

If the assembler encounters an identifier while parsing the operands of an inline assembly instruction, it searches for the identifier in the C symbol table. The names of the 80x86 registers are excluded from this search. Either uppercase or lowercase forms of the register names can be used.

Inline assembly and register variables

Inline assembly code can freely use SI or DI as scratch registers. If you use SI or DI in inline assembly code, the compiler won't use these registers for register variables.

Inline assembly, offsets, and size overrides

When programming, you don't need to be concerned with the exact offsets of local variables. Simply using the name will include the correct offsets.

However, it may be necessary to include appropriate WORD PTR, BYTE PTR, or other size overrides on assembly instruction. A DWORD PTR override is needed on LES or indirect far call instructions.

Using C structure members

You can reference structure members in an inline assembly statement in the usual fashion (that is, *variable.member*). In such a case, you are dealing with a variable, and you can store or retrieve values. However, you can also directly reference the member name (without the variable name) as a form of numeric constant. In this situation, the constant equals the offset (in bytes) from the start of the structure containing that member. Consider the following program fragment:

```
struct myStruct {
    int a_a;
    int a_b;
    int a_c;
} myA ;

myfunc()
{
    ...
    asm {mov ax, myA.a_b
        mov bx, [di].a_c
        }
    ...
}
```

We've declared a structure type named *myStruct* with three members, *a_a*, *a_b*, and *a_c*; we've also declared a variable *myA* of type *myStruct*. The first inline assembly statement moves the value contained in *myA.a_b* into the register AX. The second moves the value at the address $[di] + \text{offset}(a_c)$ into the register BX (it takes the address stored in DI and adds to it the offset of *a_c* from the start of *myStruct*). In this sequence, these assembler statements produce the following code:

```
mov ax, DGROUP : myA+2
mov bx, [di+4]
```

Why would you even want to do this? If you load a register (such as DI) with the address of a structure of type *myStruct*, you can use the member names to directly reference the members. The member name actually can be used in any position where a numeric constant is allowed in an assembly statement operand.

The structure member must be preceded by a dot (.) to signal that a member name, rather than a normal C symbol, is being used. Member names are replaced in the assembly output by the numeric offset of the structure member (the numeric offset of `a_c` is 4), but no type information is retained. Thus members can be used as compile-time constants in assembly statements.

However, there is one restriction. If two structures that you are using in inline assembly have the same member name, you must distinguish between them. Insert the structure type (in parentheses) between the dot and the member name, as if it were a cast. For example,

```
asm mov bx,[di].(struct tm)tm_hour
```

Using jump instructions and labels

You can use any of the conditional and unconditional jump instructions, plus the loop instructions, in inline assembly. They are only valid inside a function. Since no labels can be defined in the **asm** statements, jump instructions must use C **goto** labels as the object of the jump. If the label is too far away, the jump will be automatically converted to a long-distance jump. Direct far jumps cannot be generated.

In the following code, the jump goes to the C **goto** label *a*.

```
int x()
{
a:                /* This is the goto label "a" */
...
asm jmp a        /* Goes to label "a" */
...
}
```

Indirect jumps are also allowed. To use an indirect jump, you can use a register name as the operand of the jump instruction.

Interrupt functions

The 80x86 reserves the first 1024 bytes of memory for a set of 256 far pointers—known as interrupt vectors—to special system routines known as *interrupt handlers*. These routines are called by executing the 80x86 instruction

```
int int#
```

where *int#* goes from 0h to FFh. When this happens, the computer saves the code segment (CS), instruction pointer (IP), and status flags, disables the interrupts, then does a far jump to the location

pointed to by the corresponding interrupt vector. For example, one interrupt call you're likely to see is

```
int 21h
```

which calls most DOS routines. But many of the interrupt vectors are unused, which means, of course, that you can write your own interrupt handler and put a **far** pointer to it into one of the unused interrupt vectors.

To write an interrupt handler in Borland C++, you must define the function to be of type **interrupt**; more specifically, it should look like this:

```
void interrupt myhandler(bp, di, si, ds, es, dx,  
                        cx, bx, ax, ip, cs, flags, ... );
```

As you can see, all the registers are passed as parameters, so you can use and modify them in your code without using the pseudo-variables discussed earlier in this chapter. You can also pass additional parameters (*flags, ...*) to the handler; those should be defined appropriately.

A function of type **interrupt** will automatically save (in addition to SI, DI, and BP) the registers AX through DX, ES, and DS. These same registers are restored on exit from the interrupt handler.

Interrupt handlers may use floating-point arithmetic in all memory models. Any interrupt handler code that uses an 80x87 must save the state of the chip on entry and restore it on exit from the handler.

An interrupt function can modify its parameters. Changing the declared parameters will modify the corresponding register when the interrupt handler returns. This may be useful when you are using an interrupt handler to act as a user service, much like the DOS INT 21 services. Also, note that an interrupt function exits with an IRET (return from interrupt) instruction.

So, why would you want to write your own interrupt handler? For one thing, that's how most memory-resident routines work. They install themselves as interrupt handlers. That way, whenever some special or periodic action takes place (clock tick, keyboard press, and so on), these routines can intercept the call to the routine handling the interrupt and see what action needs to take place. Having done that, they can then pass control on to the routine that was there.

Using low-level practices

You've already seen a few examples of how to use these different low-level practices in your code; now it's time to look at a few more. Let's start with an interrupt handler that does something harmless but tangible (or, in this case, audible): It beeps whenever it's called.

First, write the function itself. Here's what it might look like:

```
#include <dos.h>

void interrupt mybeep(unsigned bp, unsigned di, unsigned si,
                    unsigned ds, unsigned es, unsigned dx,
                    unsigned cx, unsigned bx, unsigned ax)
{
    int i, j;
    char originalbits, bits;
    unsigned char bcount = ax >> 8;

    /* Get the current control port setting */
    bits = originalbits = inportb(0x61);

    for (i = 0; i <= bcount; i++){
        /* Turn off the speaker for awhile */
        outportb(0x61, bits & 0xfc);
        for (j = 0; j <= 100; j++)
            ; /* empty statement */

        /* Now turn it on for some more time */
        outportb(0x61, bits | 2);
        for (j = 0; j <= 100; j++)
            ; /* another empty statement */
    }

    /* Restore the control port setting */
    outportb(0x61, originalbits);
}
```

Next, write a function to install your interrupt handler. Pass it the address of the function and its interrupt number (0 to 255 or 0x00 to 0xFF).

```
void install(void interrupt (*faddr)(), int inum)
{
    setvect(inum, faddr);
}
```

Finally, call your beep routine to test it out. Here's a function to do just that:

```
void testbeep(unsigned char bcount, int inum)
{
    _AH = bcount;
    geninterrupt(inum);
}
```

Your **main** function might look like this:

```
main()
{
    char ch;

    install(mybeep,10);
    testbeep(3,10);
    ch = getch();
}
```

You might also want to preserve the original interrupt vector and restore it when your main program is finished. Use the **getvect** and **setvect** functions to do this.

ANSI implementation-specific standards

Certain aspects of the ANSI C standard are not defined exactly by ANSI. Instead, each implementor of a C compiler is free to define these aspects individually. This chapter tells how Borland has chosen to define these implementation-specific standards. The section numbers refer to the February 1990 ANSI Standard. Remember that there are differences between C and C++; this appendix addresses C only.

2.1.1.3 How to identify a diagnostic.

When the compiler runs with the correct combination of options, any message it issues beginning with the words *Fatal*, *Error*, or *Warning* are diagnostics in the sense that ANSI specifies. The options needed to ensure this interpretation are as follows:

Table A.1
Identifying diagnostics in C++

Option	Action
-A	Enable only ANSI keywords.
-C-	No nested comments allowed.
-p-	Use C calling conventions.
-i32	At least 32 significant characters in identifiers.
-w-	Turn off all warnings except the following.
-wbei	Turn on warning about inappropriate initializers.
-wdcl	Turn on warning about declarations without type or storage class.
-wcpt	Turn on warning about nonportable pointer comparisons.
-wdup	Turn on warning about duplicate nonidentical macro definitions.
-wsus	Turn on warning about suspicious pointer conversion.

Table A.1: Identifying diagnostics in C++ (continued)

-wrpt	Turn on warning about nonportable pointer conversion.
-wvrt	Turn on warning about void functions returning a value.
-wbig	Turn on warning about constants being too large.
-wucp	Turn on warning about mixing pointers to signed and unsigned char.
-wstu	Turn on warning about undefined structures.
-wext	Turn on warning about variables declared both as external and as static.
-wfdt	Turn on warning about function definitions using a typedef.

None of the following options can be used:

-ms!	SS must be the same as DS for small data models.
-mm!	SS must be the same as DS for small data models.
-mt!	SS must be the same as DS for small data models.
-zGxx	The BSS group name may not be changed.
-zSxx	The data group name may not be changed.

Other options not specifically mentioned here can be set to whatever you desire.

2.1.2.2.1 The semantics of the arguments to main.

The value of *argv*[0] is a pointer to a null byte when the program is run on DOS versions prior to version 3.0. For DOS version 3.0 or later, *argv*[0] points to the program name.

The remaining *argv* strings point to each component of the DOS command-line arguments. Whitespace separating arguments is removed, and each sequence of contiguous nonwhitespace characters is treated as a single argument. Quoted strings are handled correctly (that is, as one string containing spaces).

2.1.2.3 What constitutes an interactive device.

An interactive device is any device that looks like the console.

2.2.1 The collation sequence of the execution character set.

The collation sequence for the execution character set uses the signed value of the character in ASCII.

2.2.1 Members of the source and execution character sets.

The source and execution character sets are the extended ASCII set supported by the IBM PC. Any character other than ^Z (Control-Z) can appear in string literals, character constants, or comments.

2.2.1.2 Multibyte characters.

No multibyte characters are supported in Borland C++.

2.2.2 The direction of printing.

Printing is from left-to-right, the normal direction for the PC.

2.2.4.2 The number of bits in a character in the execution character set.

There are 8 bits per character in the execution character set.

3.1.2 The number of significant initial characters in identifiers.

The first 32 characters are significant, although you can use a command-line option (-i) to change that number. Both internal and external identifiers use the same number of significant digits. (The number of significant characters in C++ identifiers is unlimited.)

3.1.2 Whether case distinctions are significant in external identifiers.

The compiler will normally force the linker to distinguish between uppercase and lowercase. You can use a command-line option (-l-c) to suppress the distinction.

3.1.2.5 The representations and sets of values of the various types of integers.

Type	Minimum value	Maximum value
signed char	-128	127
unsigned char	0	255
signed short	-32,768	32,767
unsigned short	0	65,535
signed int	-32,768	32,767
unsigned int	0	65,535
signed long	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295

All **char** types use one 8-bit byte for storage.

All **short** and **int** types use 2 bytes.

All **long** types use 4 bytes.

If alignment is requested (-a), all non**char** integer type objects will be aligned to even byte boundaries. Character types are never aligned.

3.1.2.5 The representations and sets of values of the various types of floating-point numbers.

The IEEE floating-point formats as used by the Intel 8087 are used for all Borland C++ floating-point types. The **float** type uses 32-bit IEEE real format. The **double** type uses 64-bit IEEE real format. The **long double** type uses 80-bit IEEE extended real format.

3.1.3.4 The mapping between source and execution character sets.

Any characters in string literals or character constants will remain unchanged in the executing program. The source and execution character sets are the same.

3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant.

Wide characters are not supported. They are treated as normal characters. All legal escape sequences map onto one or another character. If a hex or octal escape sequence is used that exceeds the range of a character, the compiler issues a message.

3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters for a wide character constant.

Wide character constants are recognized, but treated in all ways like normal character constants. In that sense, the locale is the "C" locale.

3.1.3.4 The value of an integer constant that contains more than one character, or a wide character constant that contains more than one multibyte character.

Character constants can contain one or two characters. If two characters are included, the first character occupies the low-order byte of the constant, and the second character occupies the high-order byte.

3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.

These conversions are performed by simply truncating the high-order bits. Signed integers are stored as 2's-complement values, so the resulting number is interpreted as such a value. If the high-

order bit of the smaller integer is nonzero, the value is interpreted as a negative value; otherwise, it is positive.

3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.

The integer value is rounded to the nearest representable value. Thus, for example, the **long** value ($2^{31} - 1$) is converted to the **float** value 2^{31} . Ties are broken according to the rules of IEEE standard arithmetic.

3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number.

The value is rounded to the nearest representable value. Ties are broken according to the rules of IEEE standard arithmetic.

3.3 The results of bitwise operations on signed integers.

The bitwise operators apply to signed integers as if they were their corresponding unsigned types. The sign bit is treated as a normal data bit. The result is then interpreted as a normal 2's complement signed integer.

3.3.2.3 What happens when a member of a union object is accessed using a member of a different type.

The access is allowed and will simply access the bits stored there. You'll need a detailed understanding of the bit encodings of floating-point values in order to understand how to access a floating-type member using a different member. If the member stored is shorter than the member used to access the value, the excess bits have the value they had before the short member was stored.

3.3.3.4 The type of integer required to hold the maximum size of an array.

For a normal array, the type is **unsigned int**, and for huge arrays the type is **signed long**.

3.3.4 The result of casting a pointer to an integer or vice versa.

When converting between integers and pointers of the same size, no bits are changed. When converting from a longer type to a shorter, the high-order bits are truncated. When converting from a shorter integer type to a longer pointer type, the integer is first widened to an integer type that is the same size as the pointer type. Thus signed integers will sign-extend to fill the new bytes.

Similarly, smaller pointer types being converted to larger integer types will first be widened to a pointer type that is as wide as the integer type.

3.3.5 The sign of the remainder on integer division.

The sign of the remainder is negative when only one of the operands is negative. If neither or both operands are negative, the remainder is positive.

3.3.6 The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t`.

The type is **signed int** when the pointers are near, or **signed long** when the pointers are far or huge. The type of `ptrdiff_t` depends on the memory model in use. In small data models, the type is **int**. In large data models, the type is **long**.

3.3.7 The result of a right shift of a negative signed integral type.

A negative signed value is sign-extended when right shifted.

3.5.1 The extent to which objects can actually be placed in registers by using the *register* storage-class specifier.

Objects declared with any two-byte integer or pointer types can be placed in registers. The compiler will place any small auto objects into registers, but objects explicitly declared as *register* will take precedence. At least two and as many as six registers are available. The number of registers actually used depends on what registers are needed for temporary values in the function.

3.5.2.1 Whether a plain `int` bit-field is treated as a signed `int` or as an unsigned `int` bit field.

Plain `int` bit fields are treated as **signed int** bit fields.

3.5.2.1 The order of allocation of bit fields within an `int`.

Bit fields are allocated from the low-order bit position to the high-order.

3.5.2.1 The padding and alignment of members of structures.

By default, no padding is used in structures. If you use the alignment option (**-a**), structures are padded to even size, and any members that do not have character or character array type will be aligned to an even offset.

3.5.2.1 Whether a bit-field can straddle a storage-unit boundary.

When alignment (**-a**) is not requested, bit fields can straddle word boundaries, but are never stored in more than two adjacent bytes.

3.5.2.2 The integer type chosen to represent the values of an enumeration type.

If all enumerators can fit in an **unsigned char**, that is the type chosen. Otherwise, the type is **signed int**.

3.5.3 What constitutes an access to an object that has volatile-qualified type.

Any reference to a volatile object will access the object. Whether accessing adjacent memory locations will also access an object depends on how the memory is constructed in the hardware. For special device memory, like video display memory, it depends on how the device is constructed. For normal PC memory, volatile objects are only used for memory that might be accessed by asynchronous interrupts, so accessing adjacent objects has no effect.

3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type.

There is no specific limit on the number of declarators. The number of declarators allowed is fairly large, but when nested deeply within a set of blocks in a function, the number of declarators will be reduced. The number allowed at file level is at least 50.

3.6.4.2 The maximum number of case values in a switch statement.

There is no specific limit on the number of cases in a switch. As long as there is enough memory to hold the case information, the compiler will accept them.

3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value.

All character constants, even constants in conditional directives use the same character set (execution). Single-character character constants will be negative if the character type is signed (default and **-K** not requested).

3.8.2 The method for locating includable source files.

For include file names given with angle brackets, if include directories are given in the command line, then the file is searched for in each of the include directories. Include directories are searched in this order: First, using directories specified on the command line, then using directories specified in `TURBOC.CFG`. If no include directories are specified, then only the current directory is searched.

3.8.2 The support for quoted names for includable source files.

For quoted file names, the file is first searched for in the current directory. If not found, Borland C++ searches for the file as if it were in angle brackets.

3.8.2 The mapping of source file name character sequences.

Backslashes in include file names are treated as distinct characters, not as escape characters. Case differences are ignored for letters.

3.8.8 The definitions for `__DATE__` and `__TIME__` when they are unavailable.

The date and time are always available, and will use the DOS date and time.

4.1.1 The decimal point character.

The decimal point character is a period (`.`).

4.1.5 The type of the `sizeof` operator, `size_t`.

The type `size_t` is **unsigned int**.

4.1.5 The null pointer constant to which the macro `NULL` expands.

An integer or a long 0, depending upon the memory model.

4.2 The diagnostic printed by and the termination behavior of the `assert` function.

The diagnostic message printed is "Assertion failed: *expression*, file *filename*, line *nn*", where *expression* is the asserted expression which failed, *filename* is the source file name, and *nn* is the line number where the assertion took place.

abort is called immediately after the assertion message is displayed.

4.3 The implementation-defined aspects of character testing and case mapping functions.

None, other than what is mentioned in 4.3.1.

4.3.1 The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint` and `isupper` functions.

First 128 ASCII characters.

4.5.1 The values returned by the mathematics functions on domain errors.

An IEEE NAN (not a number).

4.5.1 Whether the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow range errors.

No, only for the other errors—domain, singularity, overflow, and total loss of precision.

4.5.6.4 Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero.

No. `fmod(x, 0)` returns 0.

4.7.1.1 The set of signals for the signal function.

`SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`.

4.7.1.1 The semantics for each signal recognized by the signal function.

See the description of `signal` in the *Library Reference*.

4.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function.

See the description of `signal` in the *Library Reference*.

4.7.1.1 If the equivalent of `signal(sig, SIG_DFL)`; is not executed prior to the call of a signal handler, the blocking of the signal that is performed.

The equivalent of `signal(sig, SIG_DFL)` is always executed.

4.7.1.1 Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the signal function.

No, it is not.

4.9.2 Whether the last line of a text stream requires a terminating newline character.

No, none is required.

4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in.

Yes, they do.

4.9.2 The number of null characters that may be appended to data written to a binary stream.

None.

4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file.

The file position indicator of an append-mode stream is initially placed at the beginning of the file. It is reset to the end of the file before each write.

4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point.

A write of 0 bytes *may* or *may not* truncate the file, depending upon how the file is buffered. It is safest to classify a zero-length write as having indeterminate behavior.

4.9.3 The characteristics of file buffering.

Files can be fully buffered, line buffered, or unbuffered. If a file is buffered, a default buffer of 512 bytes is created upon opening the file.

4.9.3 Whether a zero-length file actually exists.

Yes, it does.

4.9.3 Whether the same file can be open multiple times.

Yes, it can.

4.9.4.1 The effect of the remove function on an open file.

No special checking for an already open file is performed; the responsibility is left up to the programmer.

4.9.4.2 The effect if a file with the new name exists prior to a call to rename.

`rename` will return a `-1` and `errno` will be set to `EEXIST`.

4.9.6.1 The output for %p conversion in fprintf.

In near data models, four hex digits (XXXX). In far data models, four hex digits, colon, four hex digits (XXXX:XXXX).

4.9.6.2 The input for %p conversion in fscanf.

See 4.9.6.1.

4.9.6.2 The interpretation of an – (hyphen) character that is neither the first nor the last character in the scanlist for a %[conversion in fscanf.

See the description of **scanf** in the *Library Reference*.

4.9.9.1 The value to which the macro errno is set by the fgetpos or ftell function on failure.

EBADF Bad file number.

4.9.10.4 The messages generated by perror.

Error 0	Invalid data
Invalid function number	No such device
No such file or directory	Attempted to remove current directory
Path not found	Not same device
Too many open files	No more files
Permission denied	Invalid argument
Bad file number	Arg list too big
Memory arena trashed	Exec format error
Not enough memory	Cross-device link
Invalid memory block address	Math argument
Invalid environment	Result too large
Invalid format	File already exists
Invalid access code	

See **perror** in the *Library Reference* for details.

4.10.3 The behavior of calloc, malloc, or realloc if the size requested is zero.

calloc and **malloc** will ignore the request. **realloc** will free the block.

4.10.4.1 The behavior of the abort function with regard to open and temporary files.

The file buffers are not flushed and the files are not closed.

4.10.4.3 The status returned by exit if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE.

Nothing special. The status is returned exactly as it is passed. The status is represented as a **signed char**.

4.10.4.4 The set of environment names and the method for altering the environment list used by getenv.

The environment strings are those defined in DOS with the SET command. **putenv** can be used to change the strings for the duration of the current program, but the DOS SET command must be used to change an environment string permanently.

4.10.4.5 The contents and mode of execution of the string by the system function.

The string is interpreted as a DOS command. COMMAND.COM is executed and the argument string is passed as a command to execute. Any DOS built-in command, as well as batch files and executable programs, can be executed.

4.11.6.2 The contents of the error message strings returned by strerror.

See 4.9.10.4.

4.12.1 The local time zone and Daylight Saving Time.

Defined as local PC time and date.

4.12.2.1 The era for clock.

Represented as clock ticks, with the origin being the beginning of the program execution.

4.12.3.5 The formats for date and time.

Borland C++ implements ANSI formats.

-
- ;
 - null statement *22, 99*
 - statement terminator *22, 99*
 - /* */ (comments) *7*
 - /**/ (token pasting) *7*
 - // (comments) *8*
 - operator
 - decrement *81, 84*
 - ? : operator
 - conditional expression *82, 94*
 - :: (scope resolution operator) *82, 108*
 - * and ->* operators (dereference pointers) *82, 97*
 - 1 command-line compiler option *403*
 - 87 environment variable *369*
 - \\ escape sequence (display backslash character) *15*
 - \" escape sequence (display double quote) *15*
 - \? escape sequence (display question mark) *15*
 - \' escape sequence (display single quote) *15*
 - : (labeled statement) *23*
 - != operator
 - huge pointer comparison and *345*
 - not equal to *82, 92*
 - && operator
 - logical AND *81, 93*
 - ++ operator
 - increment *81, 84*
 - << operator
 - put to *See overloaded operators, >> (put to)*
 - shift bits left *81, 89*
 - <= operator
 - less than or equal to *82, 91*
 - == operator
 - equal to *91*
 - huge pointer comparison and *345*
 - >= operator
 - greater than or equal to *82, 91*
 - >> operator
 - get from *See overloaded operators, << (get from)*
 - shift bits right *81, 89*
 - || operator
 - logical OR *81, 94*
 - > operator (selection) *82*
 - overloading *140*
 - structure member access *67, 83*
 - union member access *83*
 - * (pointer declarator) *23*
 - \ (string continuation character) *19*
 - ## symbol
 - overloading and *135*
 - preprocessor directives *80*
 - token pasting *7, 163*
 - ! operator
 - logical negation *81, 86*
 - % operator
 - modulus *81, 88*
 - remainder *81, 88*
 - & operator
 - address *81, 85*
 - bitwise AND *81, 92*
 - truth table *93*
 - position in reference declarations *39, 106*
 - * operator
 - indirection *81, 86*
 - pointers and *58*
 - multiplication *81, 88*
 - + operator
 - addition *81, 88*
 - unary plus *81, 86*
 - , operator
 - evaluation *82, 96*
 - function argument lists and *22*
 - operator
 - subtraction *81, 89*
 - unary minus *81, 86*

- / operator
 - division *81, 88*
 - rounding *88*
- < operator
 - less than *82, 90*
- = operator
 - assignment *81, 95*
 - compound *96*
 - overloading *139*
 - equal to *82*
 - initializer *23*
- > operator
 - greater than *82, 91*
- ^ operator
 - bitwise XOR *81, 93*
 - truth table *93*
- | operator
 - bitwise inclusive OR *81, 93*
 - truth table *93*
- ~ operator
 - bitwise complement *81, 86*
- . operator (selection) *82*
 - structure member access *67, 83*
- 1's complement (~) *81, 86*
- # symbol
 - conditional compilation and *166*
 - converting strings and *164*
 - null directive *159*
 - overloading and *135*
 - preprocessor directives *24, 80, 158*
- 80x87 coprocessors *See* numeric coprocessors
- 80x86 processors
 - address segment:offset notation *343*
 - inline assembly language and *403*
 - registers *340-342*
- _OvrInitEms (function) *364*
- _OvrInitExt (function) *364*

A

- A BCC option (ANSI keywords) *178*
- a BCC option (word alignment) *416, 417*
- a command-line option (word alignment) *69*
- \a escape sequence (audible bell) *15*
- abbreviations
 - CLASSLIB names and *228*
- abort (function)
 - destructors and *133*

- open and temporary files and *421*
- abstract classes *182, 216, See* classes, abstract
- abstract data types
 - BIDS class names *228*
 - class library and *224*
- AbstractArray class *242*
- access
 - classes *120-122*
 - base *120*
 - default *120*
 - derived *120*
 - qualified names and *121*
 - data members and member functions *118*
 - friend classes *121*
 - friend functions *118*
 - overriding *119*
 - structure members *67, 83, 119*
 - unions
 - members *83, 119*
 - objects *415*
 - volatile objects *417*
- accounting applications *372*
- active page
 - defined *390*
 - setting *389*
- adapters, video *See* video adapters
- add
 - Array member function *246*
 - Bag member function *250*
 - Btree member function *255*
 - Collection member function *259*
 - Dictionary member function *268*
 - DoubleList member function *268*
 - HashTable member function *273*
 - List member function *275*
 - Sets member function *286*
- addAt
 - Array member function *246*
- addAtHead
 - DoubleList member function *268*
- addAtTail
 - DoubleList member function *268*
- addition operator (+) *81, 88*
- address operator (&) *81, 85*
- addresses, memory *See* memory addresses
- adjustfield, ios data member *199*

- ADT
 - header files 230
- ADT (abstract data types) 224
- aggregate data types *See* data types
- alert (\a) 15
- algorithms
 - #include directive 165
- aliases *See* referencing and dereferencing
- alignment
 - bit fields and 417
 - structure members 416
 - word 69, 416, 417
- alloc.h (header file)
 - malloc.h and 309
- allocate, streambuf member function 207
- allocation, memory *See* memory, allocation
- ancestors *See* classes, base
- AND operator (&) 81, 92
 - truth table 93
- AND operator (&&) 81, 93
- angle brackets 165
- anonymous unions
 - member functions and 72
- ANSI
 - C standard
 - Borland C++ and 3
 - date and time formats 422
 - diagnostics 411
 - extended character sets 412
 - implementation-specific items 411-422
 - integer values 413
 - keywords 9
 - predefined macro 178
 - main function
 - semantics of arguments to 412
 - multibyte characters 413
- argsused pragma 171
- arguments *See also* parameters
 - actual
 - calling sequence 64
 - command line *See* command-line compiler
 - conversions 64
 - converting to strings 164
 - default
 - constructors and 124, 126
 - to #define directive 162
 - fmod function and 419
 - function calls and 64
 - functions taking none 62
 - matching number of 65
 - parameters vs. 3
 - passing
 - C-language style 48
 - type checking 61
 - variable number of 23
 - Pascal and 50
- arithmetic, pointers *See* pointers, arithmetic
- arithmetic types 40
- Array class 245
- ArrayIterator, AbstractArray friend 245
- ArrayIterator class 247
- arrays 59
 - classes for 242, 245
 - classes for sorted 289
 - of classes
 - initializing 110
 - constructors for
 - order of calling 129
 - delete operator and 109
 - elements
 - comparing 90
 - indeterminate 59
 - structures and 60
 - initialization 43, 44
 - integer types for 415
 - pointers to 416
 - multidimensional 59
 - new operator and 109
 - sizeof and 87
 - subscripts 21, 82
 - overloading 140
- arraySize, AbstractArray member function 243
- ascending sort 289
- ASCII codes
 - extended character sets 412
- asm (keyword) 400
 - braces and 401
 - how to use 98
- .ASM files *See* assembly language
- asm statement
 - inline pragma and -B BCC option and 173
- aspect ratio
 - determining current 397
 - setting 389

- assembler
 - built in 399
- assembly language *See also* opcodes
 - built-in assembler *See* built-in assembler
 - huge functions and 52, 175
 - inline 399
 - 80186 instructions 403
 - braces and 401
 - C structure members and 405
 - restrictions 406
 - calling functions 404
 - commenting 401
 - directives 404
 - floating point in 370
 - goto in 406
 - inline pragma and 173
 - jump instructions 403, 406
 - option (-B) 173, 399
 - referencing data in 404
 - register variables in 404
 - semicolons and 401
 - size overrides in 404
 - syntax 400
 - variable offsets in 404
 - routines
 - overlays and 363
 - statement syntax 98
- assert (function)
 - message and behavior 418
- assertion macros 240
- assignment operator
 - overloading 139
- assignment operator (=) 81, 95
 - compound 96
- Association class 219, 248
 - example program 240
- associativity 78, *See also* precedence
 - expressions 76
- asterisk (*) 23
- atexit (function)
 - destructors and 133
- attach
 - filebuf member function 196
 - fstreambase member function 198
- attributes
 - cell
 - blink 383
 - colors 382
 - control functions 379
 - screen cells 375, 382
- auto (keyword) 45
 - class members and 113
 - external declarations and 36
 - register keyword and 30
- automatic objects 30, *See* objects, automatic
- auxiliary carry flag 342
- AX register 340

B

- B BCC option (inline assembler code) 399
 - inline pragma and 173
- b command-line option (enumerations) 73
- \b escape sequence (backspace) 15
- background color *See* graphics, colors, background
- backslash character
 - hexadecimal and octal numbers and 14
 - line continuation 164
- backslash character (\) 15
- backspace character (\b) 15
- bad, ios member function 200
- Bag class 250
- banker's rounding 374
- base, streambuf member function 207
- base address register 341
- base classes *See* classes
- _based (keyword) 310
- BaseDate class 252
- basefield, ios data member 199
- BaseTime class 253
- BASM (built-in assembler) *See* built-in assembler
- BCD 372
 - converting 373
 - number of decimal digits 373
 - range 373
 - rounding errors and 373
- __BCPLUSPLUS__ macro 175
- bell (\a) 15
- BGI OBJ (graphics converter)
 - initgraph function and 387
- BI_ prefix
 - class names 228
- BIDS *See* Borland International Data Structures

BIDS template library 224
binary coded decimal *See* BCD
binary operators *See* operators
binary streams

 null characters and 420

BIOS

 video output and 383

bit fields

 alignment and 417

 hardware registers and 71

 how treated 416

 integer 71

 order of allocation 416

 portable code and 71

 structures and 70

 unions and 73

bit images

 functions for 389

bit-mapped fonts *See* fonts

bitalloc, ios member function 200

bits

 blink enable 379

 color 379

 shifting 81, 89

bitwise

 AND operator (&) 81, 92

 truth table 93

 complement operator (~) 81, 86

 operators

 signed integers and 415

 OR operator (|) 81, 93

 truth table 93

 XOR operator (^) 81, 93

 truth table 93

blen, streambuf member function 207

blink enable bit 379

block

 scope 28

 statements 98

Boolean data type 99

Borland C++

 ANSI implementation-specific items 411-422

 converting to from Microsoft C 299-311

 extensions 9

Borland International Data Structures (BIDS)

 224

__BORLANDC__ macro 176

bp, ios data member 199

BP register 341

 overlays and 363

braces 21

 asm keyword and 401

brackets 21, 82

 overloading 140

break statements 103

 loops and 103

Btree class 255

Btreeiterator class 257

buffered files 420

buffers

 C++ streams and 196, 197

 overlays

 default size 362

built-in assembler 399, 400

BX register 340

C

C++ 105-145

 C code and 176

 classes *See* classes

 comments 8

 complex numbers *See* complex numbers

 constants *See* constants

 constructors *See also* constructors

 conbuf 194

 constream 195

 filebuf 196

 fstream 197

 fstreambase 197, 198

 ifstream 189, 198, 199

 ios 199

 iostream 202

 iostream_withassign 202

 istream 203

 istream_withassign 204

 istrstream 204

 ofstream 189, 205

 ofstream_withassign 206

 ostrstream 206, 207

 streambuf 207

 strstream 211

 strstreambase 210

 strstreambuf 210

- conversions *See* conversions, C++
- data members *See* data members
- declarations *See* declarations
- destructors *See* destructors
- DLLs and 336
- enumerations *See* enumerations
- file operations *See* files
- fill characters 186
- floating-point precision 186
- for loops *See* loops, for, C++
- formatting *See* formatting, C++
- Fourier transforms example 371
- functions
 - C functions and 32
 - friend 113
 - access 118
 - inline *See* functions, inline
 - name mangling and 32
 - pointers to 55
 - taking no arguments 62
 - virtual 140
 - pure keyword and 142
- inheritance *See* inheritance
- initializers 45
- iterators *See* iterators
- keywords 9
- member functions *See* member functions
- members *See* data members; member
 - functions
- name spaces 69
- operators *See* operators, C++; overloaded operators
- output *See* output, C++
- parameters *See* parameters
- referencing and dereferencing *See* referencing and dereferencing
- scope *See* scope
- streams *See* streams, C++
- structures *See* structures
- this
 - nonstatic member functions and 113
 - static member functions and 115
- unions *See* unions
- visibility *See* visibility

C language

- argument passing 48
- C++ code and 176
 - calling conventions 176, 178
- C prefix
 - class names 228
- calling conventions *See also* parameters, passing; Pascal
- calloc (function)
 - zero-size memory allocation and 421
- calls
 - far, functions using 52
 - near, functions using 52
- carriage return character 15
- carry flag 342
- case
 - preserving 50
 - sensitivity
 - external identifiers and 413
 - forcing 48
 - global variables and 48
 - identifiers and 10
 - pascal identifiers and 11
 - statements *See* switch statements
- cast expressions
 - syntax 85
- `__CDECL__` macro 176
- `cdecl` (keyword) 48, 50
 - function modifiers and 52
- `_cdecl` (keyword)
 - Microsoft C 310
- cells, screen *See* screens, cells
- characters
 - blinking 383
 - colors 382, 383
 - constants *See* constants, character
 - data type char *See* data types, char
 - decimal point 418
 - fill
 - setting 186
 - in screen cells 375
 - intensity
 - setting 379
 - multibyte 413
 - newline
 - inserting 186
 - text streams and 420
- null
 - binary stream and 420

- sets
 - execution 412
 - collation sequence 412
 - number of bits in 413
 - source and 414
 - extended 412
 - for character constants 417
 - testing for 419
- unsigned char data type
 - range 19
- whitespace
 - extracting 186
- wide 414
- CHECK macro 240
- .CHR files *See* fonts, files
- circles
 - roundness of 389
- CL options
 - command-line compiler options and 302
- class templates 227
- _CLASSDEF (Windows DLL compatibility) 242
- classes 111-124, *See also* C++; individual class
 - names; inheritance
 - abstract 142, 182
 - abstract vs instance 216
 - access 120-122
 - default 120
 - qualified names and 121
 - arrays
 - sorted 289
 - arrays of
 - initialization 110
 - auto keyword and 113
 - base
 - calling constructor from derived class 130
 - constructors 131
 - pointers to
 - destructors and 134
 - private
 - friend keyword and 121
 - protected keyword and 120
 - unions and 120
 - virtual 122
 - constructors and 128
 - class names and 112
 - collections 223
 - container *See* container class library
 - data types and 38
 - date and time 264, 294
 - debugging modes 241
 - declarations
 - incomplete 112
 - derived
 - base class access and 120
 - calling base class constructor from 130
 - constructors 131
 - DLLs and 336
 - extern keyword and 113
 - friends 122-124
 - access 121
 - hierarchies 216
 - ios family 183
 - object-based 218
 - streambuf 182
 - traversing 247
 - initialization *See* initialization, classes
 - lists 268
 - member functions *See* member functions
 - members, defined 113
 - naming *See* identifiers
 - objects 111, 113
 - initialization *See* initialization, classes, objects
 - priority queues 282
 - queue 284
 - queues
 - double-ended 265
 - register keyword and 113
 - scope *See* scope, classes
 - sequence 223, 265, 284, 289
 - rules for 223
 - sizeof operator and 87
 - sortable objects 286
 - stack 289
 - streams and 181
 - files 181
 - formatted I/O 182
 - memory buffers 181, 182
 - strings 181
 - string 292
 - syntax 111
 - unions and 73
 - CLASSLIB naming conventions 228

- `_CLASSTYPE` (Windows DLL compatibility) 242
- `_clear87` (function)
 - floating point exceptions and 370
- `clear`, ios member function 200
- clipping, defined 391
- `clock` (function)
 - era 422
- `close`
 - `filebuf` member function 196
 - `fstreambase` member function 198
- `creol`, `conbuf` member function 194
- `clrscr`
 - `conbuf` member function 194
 - `constream` member function 195
- Code Generation dialog box 69
- code models *See* memory models
- code segment 342
 - storing virtual tables in, `-WD` option and 324
- Collection class 223, 258
- collections
 - ordered 224
 - random access to 242
 - unordered 224
 - Bag class 250
 - Dictionary class 267
 - DoubleList class 268
 - HashTable class 272
 - List class 275
 - Set class 285
- colons 23
- color *See* graphics, colors
- Color/Graphics Adapter (CGA) *See also*
 - graphics; graphics drivers; video adapters
 - background and foreground colors 394
 - color palettes 393, 394
 - resolution 393
 - high 394
- colors *See* graphics, colors
- .COM files
 - memory models and 346
- COMDEFs
 - generating 309
- comma
 - operator 82, 96
 - separator 22

- command-line compiler
 - compiling and linking with
 - Windows applications 320
 - DLLs and 332
 - INCLUDE environment variable and 300
 - LIB environment variable and 300
 - nested comments 8
 - options
 - `±1` (80186 instructions) 403
 - `-b` (enumerations) 73
 - `-Wx` (Windows applications) 322
 - alignment (`-a`) 69
 - bit fields and 417
 - ANSI diagnostics and 411
 - ANSI keywords (`±A`) 178
 - assembly language and 399
 - `-B` (inline assembler code) 399
 - inline pragma and 173
 - changing from within programs 173
 - CL options versus 302
 - compatibility 306
 - data segment
 - name 352
 - define identifiers (`-D`) 161
 - .DLLs with all exportables (`-WD`) 324
 - DLLs with explicit exports (`-WDE`) 324
 - enumerations (`-b`) 73
 - far objects (`-zE`
 - `-zF`
 - and `-zH`) 352
 - floating point
 - code generation (`-f87`) 368
 - emulation (`-f`) 368
 - fast (`-ff`) 368
 - inline assembler code (`±B`)
 - inline pragma and 173
 - inline assembler code (`-B`) 399
 - .OBJs with explicit exports (`-WN`) 323
 - overlays (`-Y`) 178, 361
 - overlays (`-Yo`) 360
 - Pascal calling conventions (`-p`) 50, 51, 176, 178
 - pragmas for 173
 - smart callbacks (`-WS`) 323
 - undefine (`-U`) 161
 - `-WDE` (DLLs with explicit exports) 324
 - Windows applications (`-W`) 319, 322, 325

- word alignment (-a) 69
- Y (overlays) 178, 361
- zX (code and data segments) 352
- Windows and 318
- commas
 - nested
 - macros and 163
- comments 7
 - // 8
 - /**/ 7
 - as whitespace 6
 - inline assembly language code 401
 - nested 7
 - token pasting and 7
 - whitespace and 8
- __COMPACT__ macro 176
- compact memory model *See* memory models, compact
- compatibility
 - command-line options 306
 - with Microsoft C 299-311
- compilation
 - speeding up 172, 173
 - Windows applications 320
- compiling
 - conditional
 - # symbol and 166
- complement
 - bitwise 81, 86
- complex declarations *See* declarations
- complex.h (header file)
 - complex numbers and 371
- complex numbers
 - << and >> operators and 371
 - C++ operator overloading and 371
 - example 371
 - header file 371
 - using 371
- component selection *See* operators, selection (. and ->)
- compound assignment operators 96
- conbuf (class) 194
- concatenating strings *See* strings, concatenating
- condFuncType definition 279
- conditional compilation
 - # symbol and 166
 - __cplusplus macro and 176
- conditional operator (? :) 94
- conforming extensions 3
- conio.h (header file)
 - console control and 377
 - constream and 192
- console
 - I/O
 - functions 377
- Console stream manipulators 192
- const (keyword) 47
 - C++ and 47
 - formal parameters and 64
 - pointers and 47, 57
- constant expressions 20
- constants 11, 47, *See also* numbers
 - Borland C++ 15
 - C++ 47
 - case statement
 - duplicate 100
 - character 12, 14
 - character set 417
 - extending 15
 - integer and 42
 - two-character 15
 - values 414
 - wide 16, 414
 - data types 13
 - decimal 11, 12
 - data types 13
 - suffixes 13
 - enumerations *See* enumerations
 - expressions *See* constant expressions
 - floating point 12, 16
 - data types 16
 - negative 16
 - ranges 17
 - fractional 12
 - hexadecimal 12, 13
 - integer 11, 12
 - internal representations of 19
 - manifest 175
 - null pointer
 - NULL macro and 418
 - octal 12
 - pointers and 57
 - string *See* strings, literal
 - suffixes and 13

- syntax 12
 - ULONG_MAX and UINT_MAX 89
- constrea.h 192
- constrea.h: 193
- constream (class) 195
- constructors 124-129, *See also* initialization
 - AbstractArray member function 243
 - Array member function 246
 - ArrayIterator member function 247
- arrays
 - order of calling 129
- Association member function 248
- Bag member function 250
- base class
 - calling
 - from derived class 130
 - order 131
- Basedate member function 252
- Basetime member function 253
- Btree member function 255
- BtreeIterator member function 257
- calling 125
- class initialization and 129
- classes
 - virtual base 128
- Collection member function 259
- Container member function 261
- copy 127
 - class object initialization and 129
- Date member function 264
- default arguments and 124, 126
- default parameters 126
- defaults 126
- delete operator and 125
- derived class
 - order of calling 131
- Dictionary member function 268
- DoubleList member function 269
- DoubleListIterator member function 270
- HashTable member function 273
- HashTableIterator member function 274
- inheritance and 124
- invoking 125
- List member function 275
- ListIterator member function 276
- new operator and 125
- non-inline
 - placement of 132
- Object member function 279
- order of calling 128
- overloaded 127
- Sets member function 286
- String member function 292
- Time member function 294
- Timer member function 295
- TShouldDelete member function 296
- unions and 125
- virtual 124
- consumer (streams) 181
- container class library
 - directories 238
 - examples 240
 - INCLUDE 238
 - lib 239
 - source 239
 - example programs 240
 - makefile and 239
 - memory models and 239
 - reference section 241
- container classes 218, 219, 259
 - functions of 260
- container hierarchy
 - object-based 216
- ContainerIterator class 263
 - containers and 263
 - hierarchy 222
- containers
 - basics 216
 - ContainerIterator class and 263
 - direct 228
 - elements of 260
 - equality testing 260
 - flushing 221, 260
 - implementation 225
 - indirect 228
- continue statements 103
 - loops and 103
- continuing lines 6, 19, 164
- _control87 (function)
 - floating point exceptions and 370
- control lines *See* directives
- conversions 41

- argument *See* arguments, conversions
- arguments to strings 164
- arrays 60
- BCD 373
- C++ 186
 - setting base for 186
- character
 - integers and 42
- decimal 186
- floating point
 - to smaller floating point 415
- hexadecimal 186
- integers
 - character and 42
 - to floating point 415
 - to pointers 415
- octal 186
- pointers 59
 - to integers 415
- sign extension and 42
- special 42
- standard 42
- when value can't be represented 414
- coordinates
 - origin 377
 - returning 380
 - starting positions 376, 380
- coprocessors *See* numeric coprocessors
- copy constructors *See* constructors, copy
- __cplusplus macro 176
- CPP.EXE (preprocessor) 157
- CPU (central processing unit) *See* 80x86 processors
- _cs (keyword) 48, 350
- CS register 342, 344
- current
 - ArrayIterator member function 247
 - BtreeIterator member function 257
 - ContainerIterator member function 263
 - DoubleListIterator member function 270
 - HashTableIterator member function 274
 - ListIterator member function 277
- current position, files *See* file-position indicator
- cursor
 - changing 380
 - control
 - header file 377

- manipulating onscreen 378
- position
 - setting 378
- CX register 341

D

- D BCC option (define identifier) 161
- \D escape sequence (display a string of octal digits) 15
- data
 - static, DLLs and 336
- data members *See also* member functions
 - access 118
 - dereference pointers 82, 97
 - private 118
 - protected 118
 - public 118
 - scope 116-119
 - static 115
 - declaration 116
 - definition 116
 - uses 116
- data models *See* memory models
- data segment
 - naming and renaming 352
 - removing virtual tables from, -WD option and 324
- data segments 342
- data structures *See* structures
- data type
 - template argument 147
- data types 25, *See also* constants; floating point; integers; numbers
 - aggregate 38
 - arithmetic 40
 - BCD *See* BCD
 - Boolean 99
 - C++ streams and 184, 188
 - char 40
 - range 19
 - signed and unsigned 15, 40
 - classes and 38
 - conversions *See* conversions
 - declarations 39
 - declaring 38
 - default 38
 - derived 38

- enumerations *See* enumerations
 - range 19
- function return types 61
- fundamental 38, 39
 - creating 40
- identifiers and 26, 27
- integers *See* integers
- integral 40
- internal representations 40
- memory use 87
- new, defining 46
- parameterized *See* templates
- ranges 19
- scalar 38
 - initializing 43
- size_t 87, 137, 138
- sizeof operator 418
- table of 19
- taxonomy 38
- text_modes 381
- types of 38
- unsigned char
 - range 19
- void 39
- wchar_t 16
- date *See also* time
 - formats 422
 - local
 - how defined 422
 - macro 177
- __DATE__ macro 177
 - availability 418
 - #define and #undef directives and 162
- Date class 264
- dates
 - class 264
- Day
 - Basedate member function 252
- deallocation, memory *See* memory, allocation
- __DEBUG macro 240, 241
- debugging
 - overlays 362
- dec (manipulator) 186
- decimal constants *See* constants, decimal
- decimal point
 - how displayed 418
- declarations 25
- arrays 59
- C++ 38
 - incomplete 112
- complex 53
 - examples 53, 54
- data types 38
 - default 38
- defining 26, 31, 33, 44
 - extern keyword and 45
- examples 39
- external 31, 36
 - storage class specifiers and 36
- function *See* functions, declaring
- incomplete class 112
- with initializers
 - bypassing 103
- mixed languages 50
- modifiers and 47
- objects 34
- Pascal 50
- point of 143
- pointers 56
- referencing 26, 33
 - extern keyword and 45
- simple 44
- static data members 116
- structures *See* structures, declaring
- syntax 33, 34
- tentative definitions and 33
- unions 73
- declarators
 - number of 417
 - pointers (*) 23
 - syntax 54
- decrement operator (--) 81, 84
- decrNofKeys
 - Btree member function 255
- default (label)
 - switch statements and 100
- default constructors *See* constructors, default
- #define directive 159
 - argument lists 162
 - global identifiers and 162
 - keywords and 162
 - redefining macros with 160
 - with no parameters 159
 - with parameters 162

- defined operator 167
- defining declarations *See* declarations, defining
- definitions *See* declarations, defining
 - function *See* functions, definitions
 - tentative 33
- delete
 - Error member function 271
- delete (operator) 108
 - arrays and 109
 - constructors and destructors and 125
 - destructors and 132, 133
 - dynamic duration objects and 30
 - overloading 137
 - pointers and 132
- delline, conbuf member function 194
- delObj
 - TShouldDelete member function 296
- delta
 - AbstractArray data member 242
- Deque class 265
- dereferencing *See* referencing and dereferencing
- derived classes *See* classes
- derived data types *See* data types
- descendants *See* classes, derived
- destroy
 - AbstractArray member function 243
 - Collection member function 259
- destroyFromHead
 - DoubleList member function 269
- destroyFromTail
 - DoubleList member function 269
- destructors 124, 132-135, *See also* initialization
 - abort function and 133
 - atexit function and 133
 - base class pointers and 134
 - calling 125
 - class initialization and 129
 - delete operator and 125, 132, 133
 - exit function and 133
 - global variables and 133
 - inheritance and 124
 - invoking 125, 132
 - explicitly 133
 - new operator and 125, 133
 - pointers and 132
 - #pragma exit and 133
 - unions and 125
 - virtual 124, 134
- detach
 - AbstractArray member function 243
 - Bag member function 250
 - Btree member function 255
 - Collection member function 259
 - DoubleList member function 269
 - HashTable member function 273
 - List member function 275
 - SortedArray member function 289
- detachFromHead
 - DoubleList member function 269
- detachFromTail
 - DoubleList member function 269
- detachLeft
 - PriorityQueue member function 283
- DI register 341
- diagnostic messages
 - ANSI 411
- Dictionary class 267
 - example program 240
- digits
 - hexadecimal 12
 - nonzero 12
 - octal 12
- dir.h (header file)
 - direct.h and 309
- direct and indirect data structures 225
- direct.h (header file)
 - dir.h and 309
- direct member selector *See* operators, selection (. and ±>)
- direct video output 383
- direction flag 342
- directives 167, 157-179, *See also* individual
 - directive names; macros
 - ## symbol
 - overloading and 135
 - # symbol 24
 - overloading and 135
 - conditional 167
 - nesting 167
 - conditional compilation and 166
 - error messages 170
 - keywords and 162
 - line control 169

- Microsoft compatibility 308
- pragmas *See* pragmas
- sizeof and 87
- syntax 158
- usefulness of 157
- directories
 - container class library 238
 - include files
 - how searched 418
- DIRECTRY (container class library example program) 240
- division operator (/) 81, 88
 - rounding 88
- __DLL__ macro 177
- DLLs
 - building 313-338
 - C++ and
 - classes 336
 - mangled names 338
 - compiler options and 336
 - compiling and linking 332
 - creating 324, 333
 - defined 332
 - exit point 334
 - initialization functions 334
 - LibMain function and 334
 - libraries 330, 331
 - linking
 - Resource Compiler and 331
 - macro 177
 - memory models 331
 - memory models and 326
 - pointers and 335
 - smart callbacks and 324
 - startup files 331
 - static data 336
 - virtual tables and 336
 - WEP function 334
- do while loops *See* loops, do while
- doallocate, ostreambuf member function 211
- domain errors
 - mathematics functions and 419
- DOS
 - environment
 - 87 variable 369
 - strings
 - changing permanently 422

- dot operator (selection) *See* operators, selection
 - (. and ±>)
- double quote character
 - displaying 15
- DoubleList class 268
- DoubleListIterator class 270
- drawing color *See* graphics, colors
- drawing functions 387
- _ds (keyword) 48, 350
- DS register 342, 344
- duplicate case constants 100
- duration 29
 - dynamic
 - memory allocation and 30
 - local
 - scope and 30
 - pointers 56
 - static 29
- DX register 341
- dynamic duration
 - memory allocation and 30
- dynamic memory allocation *See* memory, allocation

E

- eback, ostreambuf member function 207
- ebuf, ostreambuf member function 208
- egptr, ostreambuf member function 208
- elaborated type specifier 112
- elements
 - ordering definition 228
 - parsing 6
- #elif directive 167
- ellipsis (...) 23
 - prototypes and 62, 65
- #else directive 167
- __emit__() 310
- _emit (keyword) 310
- empty statements 99
- empty strings 18
- emulating the 80x87 math coprocessor *See*
 - floating point, emulating
- enclosing block 28
- #endif directive 167
- endl (manipulator) 186
- ends (manipulator) 186

- Enhanced Graphics Adapter (EGA) *See also*
 - graphics drivers; video adapters
 - color control on 395
- enum (keyword) *See* enumerations
- enumerations 73
 - C++ 74
 - class names and 112
 - command-line option (-b) 73
 - constants 12, 17, 74
 - default values 17
 - conversions 42
 - default type 73
 - name space 28
 - range 19
 - scope, C++ 75
 - structures and
 - name space in C++ 69
 - tags 74
 - name spaces 75
 - values 417
- environment
 - DOS
 - 87 variable 369
 - variables 300
 - Resource Compiler and 301
- eof, ios member function 200
- epptr, streambuf member function 208
- equal to operator (=) 82
- equal-to operator (==) 91
- equality operators *See* operators, equality
- era, clock function and 422
- Error class 218, 271
- #error directive 170
- errors
 - domain
 - mathematics functions and 419
 - expressions 79
 - floating point
 - disabling 370
 - graphics, functions for handling 395
 - math, masking 370
 - messages
 - assert function 418
 - graphics 396
 - perror function 421
 - strerror function 422
 - out of memory 339
 - preprocessor directive for 170
 - underflow range
 - mathematics functions and 419
- _es (keyword) 48, 350
- ES register 342
- escape sequences 12, 14
 - length 14
 - number of digits in 14
 - octal
 - non-octal digits and 15
 - source files and 418
 - table of 15
- evaluation order *See* precedence
- examples directory
 - container class library 240
- exclusive OR operator (^) 81, 93
 - truth table 93
- execution character sets *See* characters, sets, execution
- exit (functions) 422
 - destructors and 133
- exit pragma 171
- exit procedure, Windows 334
- expanded memory *See* extended and expanded memory
- exponents 12
- _export (keyword) 48, 52
 - Windows applications and 323, 324
- expressions
 - associativity 76
 - cast, syntax 85
 - constant 20
 - conversions and 41
 - decrementing 84
 - empty (null statement) 22, 99
 - errors and overflows 79
 - floating point
 - precedence 79
 - function
 - sizeof and 87
 - grouping 21
 - incrementing 84
 - precedence 76, 78
 - statements 22, 99
 - syntax 77
 - table 77

extended and expanded memory

 _OvrInitEms and 364

 _OvrInitExt and 364

 overlays and 364

 swapping 364

extensions 9

extent *See* duration

extern (keyword) 45, *See also* identifiers,

 external

 arrays and 59

 class members and 113

 const keyword and 47

 linkage and 31

 name mangling and 32

external

 declarations 31

 identifiers *See* identifiers, external

 linkage *See* linkage

extra segment 342

extraction operator (<<) *See* overloaded

 operators, << (get from)

extractors *See* input, C++

F

-f87 command-line compiler option (generate floating-point code) 368

-f command-line compiler option (emulate floating point) 368

\f escape sequence (formfeed) 15

fail, ios member function 200

far

 calls

 memory model and 361

 requirement 361

 functions *See* functions, far

 objects *See* objects, far

 pointers *See* pointers, far

far (keyword) 48, 344, 350, 356

_FAR (Windows DLL compatibility) 242

_fastcall (keyword) 48, 52

fd, filebuf member function `int fd()` 196

FDS

 header files 230

FDS (fundamental data structures) 224

-ff command-line compiler option (fast floating point) 368

fgetpos (function)

 errno value on failure of 421

field width *See* formatting, width (C++)

__FILE__ macro 177

 #define and #undef directives and 162

file descriptor 196

file-position indicator

 initial position 420

file scope *See* scope

filebuf (class) 196

files *See also* individual file-name extensions

 appending

 file-position indicator and 420

 .ASM *See* assembly language

 buffering 420

 buffers

 C++ 196, 197

 current

 macro 177

 font *See* fonts

 graphics driver, linking 387

 header *See* header files

 include *See* include files

 including 165

 names

 searching for 418

 open

 abort function and 421

 remove function and 420

 opening

 default mode 190

 multiple times 420

 project

 graphics library listed in 384

 renaming

 preexisting file name and 420

 scope *See* scope

 source

 escape sequences and 418

 startup

 DLLs and 331

 streams

 C++ operations 197

 temporary

 abort function and 421

 truncation while writing to 420

 zero-length 420

- fill, ios member function 200
- fill characters
 - C++ 186, 187
- filling functions 387
- financial applications 372
- findmember
 - Bag member function 250
 - Btree member function 255
 - Collection member function 259
 - HashTable member function 273
- firstThat
 - Bag member function 250
 - Container member function 261
 - Object member function 279
- flags
 - format state *See* formatting, format state
 - flags (C++)
 - ios (class)
 - setting 186
 - ios member function 200
 - register 340, 341
- floatfield, ios data member 199
- floating point *See also* data types; integers;
 - numbers
 - arithmetic
 - interrupt functions and 407
 - constants *See* constants
 - conversions *See* conversions
 - decimal point character 418
 - double
 - range 19
 - emulating 368
 - exceptions
 - disabling 370
 - expressions
 - precedence 79
 - fast 368
 - formats 414
 - libraries 367
 - long double
 - range 19
 - Microsoft C and 310
 - precision
 - setting 186
 - ranges 19
 - registers and 370
 - using 367
- flow-control statements *See* if statements;
 - switch statements
- flush
 - Bag member function 250
 - Btree member function 256
 - Container member function 261
 - Deque member function 266
 - DoubleList member function 269
 - HashTable member function 273
 - List member function 275
 - ostream member function 206
 - PriorityQueue member function 283
 - Stacks member function 291
- flush (manipulator) 186
- fmod (function)
 - second argument of zero 419
- fonts
 - bit-mapped
 - stroked vs. 391
 - when to use 391
 - clipping 391
 - files
 - loading and registering 391
 - height and width 391
 - information on current settings 398
 - registering 392
 - setting size 391
 - stroked
 - advantages of 391
- for loops *See* loops, for
- forEach
 - Bag member function 251
 - Container member function 261
 - Object member function 280
- foreground color *See* graphics, colors,
 - foreground
- formal parameters *See* parameters, formal
- format state flags *See* formatting, C++, format
 - state flags
- formatting
 - C++
 - classes for 182
 - fill character 186, 187
 - format state flags 184
 - I/O 186, *See also* manipulators
 - output 184
 - padding 187

- width functions *See also* manipulators
 - setting 186
- streams and
 - clearing 186
- formatting flags 199
- formfeed character 15
- fortran (keyword) 310
 - _pascal keyword and 310
- forward references 26
- Fourier transforms
 - complex number example 371
- FP_OFF 355
- FP_SEG 355
- fprintf (function)
 - %p conversion output 421
- free
 - MemBlocks member function 278
- free (function)
 - delete operator and 108
 - dynamic duration objects and 30
- freeze, strstreambuf member function 211
- friend (keyword) 113, 122-124
 - base class access and 121
 - functions and *See* C++, functions, friend
- fscanf (function)
 - %p conversion input 421
- fstream (class) 197
- fstreambase (class) 197
- ftell (function)
 - errno value on failure of 421
- function call operator *See* parentheses
- function operators *See* overloaded operators
- function template 148
- functions 60-65
 - arguments
 - no 62
 - attribute control 379
 - calling 64, *See also* parentheses
 - in inline assembly code 404
 - operators () 83
 - overloading operator for 140
 - rules 64
- cdecl and 51
- class names and 112
- color control 392
- comparing 92

- console
 - I/O 377
- declaring 60, 61
 - as near or far 352
- default types for memory models 52
- definitions 60, 63
- drawing 387
- duration 30
- error-handling, graphics 395
- exit 171
- export
 - Windows applications and 322, 323
- exporting 324
- external 45
 - declarations 31
- far 52
 - declaring 353
 - memory model size and 352
- filling 387
- friend *See* C++, functions, friend
- graphics *See also* graphics
 - drawing operations 387
 - fill operations 388
 - using 384-398
- graphics system control 385
- huge 52
 - assembly language and 52
 - _loads and 52
 - saving registers 175
- image manipulation 389
- inline
 - assembly language *See* assembly language,
 - inline
 - C++ 114
 - linkage 115
- internal linkage 46
- interrupt *See* interrupts, functions
- linking C and C++ 32
- main 60
- mathematical
 - domain errors 419
 - underflow range errors 419
- member *See* member functions
- memory
 - models and 48
- mode control 379
- name mangling and 32

- near 52
 - declaring 353
 - memory models and 352
- no arguments 39
- not returning values 39
- operators *See* overloaded operators
- overloaded *See* overloaded functions
- Pascal
 - calling 50
- pixel manipulation 389
- pointers 55
 - calling overlaid routines 362
 - object pointers vs. 54
- pointers to
 - void 55
- prototypes *See* prototypes
- recursive
 - memory models and 352
- return statements and 104
- return types 61
- scope *See* scope
- screen manipulation 389
- sizeof and 87
- startup 171
- state queries 380, 396
- static 31
- stdarg.h header file and 62
- storage class specifiers and 32
- structures and 67
- text
 - manipulation 377
 - output
 - graphics mode 390
- type
 - modifying 52
- viewport manipulation 389
- Windows 324
- windows 379
- fundamental data structure
 - class templates 227
- fundamental data structures
 - class library and 224
 - Object-based classes 229
- fundamental data types *See* data types

G

- gbump, streambuf member function 208

- gcount, istream member function 203
- generic pointers 39, 56
- get
 - istream member function 203
 - PriorityQueue member function 283
 - Queue member function 285
- get from operator (>>) *See* overloaded operators, >> (get from)
- getenv (function)
 - environment names and methods 422
- getItemsInContainer
 - Bag member function 251
 - Container member function 261
 - Deque member function 266
 - PriorityQueue member function 283
 - Stacks member function 291
- getLeft
 - Deque member function 266
- getline, istream member function 203
- getRight
 - Deque member function 266
- global identifiers *See* identifiers, global
- global variables 28, *See also* variables
 - case sensitivity and 48
 - destructors and 133
 - _ovrbuffer 358, 362
 - underscores and 48
 - _wscroll 378
- good, ios member function 200
- goto statements 103
 - assembly language and 406
- labels
 - name space 28
- gotoxy, conbuf member function 194
- gptr, streambuf member function 208
- grammar
 - tokens *See* tokens
- graphics *See also* graphics drivers
 - buffers 390
 - circles
 - aspect ratio 389
 - colors *See also* graphics, palettes
 - background 379
 - CGA 394
 - defined 382, 393
 - list 383
 - setting 379

- CGA 393, 394
- drawing 393
- EGA/VGA 395
- foreground 379
 - CGA 394
 - defined 382
 - list 383
 - setting 379
- functions 392
- information on current settings 398
- coordinates *See* coordinates
- default settings
 - restoring 386
- displaying 393
- drawing functions 387
- errors
 - functions to handle 395
- fill
 - operations 388
 - patterns 388
 - using 397
- functions
 - using 384-398
- header file 384
- library 384
- line style 388
- memory for 387
- page
 - active
 - defined 390
 - setting 389
 - visual
 - defined 390
 - setting 389
- palettes *See also* graphics, colors
 - defined 392
 - functions 392
 - information on current 398
- pixels *See also* screens, resolution
 - colors
 - current 398
 - functions for 389
 - setting color of 392
 - setting
 - clearing screen and 390
 - state queries 396

- system
 - control functions 385
 - shutting down 386
 - state queries 397
- text and 390
- viewports
 - defined 377
 - functions 389
 - information on current 398
- graphics drivers *See also* Color/Graphics Adapter (CGA); Enhanced Graphics Adapter (EGA); graphics; video adapters; Video Graphics Array Adapter (VGA)
 - current 386, 397
 - returning information on 398
 - linking 387
 - loading and selecting 386, 387
 - new
 - adding 386
 - registering 387
 - returning information on 397, 398
 - supported by Borland C++ 385
- graphics.h (header file) 384
- greater-than operator (>) 82, 91
- greater-than or equal-to operator (>=) 82, 91

H

- hash table
 - iterators 274
- HashTable class 272
- HashTableIterator class 274
- hashValue
 - Association member function 248
 - Basedate member function 252
 - Basetime member function 254
 - Btree member function 256
 - Container member function 261
 - HashTable member function 273
 - List member function 276
 - Object member function 280
 - PriorityQueue member function 283
 - Sortable member function 288
 - String member function 292
- hasMember
 - Bag member function 251
 - Btree member function 256
 - Collection member function 259

- PriorityQueue member function 283
- hdrfile
 - pragma 172
- hdrstop
 - pragma 173
- header files *See also* include files
 - Borland C++ versus Microsoft C 309
 - complex numbers 371
 - extern keyword and 33
 - function prototypes and 62
 - graphics 384
 - #include directive and 165
 - Microsoft C 308
 - name mangling and 33
 - precompiled 172, 173
 - prototypes and 60
 - variable parameters 62
- heap 30
 - objects *See* objects, heap
- Help compiler *See also* errors
- Hercules card *See* graphics drivers; video adapters
- hex (manipulator) 186
- hexadecimal
 - constants *See* constants, hexadecimal
 - digit 12
- hidden objects 29
- hiding *See* scope, C++
- hierachy *See* classes, hierarchies
- highvideo, conbuf member function 194
- horizontal tab 15
- hour
 - Basetime member function 254
- huge
 - functions
 - saving registers and 175
 - memory model *See* memory models
 - pointers *See* pointers, huge
- __HUGE__ macro 176
- huge (keyword) 48, 344, 350
 - assembly language and 52
- hundredths
 - Basetime member function 254

I

- i_add
 - Btree member function 256

- I prefix
 - class names 228
- IDE *See* Integrated Development Environment overlays and 361
- identifiers 10
 - Borland C++ keywords as 3
 - case 50
 - sensitivity and 10
 - classes 111
 - data types and 26, 27
 - declarations and 26
 - declaring 44
 - defined operator and 167
 - defining 161
 - duplicate 29
 - duration 29
 - enumeration constants 17
 - external *See also* extern (keyword)
 - case sensitivity and 413
 - name mangling and 32
 - global 175
 - #define and #undef directives and 162
 - length 413
 - linkage 31
 - attributes 31
 - mixed languages 50
 - name spaces *See* name spaces
 - no linkage attributes 32
 - Pascal 50
 - pascal (keyword)
 - case sensitivity and 11
 - rules for creating 10
 - scope *See* scope
 - significant characters in 413
 - storage class and 27
 - testing for definition 168
 - undefining 161
 - unique 31
- IEEE
 - floating-point formats 41, 414
 - rounding 374, 415
- #if directive 167
- if statements 99
 - nested 99
- #ifdef directive 168
- #ifndef directive 168
- ifstream (class) 198

- constructor 189
- insertion operations 189
- ignore, istream member function 203
- Imp suffix
 - class names 228
- implementation-specific ANSI items 411-422
- import libraries
 - module definition files and 333
- in_avail, streambuf member function 208
- INCLUDE directory
 - container class library 238
- INCLUDE environment variable 300
 - Resource Compiler and 301
 - windows.h and 301
- include files *See also* header files
 - #include directive and 165
 - paths 300
 - search algorithm for 165
 - searching for 418
- #include directive 165
 - search algorithm 165
- inclusive OR operator (|) 81, 93
 - truth table 93
- incomplete declarations
 - classes 112
 - structures 70
- increment operator (++) 81, 84
- incrNofKeys
 - Btree member function 256
- indeterminate arrays
 - structures and 60
- indirect member selector *See* operators, selection
- indirection operator (*) 81, 86
 - pointers and 58
- inequality operator (!=) 82, 92
- inheritance *See also* classes
 - constructors and destructors 124
 - multiple
 - base classes and 122
 - overloaded assignment operator and 139
 - overloaded operators and 136
- init, ios member function 200
- initialization 42, *See also* constructors; destructors
 - arrays 43
 - classes 129
 - objects 129
 - copy constructor and 129
 - operator 23
 - pointers 56
 - static member definitions and 116
 - structures 43
 - unions 43, 73
 - variables 44
- initialization modules 306
- initializers
 - automatic objects 45
 - C++ 45
 - new operator and 110
- initliterator
 - AbstractArray member function 244
 - Bag member function 251
 - Btree member function 256
 - Container member function 261
 - Deque member function 266
 - DoubleList member function 269
 - HashTable member function 274
 - List member function 276
 - PriorityQueue member function 283
 - Stacks member function 291
- inline
 - assembly language code *See* assembly language, inline
 - expansion 114
 - functions *See* functions, inline
 - keyword 114
 - pragma 173, 400
- InnerNode
 - Btree friend class 255
- input
 - C++
 - user-defined types 188
- inserter types 184
- inserters *See* output, C++
- insertion operator *See* overloaded operators, << (put to)
- inline, conbuf member function 194
- instance classes 216
- instances *See* classes, objects
- INT instruction 406
- integers 40, *See also* data types; floating point; numbers
 - arrays and 415

- C++ streams and 184
- casting to pointer 415
- constants *See* constants
- conversions *See* conversions
- division
 - sign of remainder 416
- enumerations and 417
- expressions
 - precedence 79
- long 40
 - range 19
- memory use 40
- pointers and 416
- range 19
- right shifted 416
- short 40
- signed
 - bitwise operators and 415
- sizes 40
- suffix 12
- unsigned
 - range 19
- values 413
- integral data types *See* characters; integers
- integrated development environment
 - DLLs and 332
 - module definition files and 329
 - nested comments command 8
 - Windows and 317
 - linking 329
- integrated environment
 - INCLUDE environment variable and 300
 - LIB environment variable and 300
 - Programmer's Workbench and 299
- intensity
 - setting 379
- interface dependencies *See* dependencies, interface
- internal linkage *See* linkage
- internal representations of data types 40
- interrupt (keyword) 48, 49, 407
- interrupts
 - beep
 - example 408
 - flag 342
 - functions
 - example of 408
 - floating-point arithmetic in 407
 - memory models and 49
 - void 49
- handlers 48
 - calling 408
 - installing 408
 - modules and 362
 - programming 407
 - registers and 49
- intrinsic pragma 173
- I/O
 - C++
 - formatting 186
 - precision 186
 - iomanip.h (header file)
 - manipulators in 185
 - ios (class) 182, 199
 - flags
 - format state 184
 - setting 186
 - ios data members 199
 - iostram.h (header file)
 - manipulators in 186
 - iostream (class) 202
 - iostream library 182
 - iostream_withassign (class) 202
 - IP (instruction pointer) register 340
 - is_open, filebuf member function 196
 - isA
 - Array member function 247
 - Association member function 248
 - Bag member function 251
 - Basedate member function 252
 - Basetime member function 254
 - Btree member function 256
 - Container member function 261
 - Date member function 264
 - Deque member function 267
 - Dictionary member function 268
 - DoubleList member function 269
 - Error member function 271
 - HashTable member function 274
 - List member function 276
 - Object member function 280
 - PriorityQueue member function 283
 - Queue member function 285
 - Set member function 286

- Sortable member function *288*
- Stack member function *291*
- String member function *292*
- Time member function *294*
- isalnum (function) *419*
- isalpha (function) *419*
- isAssociation
 - Association member function *248*
 - Object member function *280*
- isctrl (function) *419*
- isEmpty
 - Bag member function *251*
 - Container member function *262*
 - Deque member function *267*
 - PriorityQueue member function *283*
 - Stack member function *291*
- isEqual
 - AbstractArray member function *244*
 - Association member function *248*
 - Basedate member function *252*
 - Basetime member function *254*
 - Btree member function *256*
 - Container member function *262*
 - Error member function *271*
 - Object member function *280*
 - Sortable member function *288*
 - String member function *292*
- isLessThan
 - Basedate member function *252*
 - Basetime member function *254*
 - Sortable member function *288*
 - String member function *292*
- islower (function) *419*
- isprint (function) *419*
- isSortable
 - Object member function *280*
 - Sortable member function *288*
- istream (class) *202*
 - derived classes of *189*
- istream_withassign (class) *204*
- istream (class) *204*
- isupper (function) *419*
- Item
 - Btree friend class *255*
- itemsInContainer
 - Container data member *260*
- iteration statements *See loops*

- iterators
 - DoubleList *270*
 - internal and external *222*
- iterFuncType definition *261*

J

- Jg family of template switches *152*
- jump instructions, inline assembly language
 - table *403*
 - using *406*
- jump statements *See break statements; continue statements; goto statements; return statements*

K

- key
 - Association class *248*
 - Association member function *248*
- keywords *9*, *See also individual keyword names*
 - ANSI
 - predefined macro *178*
 - Borland C++
 - using as identifiers *3*
 - C++ *9*
 - combining *40*
 - macros and *162*
 - Microsoft C *310*

L

- labeled statements *98*
- labels
 - creating *23*
 - default *100*
 - function scope and *28*
 - goto statement and *103*
 - in inline assembly code *406*
- language extensions
 - conforming *3*
 - __LARGE__ macro *176*
- large code
 - data
 - and memory models *See memory models*
- Last-In-First-Out (LIFO) *225*
- lastElementIndex
 - AbstractArray data member *242*

- lastThat
 - Bag member function 251
 - Container member function 262
 - Object member function 281
- LeafNode
 - Btree friend class 255
- less-than operator (<) 82, 90
- less-than or equal-to operator (<=) 82, 91
- lexical grammar *See* elements
- lib directory
 - container class library 239
- LIB environment variable 300
- LibMain (function) 334
 - return values 334
- libraries
 - C
 - linking to C++ code 32
 - container class *See* container class library
 - DLLs and 330, 331
 - floating point
 - using 367
 - graphics 384
 - iostream 182
 - paths 300
 - prototypes and 65
 - stream class 181
 - __LINE__ macro 177
 - #define and #undef directives and 162
 - #line directive 169
 - lines
 - continuing 6, 19, 164
 - numbers 169
 - macro 177
 - LINK (Microsoft)
 - TLINK versus 307
 - linkage 31
 - C and C++ programs 32
 - external 31
 - C++ constants and 47
 - name mangling and 32
 - internal 31
 - no 31, 32
 - rules 31
 - static member functions 115
 - storage class specifiers and 31
 - Linker
 - dialog box
 - Windows and 329
 - linker
 - mixed modules and 355
 - using directly 355
 - List class 275
 - iterators 276
 - ListElement class 275
 - ListIterator class 276
 - lists
 - classes for 268
 - linked
 - traversing 276
 - literal strings *See* strings, literal
 - _loads (keyword) 48
 - huge functions and 52
 - uses for 52
 - local duration 30
 - logical AND operator (&&) 81, 93
 - logical negation operator (!) 81, 86
 - logical OR operator (||) 81, 94
 - long integers *See* integers, long
 - lookup
 - Dictionary member function 268
 - LOOKUP (container class library example program) 240
 - loops 101
 - break statement and 103
 - continue statement and 103
 - do while 101
 - for 102
 - C++ 102
 - while 101
 - string scanning and 101
 - lowerbound, AbstractArray data member 242
 - lowerBound, AbstractArray member function 244
 - lowvideo, conbuf member function 194
 - lvalues 26, *See also* rvalues
 - examples 53
 - modifiable 26

M

 - macros *See also* directives
 - argument lists 162
 - calling 162

- commas and
 - nested 163
- defining 159
 - conflicts 160
 - global identifiers and 162
- expansion 159
- far pointer creation 355
- keywords and 162
- MK_FP 355
- NULL
 - expansion 418
- parameters and 162
 - none 159
- parentheses and
 - nested 163
- precedence in
 - controlling 21
- predefined 175, *See also* individual macro names
 - ANSI keywords 178
 - C and C++ compilation 175, 176, 178, 179
 - C calling conventions 176
 - conditional compilation 176
 - current file 177
 - current line number 177
 - date 177
 - DLLs 177
 - DOS 177
 - memory models 176
 - overlays 178
 - Pascal calling conventions 178
 - templates 178
 - time 178
 - Windows applications 179
- redefining 160
- side effects and 164
- undefining 160
 - global identifiers and 162
- main (function) 60
 - pascal keyword and 50
 - semantics of arguments to 412
- MAKE (program manager)
 - makefiles
 - Windows applications and 321
 - Microsoft C and 301
 - Windows applications and 321
- makefile 239
- malloc (function)
 - dynamic duration objects and 30
 - new operator and 108
 - zero-size memory allocation and 421
- malloc.h (header file)
 - alloc.h and 309
- mangled names 32
 - DLLs and 337
- manifest constants 175
- manipulators 185, *See also* C++, formatting, width; individual manipulator names
 - parameterized 185
 - syntax 186
- math
 - BCD *See* BCD
 - coprocessors *See* numeric coprocessors
 - errors
 - masking 370
 - functions
 - domain errors and 419
 - underflow range errors and 419
- matherr (function)
 - proper use of 370
- __MEDIUM__ macro 176
- medium memory model *See* memory models
- mem.h (header file)
 - memory.h and 309
- member functions 113, *See also* data members
 - access 118
 - constructors *See* constructors
 - defined 113
 - destructors *See* destructors
 - friend 113
 - inline *See* functions, inline, C++
 - nonstatic 113
 - private 118
 - protected 118
 - public 118
 - scope 116-119
 - static 115
 - linkage 115
 - this keyword and 115
 - structures and 67
 - this keyword and 113, 115
 - unions and 72
 - virtual
 - pure 216

- members, classes *See* data members; member functions
- members, structures *See* structures, members
- MemBlocks class 277
- memory *See also* memory addresses
 - allocation 30
 - assembly language code and huge functions and 52
 - graphics system 387
 - new and delete operators and 108
 - structures 69
 - Borland C++'s usage of 339
 - data types 87
 - extended and expanded *See* extended and expanded memory
 - heap 30
 - memory models and 347
 - overlays and 358
 - paragraphs 343
 - boundary 343
 - segments in 342
 - word alignment and structures 69
- memory addresses *See also* memory calculating 341, 343-344
 - constructors and destructors 124
 - far pointers and 344
 - near pointers and 344
 - pointing to 355
 - segment:offset notation 343
 - standard notation for 343
- memory.h (header file)
 - mem.h and 309
- memory models 349, 339-357
 - changing 354
 - compact 346
 - default function type 52
 - comparison 349
 - container class library and 239
 - default
 - overriding 52
 - defined 346
 - DLLs 331
 - function pointers and 55
 - functions
 - default type
 - overriding 48
 - graphics library 384
 - huge 347
 - default function type 52
 - illustrations 347-349
 - interrupt functions and 49
 - large 347
 - default function type 52
 - macros and 176
 - medium 346
 - default function type 52
 - memory apportionment and 347
 - Microsoft C and 309
 - mixing 355
 - function prototypes and 355
 - overlays and 359, 361
 - pointers
 - modifiers and 51
 - pointers and 344, 351
 - predefined macros and 176
 - small 346
 - default function type 52
 - smart callbacks and 323
 - tiny 346
 - default function type 52
 - Windows and 346
 - Windows applications and 326
- memory-resident routines 407
- MemStack class 278
- methods *See* member functions
- Microsoft C
 - Borland C++ projects and 299
 - _cdecl keyword 310
 - CL options
 - BCC options versus 302
 - COMDEFS and 309
 - converting from 299-311
 - environment variables and 300
 - floating-point return values 310
 - header files 308
 - Borland C++ header files versus 309
 - keywords 310
 - MAKE and 301
 - memory models and 309
 - structures 310
 - TLINK and 306
- Microsoft Windows applications
 - preprocessor macro 179

- minute
 - Basetime member function 254
- mixed modules
 - linking 355
- MK_FP (run-time library macro) 355
- modifiable lvalues *See* lvalues
- modifiable objects *See* objects
- modifiers 47
 - function type 52
 - pointers 51, 351
 - table 47
- Modula-2
 - variant record types 71
- module definition files 315
 - defined 333
 - IDE options and 329
 - import libraries and 333
 - LibMain function and 334
 - TLINK and 329
 - /Tw TLINK option and 329
- modules
 - linking mixed 355
 - size limit 349
- modulus operator (%) 81, 88
- Monochrome Display Adapter *See* graphics
 - drivers; video adapters
- Month
 - Basedate member function 252
- __MSC macro 308
- __MSDOS__ macro 177
- multibyte characters 413
- multidimensional arrays *See* arrays
- multiple inheritance *See* inheritance
- multiplication operator (*) 81, 88

N

- \n (newline character) 15
- name mangling 32
- name spaces
 - scope and 28
 - structures 69
 - C++ 69
- nameOf
 - Arrays member function 247
 - Association member function 248
 - Bag member function 251
 - Basedate member function 253
 - Basetime member function 254
 - Btree member function 256
 - Container member function 262
 - Date member function 264
 - Deque member function 267
 - Dictionary member function 268
 - DoubleList member function 269
 - Error member function 272
 - HashTable member function 274
 - List member function 276
 - Object member function 281
 - PriorityQueue member function 283
 - Set member function 286
 - Sortable member function 288
 - Stacks member function 291
 - String member function 292
 - Time member function 294
- names *See* identifiers
 - mangled
 - DLLs and 338
 - qualified 117
- near (keyword) 48, 344, 350
- near functions *See* functions, near
- near pointers *See* pointers, near
- negation
 - logical (!) 81, 86
- negative offsets 341
- nested
 - classes 117
 - comments 7, 8
 - conditional directives 167
 - declarators 417
 - types 117
- new
 - Object member function 281
- new (operator) 108
 - arrays and 109
 - constructors and destructors and 125
 - destructors and 133
 - dynamic duration objects and 30
 - handling return errors 109
 - initializers and 110
 - overloading 110, 137
 - prototypes and 109
 - _new_handler (for new operator) 109
- newline characters
 - creating in output 15

- inserting 186
- NMAKE (Microsoft's MAKE utility) 301
- no linkage *See* linkage
- Node
 - Btree friend 257
 - Btree friend class 255
- non-container classes 218
- nondefining declarations *See* declarations, referencing
- nonzero digit 12
- normalized pointers *See* pointers, normalized
- normvideo, conbuf member function 194
- not equal to operator (!=) 82, 92
- not operator (!) 81, 86
- NULL
 - macro 418
 - pointers and 56
 - using 56
- null
 - characters
 - binary stream and 420
 - directive (#) 159
 - inserting in string 186
 - pointer constant 418
 - pointers 56
 - statement 22, 99
 - strings 18
- number of arguments 23
- numbers *See also* constants; data types; floating point; integers
 - base
 - setting for conversion 186
 - BCD *See* BCD
 - converting *See* conversions
 - decimal
 - conversions 186
 - hexadecimal 12
 - backslash and 14
 - conversions 186
 - displaying 15
 - lines *See* lines, numbers
 - octal 12
 - backslash and 14
 - conversions 186
 - displaying 15
 - escape sequence 15

- numeric coprocessors *See also* 80x86 processors
 - autodetecting 369
 - built in 367
 - floating-point emulation 368
 - floating-point format 414
 - registers and 370

O

- O prefix 229
 - class names 228
- .OBJ files
 - converting .BGI files to 387
 - DLLs and 330
 - Windows and 330
- Object class 216, 279
- Object container class library
 - version 3.0 changes to 214
- objectAt, AbstractArray member function 244
- ObjectBrowser
 - container class library and 216
- objects 25, *See also* C++
 - aliases 105
 - automatic 30, 222
 - initializers 45
 - class names and 112
 - detaching 221
 - duration 29
 - far
 - class names 352
 - combining into one segment 352
 - declaring 352
 - option pragma and 352
 - heap 222
 - hidden 29
 - in containers
 - counting 260
 - displaying 260
 - iterating 260
 - ownership 260
 - initializers 45
 - list of declarable 34
 - modifiable 49
 - ownership 220
 - pointers 55
 - function pointers vs. 54
 - sortable 286

- static
 - initializers 45
 - temporary 107
 - volatile 49
 - accessing 417
- oct (manipulator) 186
- octal constants *See* constants, octal
- octal digit 12
- offsets 344
 - component of a pointer 355
- ofstream (class) 205
 - base class 189
 - constructor 189
 - insertion operations 189
- opcodes 402, *See also* assembly language
 - defined 400
 - mnemonics
 - command-line compiler option (-i) 403
 - table 402
 - repeat prefixes 403
- open
 - filebuf member function 196
 - fstream member function 197
 - fstreambase member function 198
 - ifstream member function 199
 - ofstream member function 205
- open mode *See* files, opening, C++
- operands (assembly language) 400
- operating mode of screen *See* screens, modes
- operator <
 - overloaded 288
- operator =
 - String member function 293
- operator >
 - overloaded 289
- operator !=
 - overloaded 282
- operator ++
 - ArrayIterator member function 247
 - BtreeIterator member function 257
 - ContainerIterator member function 264
 - DoubleListIterator member function 270
 - HashTableIterator member function 274
 - ListIterator member function 277
- operator <<
 - Object friends 281
- operator <=
 - overloaded 289
- operator ==
 - overloaded 282
- operator >=
 - overloaded 289
- operator []
 - AbstractArray member function 244
 - Btree member function 256
- operator --
 - DoubleListIterator member function 270
- operator (keyword)
 - overloading and 135
- operator char *
 - String member function 293
- operator functions *See* overloaded operators
- operator int
 - BtreeIterator member function 257
 - ContainerIterator member function 263
 - DoubleListIterator member function 271
 - HashTableIterator member function 274
 - ListIterator member function 277
- operator int, ArrayIterator member function 247
- operators 79, 79-82
 - 1's complement (~) 81, 86
 - addition (+) 81, 88
 - address (&) 81, 85
 - AND (&) 81, 92
 - truth table 93
 - AND (&&) 81, 93
 - assignment (=) 81, 95
 - compound 96
 - overloading 139
 - binary 81
 - overloading 139
 - bitwise
 - AND (&) 81, 92
 - truth table 93
 - complement (~) 81, 86
 - inclusive OR (|) 81, 93
 - truth table 93
 - signed integers and 415
 - truth table 93
 - XOR (^) 81, 93
 - truth table 93
 - C++ 80

- delete *108*, *See* delete (operator)
- dereference pointers *82, 97*
- new *See* new (operator)
- pointer to member *See* operators, C++,
dereference pointers
- scope (::) *82, 108*
- conditional (? :) *82, 94*
- context and meaning *80*
- decrement (–) *81, 84*
- defined operator *167*
- division (/) *81, 88*
 - rounding *88*
- equality *82, 91*
- evaluation (comma) *82, 96*
- exclusive OR (^) *81, 93*
 - truth table *93*
- function call () *83*
- inclusive OR (|) *81, 93*
 - truth table *93*
- increment (++) *81, 84*
- indirection (*) *81, 86*
 - pointers and *58*
- inequality (!=) *82, 92*
- list *80*
- logical
 - AND (&&) *81, 93*
 - negation (!) *81, 86*
 - OR (||) *81, 94*
- manipulators *See* manipulators
- modulus (%) *81, 88*
- multiplication (*) *81, 88*
- OR (^) *81, 93*
 - truth table *93*
- OR (|) *81, 93*
 - truth table *93*
- OR (||) *81, 94*
- overloading *See* overloaded operators
- postfix *82*
- prefix *82*
- relational *82, 90*
- remainder (%) *81, 88*
- selection (. and ->) *82, 83*
 - overloading *140*
 - structure member access and *67, 83*
- shift bits (<< and >>) *81, 89*
- sizeof *87*
 - data type *418*
- subtraction (–) *81, 89*
- unary
 - overloading *138*
 - unary minus (–) *81, 86*
 - unary plus (+) *81, 86*
- option pragma *173*
 - far objects and *352*
- OR operator
 - bitwise inclusive (|) *81, 93*
 - truth table *93*
 - logical (||) *81, 94*
- order
 - Btree member function *256*
- ordered collections *218, 224*
- ostream (class) *205*
 - derived classes of *189*
 - flushing *186*
- ostream_withassign (class) *206*
- ostrstream (class) *206*
- out of memory error *339*
- out_waiting, streambuf member function *208*
- output
 - C++
 - user-defined types *188*
 - directing *383*
 - functions *377*
- overflow
 - conbuf member function *194*
 - filebuf member function *196*
 - strstreambuf member function *211*
- overflows
 - expressions and *79*
 - flag *342*
- __OVERLAY__ macro *178*
- overlays *357-365*
 - assembly language routines and *363*
 - BP register and *363*
 - buffers
 - default size *362*
 - cautions *362*
 - command-line options (–Yo) *360*
 - debugging *362*
 - designing programs for *361*
 - extended and expanded memory and *364*
 - how they work *357*
 - large programs *357*
 - memory map *358*

- memory models and 359, 361
- predefined macro 178
- routines, calling via function pointers 362
- overloaded constructors *See* constructors, overloaded
- overloaded functions
 - defined 113
 - templates and 147
- overloaded operators 78, 80, 135-140
 - >> (get from) 187
 - complex numbers and 371
 - << (put to) 183
 - complex numbers and 371
 - assignment (=) 139
 - binary 139
 - brackets 140
 - complex numbers and 371
 - creating 114
 - defined 113
 - delete 137
 - functions and 78
 - inheritance and 136
 - new 110, 137
 - operator functions and 135, 136
 - operator keyword and 135
 - parentheses 140
 - precedence and 78
 - selection (->) 140
 - unary 138
- _ovrbuffer (global variable) 358, 362
- ownsElements 220
 - Bag member function 252
 - TShouldDelete member function 296

P

- p command-line option (Pascal calling convention) 50, 176, 178
- cdecl and 51
- padding (C++) 187
- pages
 - active
 - defined 390
 - setting 389
 - buffers 390
 - visual
 - defined 390
 - setting 389

- painting *See* graphics, fill, operations
- palettes *See* graphics, palettes
- paragraphs *See* memory, paragraphs
- parameterized
 - manipulators *See* manipulators
 - types *See* templates
- parameters *See also* arguments
 - arguments vs. 3
 - default
 - constructors 126
 - ellipsis and 23
 - empty lists 39
 - fixed 62
 - formal 64
 - C++ 64
 - scope 64
 - function calls and 64
 - passing
 - C 48, 50
 - Pascal 48, 50
 - variable 62
- parentheses 21
 - as function call operators 83
 - macros and 21
 - nested
 - macros and 163
 - overloading 140
- parity flag 342
- parsing 6
- Pascal
 - calling conventions
 - compiler option (-p) 50
 - functions 50
 - identifiers 50
 - case sensitivity and 11
 - parameter-passing sequence 48
 - variant record types 71
 - __PASCAL__ macro 178
- pascal (keyword) 48, 50
 - function modifiers and 52
 - preserving case while using 50
- _pascal (keyword)
 - fortran keyword and 310
- pass-by-address, pass-by-value, and pass-by-var
 - See* parameters; referencing and dereferencing
- pbase, streambuf member function 208

- pbump, streambuf member function 208
- pcount, ostream member function 207
- peek, istream member function 203
- peekAtHead
 - DoubleList member function 270
- peekAtTail
 - DoubleList member function 270
- peekHead
 - List member function 276
- peekLeft
 - Deque member function 267
 - PriorityQueue member function 283
- peekRight
 - Deque member function 267
- period as an operator *See* operators, selection
(. and ->)
- perror (function)
 - messages generated by 421
- phrase structure grammar *See* elements
- pointer-to-member operators *See* operators,
C++, dereference pointers
- pointers 54, *See also* referencing and
dereferencing
 - advancing 58
 - arithmetic 58, 345
 - assignments 56
 - base class
 - destructors and 134
 - C++ 105
 - reference declarations 59
 - casting to integer 415
 - changing memory models and 354
 - to class members 82, 97
 - comparing 90, 92, 99, 345
 - while loops 101
 - const 47
 - constants and 57
 - conversions *See* conversions
 - declarations 56
 - declarator (*) 23, 58
 - default data 349
 - delete operator and 132
 - dereference 82, 97
 - DLLs and 331, 335
 - far 48
 - adding values to 345
 - comparing 344
 - declaring 353-354
 - function prototypes and 354
 - memory model size and 353
 - registers and 344
 - far memory model and 344
 - function 55
 - C++ 55
 - modifying 52
 - object pointers vs. 54
 - void 55
 - generic 39, 56
 - huge 48, 345
 - comparing
 - != operator 345
 - == operator 345
 - declaring 353-354
 - overhead of 346
 - huge memory model and 344
 - initializing 56
 - integer type for 416
 - keywords for 48
 - manipulating 344
 - memory models and 344, 351
 - to memory addresses 355
 - modifiers 51, 350
 - near 48, *See also* segments, pointers
 - declaring 353-354
 - function prototypes and 354
 - memory model size and 353
 - registers and 344
 - near memory model and 344
 - normalized 345
 - null 56
 - NULL macro and 418
 - NULL and 56
 - operator (->)
 - overloading 140
 - structure and union access 67, 82, 83
 - overlays and 362
 - pointers to 55
 - range 19
 - reassigning 56
 - referencing and dereferencing 85
 - segment 48, 350, 351
 - stack 341
 - structure members as 67
 - typecasting 59

- virtual table
 - 32-bit, -WD option and 336
 - void 56
- pop
 - Stacks member function 291
- portable code
 - bit fields and 71
- positive offsets 341
- postdecrement operator (--) 81, 84
- postfix operators 82
- postincrement operator (++) 81, 84
- pptr, streambuf member function 208
- pragma directives
 - templates and 153
- #pragma exit
 - destructors and 133
- #pragma directives 171
 - argsused 171
 - exit 171
 - hdrfile 172
 - hdrstop 173
 - inline 173, 400
 - intrinsic 173
 - option pragma 173
 - far objects and 352
 - saveregs 175
 - startup 171
 - warn 175
- precedence 78, *See also* associativity
 - controlling 21
 - expressions 76
 - floating point 79
 - integer 79
 - overloading and operators 78
- precision, ios member function 201
- precompiled headers
 - storage file 172
- PRECONDITION macro 240
- predecrement operator (--) 81, 84
- predefined macros *See* macros, predefined
- prefix opcodes, repeat 403
- prefix operators 82
- preincrement operator (++) 81, 84
- preprocessor directives *See* directives
- printContentsOn
 - AbstractArray member function 244
- printers
 - printing direction 413
- printHeader
 - Container member function 262
- printOn
 - Association member function 248
 - Basedate member function 253
 - Basetime member function 254
 - Btree member function 256
 - Container member function 262
 - Date member function 265
 - Error member function 272
 - Object member function 281
 - Sortable member function 288
 - String member function 292
 - Time member function 294
- printSeparator
 - Container member function 262
- printTrailer
 - Container member function 263
- priority queues 282
- PriorityQueue class 282
- private (keyword)
 - data members and member functions 118
 - derived classes and 120
 - unions and 73
- procedures *See* functions
- producer (streams) 181
- profilers 362
- Programmer's Platform *See* Integrated Development Environment
- Programmer's Workbench
 - integrated environment and 299
- programs
 - creating 5
 - performance
 - improving 45
 - size
 - reducing 45
 - terminate and stay resident
 - interrupt handlers and 407
 - very large
 - overlying 357
- Project Manager
 - Resource Compiler and 331
 - resources and 331

- projects
 - files
 - graphics library listed in *384*
 - Microsoft C and *299*
 - prolog and epilog code
 - generating *322*
 - promotions *See conversions*
 - protected (keyword)
 - data members and member functions *118*
 - derived classes and *120*
 - unions and *73*
 - prototypes *61-63*
 - arguments and
 - matching number of *65*
 - C++ *60*
 - ellipsis and *62, 65*
 - examples *61, 62*
 - far and near pointers and *354*
 - function calls and *64*
 - function definitions and
 - not matching *65*
 - header files and *62*
 - libraries and *65*
 - mixing modules and *355*
 - new operator and *109*
 - scope *See scope*
 - pseudovariables
 - register *10*
 - ptrAt, AbstractArray member function *244*
 - ptrToRef
 - Object member function *281*
 - public (keyword)
 - data members and member functions *118*
 - derived classes and *120*
 - unions and *73*
 - punctuators *21, 21-24*
 - pure (keyword)
 - virtual functions and *142*
 - pure specifier *37*
 - push
 - Stacks member function *291*
 - put
 - ostream member function *206*
 - PriorityQueue member function *284*
 - Queue member function *285*
 - put to operator (<<) *See overloaded operators, >> (put to)*

- putback, istream member function *203*
- putenv (function)
 - environment names and methods *422*
- putLeft
 - Deque member function *267*
- putRight
 - Deque member function *267*

Q

- qualified names *117*
- question mark
 - colon conditional operator *82, 94*
 - displaying *15*
- Queue class *284*
 - example program *240*
- queues *284*
 - double-ended *265*
- QUEUETEST (container class library example program) *240*
- quotes, displaying *15*

R

- \r (carriage return character) *15*
- RAM
 - Borland C++'s use of *339*
- ranges
 - floating-point constants *17*
- rank
 - Btree member function *256*
- rdbuf
 - constream member function *195*
 - fstream member function *197*
 - fstreambase member function *198*
 - ifstream member function *199*
 - ios member function *201*
 - ofstream member function *205*
 - strstreambase member function *210*
- rdstate, ios member function *201*
- read, istream member function *203*
- realloc (function)
 - zero-size memory allocation and *421*
- reallocate
 - MemBlocks member function *278*
- reallocate, AbstractArray member function *244*
- records *See structures*

- recursive functions
 - memory models and 352
- reference declarations 59
 - position of & 39, 106
- references
 - forward 26
- referencing and dereferencing 85, *See also*
 - pointers
 - asterisk and 23
 - C++ 105
 - functions 106
 - simple 106
 - pointers 82, 97
 - referencing data in inline assembly code 404
 - referencing declarations *See* declarations
 - register (keyword) 45
 - class members and 113
 - external declarations and 36
 - formal parameters and 64
 - local duration and 30
 - registers
 - 8086 340-342
 - AX 340
 - base point 341
 - BP 341
 - overlays and 363
 - BX 340
 - CS 342, 344
 - CX 341
 - DI 341
 - assembly language and 404
 - DS 342, 344
 - _loadds and 52
 - DX 341
 - ES 342
 - flags 340, 341
 - hardware
 - bit fields and 71
 - index 340, 341
 - interrupts and 49
 - IP (instruction pointer) 340
 - LOOP and string instruction 341
 - math operations 340, 341
 - numeric coprocessors and 370
 - objects and 416
 - pseudovariables 10
 - saving with huge functions 175
 - segment 341, 342
 - SI 341
 - assembly language and 404
 - SP 341
 - special-purpose 341
 - SS 342
 - values
 - preserving 52
 - variable declarations and 45
 - variables 45
 - in inline assembly code 404
 - relational operators *See* operators, relational
 - remainder operator (%) 81, 88
 - remove (function)
 - open files and 420
 - removeEntry, AbstractArray member function 244
 - rename (function)
 - preexisting file name and 420
 - repeat prefix opcodes 403
 - reset
 - Timer member function 295
 - resetiosflags (manipulator) 185, 186
 - resolution *See* screens, resolution
 - Timer member function 295
 - Resource Compiler
 - environment variables and 301
 - functionality 315
 - invoking 319
 - linking and 331
 - Project Manager and 331
 - Windows and 313
 - Windows applications and 315
 - resources
 - adding 319, 331
 - defined 315
 - Project Manager and 331
 - restart
 - ArrayIterator member function 247
 - BtreeIterator member function 257
 - ContainerIterator member function 264
 - DoubleListIterator member function 271
 - HashTableIterator member function 275
 - ListIterator member function 277
 - return
 - statements
 - functions and 104

- types 61
- REVERSE (container class library example program) 240
- rounding
 - banker's 374
 - direction
 - division 88
 - errors 372
 - rules 415
- routines, assembly language *See* assembly language
- rvalues 27, *See also* lvalues

S

- S prefix
 - class names 228
- saveregs pragma 175
- _saveregs (keyword) 48, 52
 - uses for 52
- sbumpc, streambuf member function 208
- scalar data types *See* data types
- scaling factor
 - graphics 389
- scanf (function)
 - >> operator and 187
- scope 27-29, *See also* visibility
 - block 28
 - block statements and 98
 - C++ 29, 143-145
 - hiding 144
 - operator (::) 82, 108
 - rules 144
 - classes 28
 - names 112
 - enclosing 143
 - enumerations 28
 - C++ 75
 - file 28
 - static storage class specifier and 31
 - formal parameters 64
 - function 28
 - prototype 28
 - global 28
 - goto and 28
 - identifiers and 11
 - local
 - duration and 30

- members 116-119
- name spaces and 28
- pointers 56
- storage class specifiers and 45-47
- structures 28
- unions 28
- variables 28
- visibility and 29
- screens *See also* graphics; text; windows
 - aspect ratio 389
 - attributes, controlling 379
 - cells
 - attributes 382
 - blinking 383
 - characters in 375
 - colors 382
 - clearing 389
 - colors 382, 392
 - coordinates 377
 - starting positions 376
 - cursor
 - changing 380
 - manipulating 378
 - modes
 - controlling 379
 - defining 375
 - graphics 376, 384, 386
 - selecting 386
 - text 375, 381, 386
 - resolution 376, *See also* graphics, pixels
 - viewports *See* graphics
- search.h (header file) 309
- searches
 - #include directive algorithm 165
- second
 - Basetime member function 254
- seekg, istream member function 204
- seekoff
 - filebuf member function 196
 - streambuf member function 208
 - strstreambuf member function 211
- seekp, ostream member function 206
- seekpos, streambuf member function 209
- _seg (keyword) 48, 350, 351
 - _segment keyword and 310
- segment:offset address notation 343
 - making far pointers from 355

- `_segment` (keyword) 310
- segmented memory architecture 342
- segments 343, 346
 - component of a pointer 355
 - memory 342
 - pointers 48, 350, 351
 - registers 341, 342
- `_segname` (keyword) 310
- selection
 - operators *See* operators, selection
 - statements *See* if statements; switch statements
- `_self` (keyword) 310
- semicolons 22, 99
- sequence
 - classes *See* classes, sequence
- Set class 285
- `setb`, `streambuf` member function 209
- `setbase` (manipulator) 185, 186
- `setbkcolor` (function)
 - CGA vs. EGA 395
- `setbuf`
 - `filebuf` member function 196
 - `fstreambase` member function 198
 - `streambuf` member function 209
 - `strstreambuf` member function 211
- `setcursortype`, `conbuf` member function 194
- `_setcursortype` (function) 380
- `setData`, `AbstractArray` member function 244
- `SetDay`
 - `Basedate` member function 253
- `setf` (function) 187
- `setf`, `ios` member function 201
- `setfill` (manipulator) 185, 186
- `setg`, `streambuf` member function 209
- `setHour`
 - `Basetime` member function 254
- `setHundredths`
 - `Basetime` member function 255
- `setiosflags` (manipulator) 185, 186
- `setMinute`
 - `Basetime` member function 255
- `SetMonth`
 - `Basedate` member function 253
- `set_new_handler` (for `new` operator) 109
- `setp`, `streambuf` member function 209
- `setprecision` (manipulator) 185, 186
- `setSecond`
 - `Basetime` member function 255
- `setstate`, `ios` member function 201
- `setw` (manipulator) 185, 186
- `SetYear`
 - `Basedate` member function 253
- `sgetc`, `streambuf` member function 209
- `sgetn`, `streambuf` member function 209
- shapes *See* graphics
- shift bits operators (<< and >>) 81, 89
- short integers *See* integers, short
- SI register 341
- side effects
 - macro calls and 164
- sign 12
 - extending 15
 - conversions and 42
 - flag 342
- `signal` (function) 419
 - signal set 419
 - signals 419
- signed (keyword) 40
- single quote character
 - displaying 15
- `sink` (streams) 181
- size overrides in inline assembly code 404
- `size_t` (data type) 87, 137, 138
- `sizeof` (operator) 87
 - arrays and 87
 - classes and 87
 - data type 418
 - example 27
 - function-type expressions and 87
 - functions and 87
 - preprocessor directives and 87
 - unions and 72
- `__SMALL__` macro 176
- small code
 - data
 - and memory models *See* memory models
- smart callbacks
 - DLLs and 324
 - memory models and 323
 - Windows applications and 323
- `snxctc`, `streambuf` member function 209
- software interrupt instruction 406
- `Sortable` class 218, 286

- ordered collections 224
- SortedArray class 289
 - example program 240
- sorts
 - ascending 289
- sounds
 - beep 408
- source (streams) 181
- source code 5
- source directory
 - container class library 239
- SP register 341
- special-purpose registers (8086) 341
- specifiers *See* type specifiers
- splicing lines 6, 19
- sputback, streambuf member function 209
- sputc, streambuf member function 209
- sputn, streambuf member function 209
- squeezeEntry
 - AbstractArray member function 245
- SS register 342
- _ss (keyword) 48, 350
- stack
 - pointers 341
 - segment 342
- Stack class 289
 - example program 240
- standard conversions *See* conversions
- start
 - Timer member function 295
- startup files
 - DLLs and 331
- startup pragma 171
- state, ios data member 199
- state queries 396-398
- statements 97-104, *See also* individual
 - statement names
 - assembly language 98
 - block 98
 - marking start and end 21
 - default 100
 - do while *See* loops, do while
 - expression 22, 99
 - for *See* loops, for
 - if *See* if statements
 - iteration *See* loops
 - jump *See* break statements; continue
 - statements; goto statements; return statements
 - labeled 98
 - null 99
 - syntax 98
 - while *See* loops, while
 - static
 - data
 - DLLs and 336
 - duration 29
 - functions 31
 - members *See* data members, static; member functions, static
 - objects *See* objects, static
 - variables *See* variables, static
 - static (keyword) 46
 - linkage and 31
 - status
 - Timer member function 295
 - _status87 (function)
 - floating point exceptions and 370
 - stdarg.h (header file)
 - user-defined functions and 62
 - __STDC__ macro 178
 - #define and #undef directives and 162
 - stdtempl.h 230
 - stop
 - Timer member function 295
 - storage class
 - identifiers and 27
 - specifiers 45
 - functions and 32
 - linkage and 31
 - register
 - objects and 416
 - static
 - file scope and 31
 - stossc, streambuf member function 209
 - str
 - ostrstream member function 207
 - strstream member function 212
 - strstreambuf member function 211
 - streambuf (class) 182, 207
 - derived classes of 182
 - streams
 - binary
 - null characters and 420

- C++
 - classes and 181
 - clearing 186
 - data types 184
 - defined 181
 - errors 189
 - file class 181
 - flushing 186
 - formatted I/O 182
 - manipulators and *See* manipulators
 - memory buffer class 181, 182
 - output 183
 - string class 181
 - tied 201
- text
 - newline character and 420
- strerror (function)
 - messages generated by 422
- String class 292
 - example program 240
- strings
 - classes for 292
 - clipping 391
 - concatenating 18
 - continuing across line boundaries 19
 - converting arguments to 164
 - empty 18
 - inserting terminal null into 186
 - instructions
 - registers 341
 - literal 6, 18
 - null 18
 - scanning
 - while loops and 101
 - streams
 - C++ 190
 - streams and 181
- STRNGMAX (container class library example program) 240
- stroked fonts *See* fonts
- strstrea.h (header file)
 - string streams and 190
- strstream (class) 211
- strstreambase (class) 210
- strstreambuf (class) 210
- struct (keyword) 66, *See also* structures
 - C++ and 67, 112
- structures 65-71
 - access
 - C++ 120
 - bit fields *See* bit fields
 - Borland C++ versus Microsoft C 310
 - C++ 111
 - C vs. 112
 - complex 371
 - declaring 66
 - functions and 67
 - incomplete declarations of 70
 - indeterminate arrays and 60
 - initializing 43
 - member functions and 67
 - members
 - access 67, 83, 119
 - as pointers 67
 - C++ 67
 - comparing 90
 - declaring 66
 - in inline assembly code 405
 - restrictions 406
 - names 69
 - padding and alignment 416
 - memory allocation 69
 - name spaces 28, 69
 - tags 66
 - typedefs and 66
 - typedefs and 66
 - unions vs. 71
 - untagged 66
 - typedefs and 66
 - within structures 67
 - word alignment
 - memory and 69
- subscripting operator *See* brackets
- subscripts for arrays 21, 82
 - overloading 140
- subtraction operator (-) 81, 89
- switch statements 100
 - case statement and
 - duplicate case constants 100
 - case values
 - number of allowed 417
 - default label and 100
- symbolic constants *See* constants, symbolic
- sync, filebuf member function 196

- sync_with_stdio, ios member function 201
- syntax
 - assembly language statements 98
 - classes 111
 - declarations 33, 34
 - declarator 54
 - directives 158
 - expressions 77
 - inline assembly language 400
 - manipulators 186
 - notation 3
 - statements 98
 - templates 145
- system (function) 422
- system control, graphics 385

T

- \t (horizontal tab character) 15
- tags
 - enumerations 74
 - name spaces 75
 - structure *See* structures, tags
- TASM *See* Turbo Assembler
- taxonomy
 - types 38
- TC prefix 229
 - class names 228
- TCDEF.SYM 172
- __TCPLUSPLUS__ macro 178
- /Td and /Tw TLINK options (target file) 329
- tellg, istream member function 204
- tellp, ostream member function 206
- template-based container library 224
- template function 148
- TEMPLATES
 - conditional compilation 239
- Templates
 - Arrays example 237
 - Deque example 237
- templates 145, *See also* syntax
 - angle brackets 150
 - approach to class library 215, 226
 - arguments 150
 - class 149
 - compiler switches 152
 - container classes and 225
 - eliminating pointers 152

- function 146
 - implicit and explicit 148
 - overriding 148
- instantiating 228
- macro 178
- type-safe
 - generic lists 151
 - using switches 153
- __TEMPLATES__ macro 178
- temporary objects 107
- tentative
 - definitions 33
- terminate and stay resident programs
 - interrupt handlers and 407
- text
 - blocks
 - moving in and out of memory 378
 - capturing to memory 378
 - colors 382
 - in graphics mode 390
 - information on current settings 398
 - justifying 391
 - manipulation
 - functions 377
 - onscreen 378
 - output and 377
 - mode types 381
 - output
 - header file 377
 - reading and writing 378
 - scrolling 378
 - streams
 - writing
 - truncation and 420
 - strings
 - clipping 391
 - size 391
 - writing to screen 378
- textattr, conbuf member function 194
- textbackground, conbuf member function 194
- textcolor, conbuf member function 195
- textmode
 - conbuf member function 195
 - constream member function 195
- textmode (function) 377
- this (keyword)
 - nonstatic member functions and 113

- static member functions and 115
- tie, ios member function 201
- tied streams 201
- time *See also* date
 - formats 422
 - local
 - how defined 422
 - macro 178
 - Timer member function 295
- __TIME__ macro 178
 - availability 418
 - #define and #undef directives and 162
- Time class 294
 - example program 240
- Timer class 295
- __TINY__ macro 176
- tiny memory model *See* memory models
- TLINK (linker)
 - LIB environment variable and 300
 - LINK (Microsoft) versus 307
 - Microsoft C and 306
 - module definition files and 329
 - options
 - .COM files (/Td and /Tw) 329
 - DLLs (/Twe) 329
 - executable files (/Td and /Tw) 329
 - target files 329
 - /Td and /Tw (target files) 329
 - Windows executable (/Td and /Tw) 329
 - target file options (/Td and /Tw) 329
 - using directly 355
 - Windows applications and 320
- tokens
 - continuing long lines of 164
 - kinds of 8
 - parsing 6
 - pasting 7, 163
 - replacement 159
 - replacing and merging 24
- top
 - Stacks member function 291
- translation units 31
- trap flag 342
- truth table
 - bitwise operators 93
- TShouldDelete class 296
- Turbo Assembler 399

- Turbo C++
 - keywords
 - using as identifiers 3
- Turbo Profiler 362
- __TURBOC__ macro 179
- type-safe
 - lists 152
- type-safe linkage *See* linkage, type-safe
- type specifiers
 - elaborated 112
 - pure 37
- type taxonomy 38
- typecasting
 - pointers 59
- typed constants *See* constants
- typedef (keyword) 46
 - name space 28
 - structure tags and 66
 - structures and 66
- typedefs
 - untagged structures and 66
- types *See* data types

U

- U BCC option (undefine) 161
- UINT_MAX (constant) 89
- ULONG_MAX (constant) 89
- unary operators 81
 - minus (-) 81, 86
 - plus (+) 81, 86
 - syntax 85
- unbuffered, streambuf member function 209
- #undef directive 160
 - global identifiers and 162
- underbars *See* underscores
- underflow
 - filebuf member function 196
 - strstreambuf member function 211
- underflow range errors
 - mathematics functions and 419
- underscores
 - generating 48
 - ignoring 48
- union (keyword)
 - C++ 112
- unions 71
 - accessing 415

- anonymous
 - member functions and 72
- base classes and 120
- bit fields and *See* bit fields
- C++ 73, 111
 - C vs. 112
- classes and 73
- constructors and destructors and 125
- declarations 73
- initialization 43, 73
- members
 - access 83, 119
- name space 28
- sizeof and 72
- structures vs. 71
- units, translation *See* translation units
- unordered collections 224
 - Bag class 250
 - Dictionary class 267
 - DoubleList class 268
 - HashTable class 272
 - List class 275
 - Set class 285
- unsetf (function) 187
- unsetf, ios member function 201
- unsigned (keyword) 40
- untagged structures *See* structures, untagged
- upperbound, AbstractArray data member 243
- upperBound, AbstractArray member function 245
- user-defined formatting flags 202
- UTIL.DOC 392

V

- \v (vertical tab character) 15
- value
 - Association class 248
 - Association member function 249
- value, passing by *See* parameters
- values
 - comparing 90
- var, passing by *See* parameters
- varargs.h (header file) 309
- variable number of arguments 23
- variables
 - automatic *See* auto (keyword)
 - declaring 44
 - external 45
 - global *See* global variables
 - initializing 44
 - internal linkage 46
 - name space 28
 - offsets in inline assembly code 404
 - pseudo *See* pseudovariables
 - register *See* registers, variables
 - volatile 49
- variant record types *See* unions
- vectors, interrupt *See* interrupts
- vertical tab 15
- video
 - adapters
 - graphics, compatible with Borland C++ 385
 - video, adapters
 - graphics, compatible with Borland C++ 385
 - video adapters *See also* Color/Graphics Adapter (CGA); Enhanced Graphics Adapter (EGA); graphics drivers; Video Graphics Array Adapter (VGA)
 - modes 375
 - output
 - directing 383
 - using 375-398
 - Video Graphics Array Adapter (VGA) *See also* graphics drivers; video adapters
 - color control 395
 - viewports *See* graphics
 - virtual
 - base classes *See* classes, base, virtual
 - destructors *See* destructors, virtual
 - functions *See* member functions, virtual
 - tables
 - 32-bit pointers and, -WD option and 336
 - DLLs and 336
 - storing in the code segment, -WD option and 336
 - virtual (keyword)
 - constructors and destructors and 124
 - functions and 140
- visibility 29, *See also* scope
 - C++ 29
 - pointers 56
 - scope and 29

- visual page
 - defined 390
 - setting 389
- void (keyword) 39
 - function pointers and 55
 - functions and 62
 - interrupt functions and 49
 - pointers 56
 - typecasting expressions and 39
- volatile (keyword) 47, 49
 - formal parameters and 64
- VROOMM 357, *See* overlays

W

- W BCC options (Windows applications) 322
- warn pragma 175
- warnings
 - disabling 171
 - overriding 175
 - pragma warn and 175
- wchar_t (wide character constants) 16, 414
 - arrays and 44
- WD BCC options (.DLLs with all exports) 324
- WDE BCC options (DLLs with explicit exports) 324
- WEP (function) 334
 - return values 335
- WHELLO (Windows program) 314
 - compiling and linking 315
- wherex, conbuf member function 195
- wherey, conbuf member function 195
- while loops *See* loops, while
- whitespace 6
 - comments and 8
 - comments as 6
 - extracting 186
- wide character constants (wchar_t) 16, 414
- width, ios member function 202
- window
 - conbuf member function 195
 - constream member function 195
- window (function)
 - default window and 376
 - example 381
- Windows
 - _export and 325
 - libraries 330

- modules
 - compiling and linking 313
 - object files 330
 - prolog and epilog code 322
- windows
 - active
 - erasing 378
 - controlling 379
 - creating 379
 - default type 376
 - defined 376
 - managing
 - header file 377
 - output in 379
 - scrolling 378
 - text
 - creating 381
 - default size 380
- Windows (Microsoft) *See* Microsoft Windows and Windows
- Windows All Functions Exportable command 322
- Windows applications 313-338
 - command-line compiler and 318
 - command-line compiler options 319, 322, 323, 324
 - _export and 324
 - export functions and 322, 323
 - IDE and 317
 - linking 329
 - memory models and 326
 - prolog and epilog code 322
 - Resource Compiler and 313, 315, 331
 - smart callbacks and 323
 - WHELLO 314
 - WinMain function and 318
- Windows DLL All Functions Exportable command 324
- Windows DLL Explicit Functions Exported command 324
- Windows Explicit Functions Exported command 323
- windows.h (header file)
 - INCLUDE environment variable and 301
- Windows Smart Callbacks command 323
- _Windows macro 179
- WinMain (function) 318

- return value *318*
- WN BCC options (.OBjs with explicit exports) *323*
- word alignment *69, 416, 417*
 - memory and structures *69*
- write, ostream member function *206*
- ws (manipulator) *186*
- WS BCC options (smart callbacks) *323*
- _wscroll (global variable) *378*
- wxxx options (warnings)
 - warn pragma and *175*

X

- x_fill, ios data member *199*
- x_flags, ios data member *199*
- x_precision, ios data member *199*
- x_tie, ios data member *200*
- x_width, ios data member *200*

- xalloc, ios member function *202*
- \xH (display a string of hexadecimal digits) *15*
- XOR operator (^) *81, 93*
 - truth table *93*

Y

- Y BCC option (overlays) *178*
- Y command-line compiler option (compiler generated code for overlays) *361*
- Year
 - Basedate member function *253*
- Yo option (overlays) *360*

Z

- ZERO
 - Object data member *279*
- zero flag *342*
- zero-length files *420*
- zX options (code and data segments) *352*

3.1

BORLAND® C++

B O R L A N D

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan and United Kingdom ■ Part #14MN-BCP03-31 ■ BOR 3858