

# 10

## Storage Mapping

---

When you interface C and C++ code with assembly language, or with code written in other programming languages, you need to know how data is mapped in the modules you are compiling. This chapter provides information about storage mapping. It contains the following sections:

§10.1: *Size and Alignment of Data Types*

§10.2: *Arrays*

§10.3: *Bit Fields*

§10.4: *Reversed Endianness*

§10.5: *Aligning Data*

§10.6: *How the Compiler Maps Variables to Storage*

§10.7: *Storage Mapping for C++ Class Objects*

---

### 10.1 Size and Alignment of Data Types

Some data types have a fixed size and byte alignment, while others might vary. Table 10.1 lists sizes and byte alignment of High C/C++ data types. (See Table 9.1 and Table 9.2 for minimum and maximum values that can be represented by High C/C++ data types.)

**Table 10.1** *Data Types: Size and Alignment*

Data Type	Size (Bytes)	Alignment (Bytes)
<code>char</code>	1	1
<code>short int</code>	2	2
<code>int</code>	4	4
<code>long int</code>	4	4

**Table 10.1 Data Types: Size and Alignment (Continued)**

Data Type	Size (Bytes)	Alignment (Bytes)
<code>long long int</code>	8	8
<code>float</code>	4	4
<code>double</code>	8	8
<code>long double</code>	16	8
Pointer	4	4
Full-function value	8	4
<code>T[n]</code>	<code>n*sizeof(T)</code>	Same as T

Table 10.1 lists the normal size and alignment of data types `long long int` and `long double`.

---

**Note:** Some targets might not implement `long long int` and `long double` types. For those targets, the size and alignment of `long long int` and `long double` are the same as the size and alignment of `long int` and `double`, respectively.

---

See toggles `Long_long_8` and `Long_double_16` in Chapter 4: *Using Compiler Toggles* for more information.

---

**Note:** A *full-function value* is a High C/C++ extension to support pointers to nested functions. It consists of a function address and a static link, and is akin to a Pascal routine. See the **High C/C++ Language Reference**.

---

---

### 10.1.1 Size of `enum` Types

The size of an **enum** type depends on the status of the `Long_enums` toggle (see Chapter 4: *Using Compiler Toggles*).

- If toggle `Long_enums` is `Off` (the default), the **enum** type maps to the smallest of one, two, or four bytes, such that all values can be represented.
- If toggle `Long_enums` is `On`, an **enum** maps to four bytes (matching the AT&T Portable C Compiler convention).

---

### 10.1.2 Size and Alignment of `struct` and `union` Types

The size of a **struct** or **union** depends on whether the compiler generates padding to align the construct's members. The compiler generates **struct** and **union** padding under the control of the `Align_members` pragma (described in Chapter 5: *Using Compiler Pragmas* and §10.5.1: *Aligning Structure Members: Pragma `Align_members`*).

The size of a **union** is the size of its largest member, padded so that its size is evenly divisible by its alignment. The size of a **struct** is the sum of the sizes of its members, including alignment padding between members. A **struct** is padded so that its size is evenly divisible by its alignment.

A **struct** or **union** is aligned according to the most stringent requirements of its members.

---

**Note:** Some implementations of High C/C++ do not support packed structures.

The reserved words `_Packed` and `_Unpacked` have been added to High C/C++ to allow control over member alignment on an individual **struct** or **union** basis.

A `_Packed struct` is not padded; an `_Unpacked struct` is padded.

---

## 10.2 Arrays

Arrays are stored in row-major order. This means that the last subscript in a multi-dimensional array varies fastest.

For example, this array:

```
int x[4][4]
```

is stored as this:

```
x[0][0], x[0][1], x[0][2], x[0][3], ..., x[3][3]
```

## 10.3 Bit Fields

The High C/C++ compiler supports signed and unsigned bit fields. A bit field cannot exceed 32 bits and is packed in consecutive bytes.

By default, bit fields are unsigned — that is, `int x:11;` specifies an unsigned bit field, and `long x:19;` specifies an unsigned long bit field. You can use the keywords **signed** and **unsigned** to specify signedness and unsignedness, respectively.

Table 10.2 shows the possible widths for bit fields, where  $w$  = maximum width.

**Table 10.2 Bit-Field Types: Width and Range of Values**

Bit-Field Type	Width (Bits)	Range of Values
<b>signed char</b>	1 to 8	$-2^{w-1}$ to $2^{w-1}-1$
<b>char</b>		0 to $2^w-1$
<b>unsigned char</b>		0 to $2^w-1$
<b>signed short</b>	1 to 16	$-2^{w-1}$ to $2^{w-1}-1$
<b>short</b>		0 to $2^w-1$
<b>unsigned short</b>		0 to $2^w-1$

**Table 10.2 Bit-Field Types: Width and Range of Values (Continued)**

Bit-Field Type	Width (Bits)	Range of Values
<b>signed int</b>	1 to 32	$-2^{w-1}$ to $2^{w-1}-1$
<b>int</b>		0 to $2^w-1$
<b>enum</b>		0 to $2^w-1$
<b>unsigned int</b>		0 to $2^w-1$
<b>signed long</b>	1 to 32	$-2^{w-1}$ to $2^{w-1}-1$
<b>long</b>		0 to $2^w-1$
<b>unsigned long</b>		0 to $2^w-1$
<b>signed long long</b>	1 to 64	$-2^{w-1}$ to $2^{w-1}-1$
<b>long long</b>		0 to $2^w-1$
<b>unsigned long long</b>		0 to $2^w-1$

*Storage boundaries for bit fields* A storage-unit boundary for a bit field is determined by the number of bytes needed to store a value for the associated type.

- **char** bit fields have boundaries on every byte.
- **short** bit fields have boundaries on even addresses.
- **long** bit fields have boundaries on addresses divisible by 4.
- **long long** bit fields are an exception, because they also have boundaries on addresses divisible by 4. (Not all targets implement **long long**. See note in §10.1.)

In *big-endian mode*, the compiler allocates bit fields from the most significant bit to the least significant bit.

In *little-endian mode*, the compiler allocates bit fields from the least significant bit to the most significant bit.

### 10.3.1 How the Compiler Maps Bit Fields

The compiler maps bit fields so that they never cross a storage-unit boundary. A bit field must fit within a four-byte word that is aligned to a four-byte boundary. If necessary, the bit field starts on the following unit boundary.

- Named bit fields* Named bit fields impose the same alignment on the enclosing **struct** or **union** as non-bit-field members.
- Unnamed bit fields of non-zero length* Unnamed bit fields of non-zero length do not affect the external alignment. In all other respects, they behave the same as named bit fields.
- Unnamed bit fields of zero length* An unnamed bit field of zero length causes alignment to occur at the next unit boundary, based on its type. Thus, a zero-length bit field of type **int** is aligned on the next four-byte boundary.

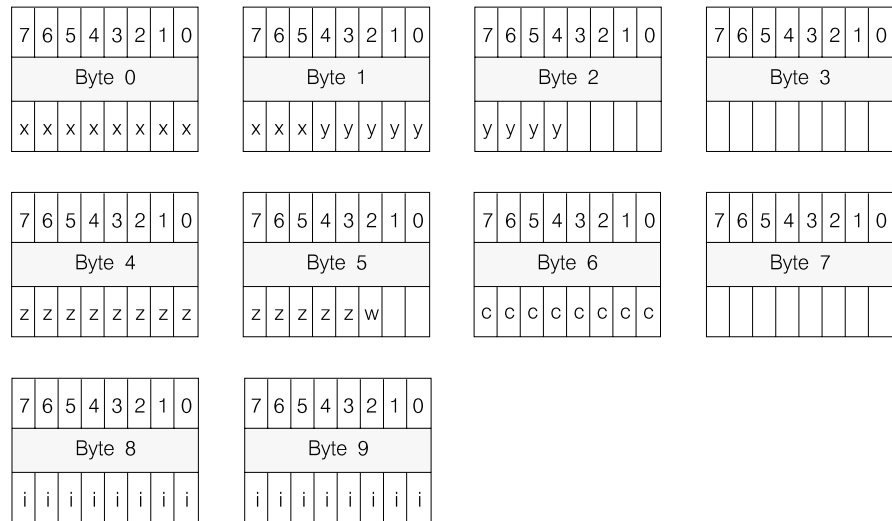
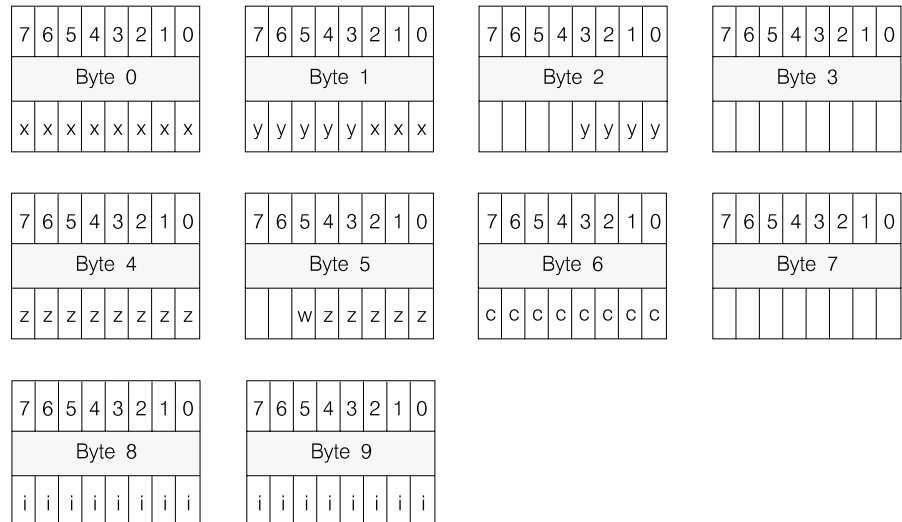
An unnamed bit field of zero length also imposes a four-byte alignment on an enclosing **struct**. For example, this **struct**:

```
struct {  
    unsigned x:11,y:9,:0,z:13,w:1;  
    char c;  
    short i;  
}
```

is mapped to memory as shown in Figure 10.1 (if compiled in big-endian mode), or as shown in Figure 10.2 (if compiled in little-endian mode).

This **struct** is aligned on address boundaries divisible by 4 because it contains **int** types. Note that the unnamed bit field *forces* padding, while alignment rules *sometimes* pad.

If **z** were changed to a **char** type, it would still be forced to begin in byte 4. If there were no unnamed bit field, **z** would begin in byte 2, 3, or 4, depending on whether it could fit in the space remaining without crossing its storage-unit boundary (which is 4).

**Figure 10.1 Mapping a struct (Big-Endian Mode)****Figure 10.2 Mapping a struct (Little-Endian Mode)**

---

## 10.4 Reversed Endianness

The type qualifier `_Reversed_Endian` enables you to declare variables that have an endianness opposite to that of the machine the program runs on. Loading and storing these variables requires byte-reversed memory accesses.

Here are some example declarations of reversed-endian variables:

- A reversed-endian variable:  
`_Reversed_Endian int a;`
- A normal pointer to a reversed-endian variable:  
`int * _Reversed_Endian a;`
- A reversed-endian pointer to a normal variable:  
`_Reversed_Endian int * a;`
- A reversed-endian pointer to a reversed-endian variable:  
`_Reversed_Endian int * _Reversed_Endian a;`

---

**Note:** At present, the High C/C++ compiler does not enforce type checking of reversed-endian variables. You must exercise caution to ensure that reversed-endian variables and pointers are not assigned to normal variables and pointers, and vice versa.

---

---

## 10.5 Aligning Data

The following sections describe pragmas you can use to align various types of data.

---

### 10.5.1 Aligning Structure Members: `Pragma Align_members`

Use `pragma Align_members` to specify how members of subsequently declared structures are to be aligned. The default setting for `Align_members` is 8.



This pragma has the form:

```
#pragma Align_members(n)
```

where *n* is a decimal integer constant denoting in bytes the *maximum* alignment boundary to be applied to a member. For example:

```
#pragma Align_members(2)
struct s {
    char c;           /* Offset 0, one-byte padding */
    int x;            /* Offset 2 */
};
```

Member *x*, which is type **int**, would ordinarily be aligned to a four-byte boundary. But because of the pending pragma, it is aligned to a two-byte boundary instead.

---

**Note:** The default alignment of data structures in a program is the natural alignment specified by the application binary interface.

---

Note that pragma **Align\_members** denotes the *maximum* boundary; an **int** member is never aligned to greater than a four-byte boundary regardless of how large a value is specified for *n*.

To pack a **struct** so that no padding is inserted between members, specify a setting of 1:

```
#pragma Align_members(1)
```

See the following sections for more information about alignment of structures:

- §10.5.2: *Aligning Data Structures: Pragma Pack*
- §10.5.3: *Aligning Structures and Saving Maximum Alignment: Pragma Push\_align\_members*
- §10.5.4: *Restoring Alignment: Pragma Pop\_align\_members*.

---

**WARNING:** If you use pragma **Align\_members** indiscriminately, you can severely impair run-time performance by trying to access **struct** members that are not properly aligned.

---

---

## 10.5.2 Aligning Data Structures: `Pragma Pack`

`Pragma Pack` is synonymous with `pragma Align_members` — it controls the default alignment of data structures.

- `Pragma Pack(1)` packs a **struct** so that no padding is inserted between members.
- `Pragma Pack()` “unpacks” the data; that is, it restores the alignment to the default (“natural”) alignment of data structures on the target machine, as specified by the application binary interface.

---

**Note:** Pragmas `Pack(n)` and `Pack()` are not stackable — `Pack()` does not revert to the previous `Pack(n)` — so these pragmas cannot be nested. For nesting, use `pragma Push_align_members` and `Pop_align_members` pairs.

---

See §10.5.3: *Aligning Structures and Saving Maximum Alignment: `Pragma Push_align_members`* for an example of when *not* to use `pragma Pack(1)/Pack()` pairs.

---

## 10.5.3 Aligning Structures and Saving Maximum Alignment: `Pragma Push_align_members`

`Pragma Push_align_members(n)` has the same effect as `pragma Align_members(n)` or `pragma Pack(n)`, except that `Push_align_members` saves the current setting of the maximum **struct** alignment.

### 10.5.3.1 Example: `Push_align_members / Pop_align_members` pairs

The following example illustrates why we recommend the use of `pragma Push_align_members(1)` and `Pop_align_members` pairs (because they

are “stackable”) rather than `pragma Pack(1)` and `Pack()` pairs (which are not stackable).

```
/* Pack structure */
#pragma Push_align_members(1)          /* push #1 */
/* --- Begin included header file --- */
/* ... */
/* Align to machine natural boundary */
#pragma Push_align_members(4)          /* push #2 */
struct first_st {
    char a, int b, char c;
}
/* Revert to previous alignment */
#pragma Pop_align_members()            /* pop #2 */
/* --- End included header file --- */
struct second_st {
    char a, int b, char c
}
/* Revert to previous alignment */
#pragma Pop_align_members()            /* pop #1 */
```

In this example, structure `first_st` is naturally aligned, while structure `second_st` is packed (aligned to 1).

### 10.5.3.2 Example: `Pack(n)/Pack()` pairs

If you replace the “push #1/pop #1” and “push #2/pop #2” pragma pairs in the preceding example with `Pack(n)/Pack()` pairs, as shown here, structures `first_st` and `second_st` will both be naturally aligned:

```
/* Pack structure */
#pragma Pack(1)                        /* push #1 */
/* --- Begin included header file --- */
/* ... */
```

```

#pragma Pack(4)                                /* push #2 */
struct first_st {
    char a, int b, char c;
}
/* Does not revert to previous alignment */
#pragma Pack()                                /* pop #2 */
/* --- End included header file --- */
struct second_st {
    char a, int b, char c;
}
/* Does not revert to previous alignment */
#pragma Pack()                                /* pop #1 */

```

This natural alignment occurs because the alignment setting is changed in the included header file, and `Pack()` does not revert to the previous alignment setting (but `Pack()` always restores the alignment setting to the natural alignment setting of the target machine).

---

### 10.5.4 Restoring Alignment: `Pragma Pop_align_members`

`Pragma Pop_align_members` cancels `Push_align_members` and restores the alignment that existed before the preceding `Push_align_members`.

If there is no pending `Push_align_members`, the alignment is restored to the default. (The default alignment of a `struct` is defined in §9.1.5: *Structures, Unions, and Bit Fields*.)

---

## 10.6 How the Compiler Maps Variables to Storage

The compiler defines up to six default control sections for placing code and data, in addition to any you might define with `pragma Code` or `pragma Data`. These are the default sections:

Name	Class	Contents
<code>.data1</code>	Data	Writable strings whose addresses are referenced from within the <code>.data</code> section

Name	Class	Contents
<code>.rdata</code>	Literal	Read-only string constants and read-only static variables (that is, those declared with the <b>const</b> qualifier)
<code>.text</code>	Text	Executable code

*Local static variables* Unless specified otherwise with a pragma `Data` specification, each local **static** variable is placed in one of the following sections:

- `.rdata`, if **const** qualified
- `.data`, if initialized
- `.bss`, if uninitialized

*Uninitialized global variables* Each uninitialized global variable with no **extern** specifier is defined as a common block.

*Initialized global variables* Each initialized global variable with no **extern** specifier is mapped into the `.data` section and given the *global* attribute.

*auto variables* Each **auto** variable is assigned either to a machine register or to storage in the routine's stack frame. The compiler attempts to place each **auto**-classed variable in a register, provided the variable's type is appropriate and its address is not required.

If `set jmp()` is called within a function, all **auto** variables are mapped to memory unless the **register** class is explicitly specified, so that the semantics of `set jmp()`/`long jmp()` work correctly.

For more about stack frames, see Chapter 11: *PowerPC Run-Time Organization*.

*extern variables* Each **extern** variable is given the *global* and *undefined* attributes.

*const-qualified variables* Each **const**-qualified static initialized variable, whether local or global, is mapped to the `.rdata` section.

For example:

```
main () {
    static const char hextable[] = "0123456789abcdef";
    int i;
    ...
    f(hextable[i]);
}
```

Any attempt to directly assign to `hextable` elicits an error message from the compiler. Any indirect attempt, such as through passing a pointer to `A` or to `hextable`, will most likely cause a protection fault or other error at execution time.

Note that string literals are placed in a `.rdata` section only if they are part of a `const` data object. For example:

```
const char * P = "a string";
```

Here `"a string"` is placed in the `.rdata` section, but `P` is placed in `data`. To place `P` in the `.rdata` section, do this:

```
const char * const P = "a string";
```

---

## 10.7 Storage Mapping for C++ Class Objects

This section describes how storage is arranged for class objects, treating such issues as virtual-function tables, virtual and non-virtual bases, and non-static members. You do not need this information to use High C/C++; it is provided for programmers who are interested in the layout of objects in storage.

---

### 10.7.1 Virtual-Function Tables

A class that contains virtual functions must have a virtual-function table, which is accessed whenever your program calls a virtual function. Each instance of a class with virtual functions contains a pointer to its virtual-function table.

For example, if `fnc()` is a virtual function in class `x`, `x.fnc()` essentially becomes `(x.vtab[index_of_fnc])()`, where `index_of_fnc` is the index of the function's address in the virtual-function table.

To avoid having multiple copies of the virtual-function tables for a class, the compiler produces a single copy of the table in the compilation unit that contains a function definition for the first non-pure-virtual, non-inline function member in a class.

In the following example, the virtual-function table is defined in the code generated for 2.cpp, and referenced from 1.cpp, in the call to `x.f`:

File `class.h`:

```
struct s { virtual f(); };
```

File 1.cpp:

```
#include "class.h"
s x; int i = x.f();
```

File 2.cpp:

```
#include "class.h"
int s::f() {
    /* Body of f. */
}
```

If a class's member functions are all pure **virtual** or **inline**, the compiler makes a copy of the class's virtual-function table for each compilation unit that makes use of the virtual-function table. The compiler issues a warning for each such class. You can turn the warning off with `pragma Offwarn()` (see Chapter 5: *Using Compiler Pragmas*).

We recommend that you *not* make a class's destructor **inline** if all other functions in the class are pure **virtual** or **inline**, because the virtual tables will then be generated in the module in which you define the destructor.

See also toggles `Import_vtabs` and `Export_vtabs` in Chapter 4: *Using Compiler Toggles*.

---

## 10.7.2 Direct and Indirect Bases

A *direct base* of a class is another class listed in the first class's declaration; for example, in:

```
class X: Y, Z { ... }
```

classes `Y` and `Z` are direct bases of `X`. An *indirect base* of a class `X` is a class that is not a base of `X` but is a base of one of `X`'s direct bases; for example, if we add:

```
class A: X { ... }
```

then class *x* is a direct base of *A*, and *Y* and *Z* are indirect bases of *A*.

---

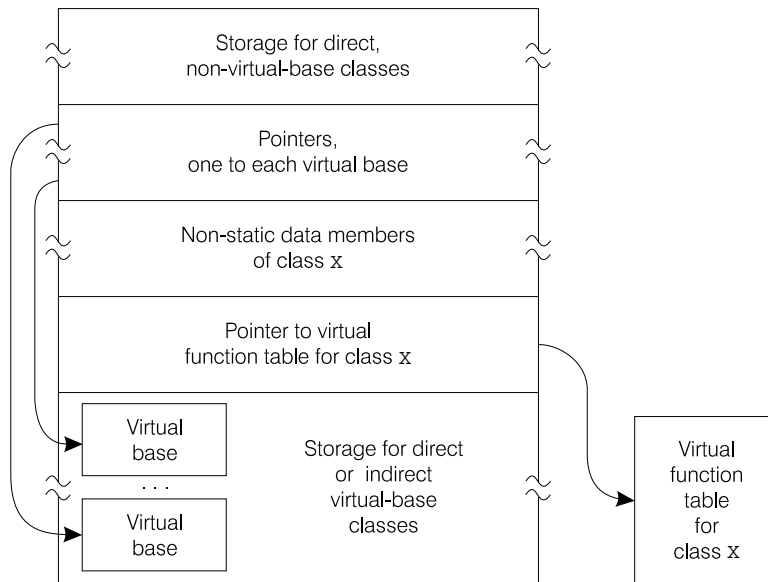
**Note:** For information about virtual and non-virtual bases, see the **High C/C++ Language Reference**.

---

Suppose a class *x* has the following base classes:

- direct non-virtual-base classes
- direct virtual-base classes
- indirect virtual-base classes

Storage for the components of an instance of an object of class *x* with two virtual base classes is organized as shown in Figure 10.3, except that redundant pointers might be optimized away.



**Figure 10.3** *Storage for an Instance of an Object of Class *x**

The non-static data members are allocated the same way as **struct** members in plain C; that is, consecutively as they appear in the source, subject to boundary alignment. Access specifiers do not affect the mapping.

The direct, non-virtual bases laid out at the beginning of class *x*'s storage do not include those bases' own virtual bases (if any); the storage for all virtual



bases is in class *x* and not in any of its bases. *x* has a pointer to each of its direct virtual bases. When *x* is constructed, these pointers are initialized; so, too, are any virtual-base pointers that reside in *x*'s direct or indirect bases (virtual or not).

So the storage for any base *B* allocated in an instance of *x* does *not* include any virtual bases of *B*. The storage for *B* in *x* may be thought of as a “truncated instance” of *B*.

If possible, rather than allocating a pointer to a virtual-function table within *x*, the compiler uses that same pointer in the first direct, non-virtual-base class *B* — provided *B* also has virtual functions. The compiler rearranges the storage of direct, non-virtual-base classes so that the first base stored has a virtual-function pointer, if any such base does. Consider this example:

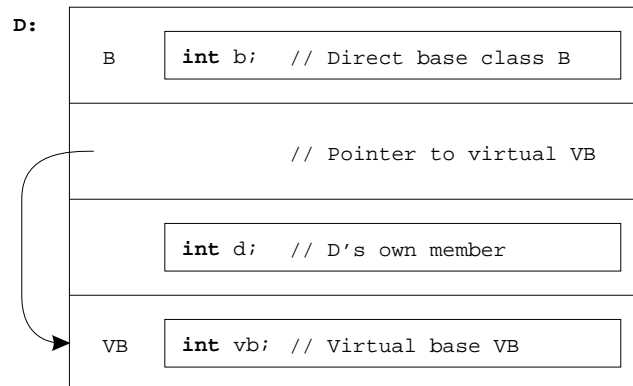
```
class A: B,X,D
```

In this case, if *x* has virtual functions and *B* does not, class *x* is stored first, followed by *B* and then *D*.

Here are two examples of class layout. The first one is reasonably simple:

```
class VB { int vb; };
class B { int b; };
class D: B, virtual VB { int d; };
```

Figure 10.4 shows the layout for an instance of class *D*.



**Figure 10.4** *Layout of an Instance of Class D*

The next example involves virtual bases that themselves have virtual bases:

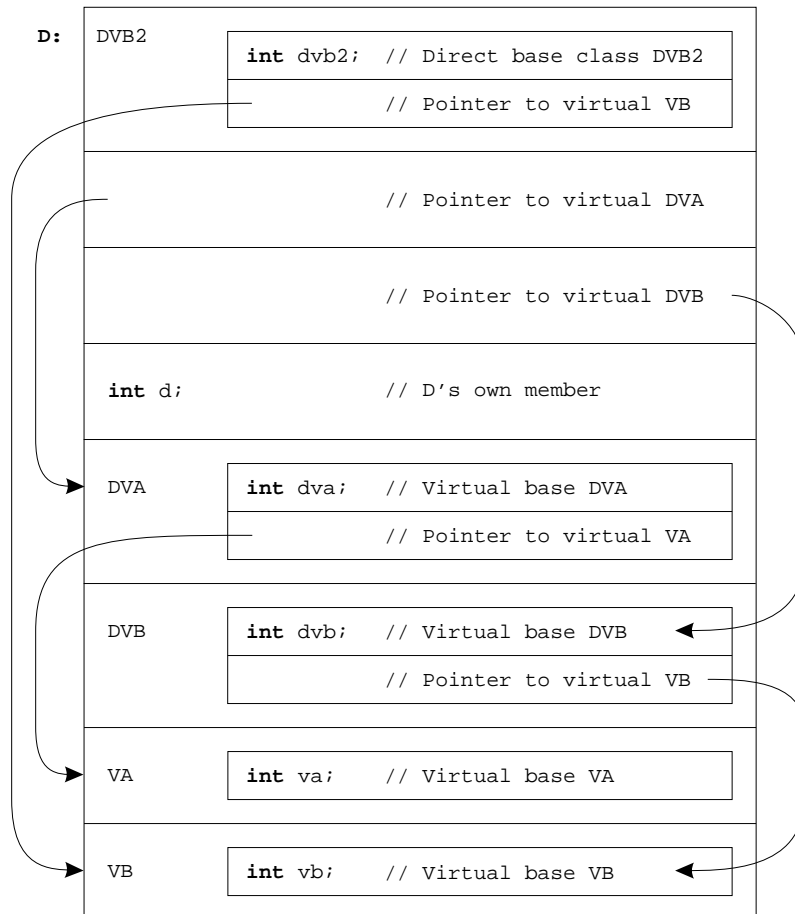
```
class VA { int va; };
class VB { int vb; };
class DVA: virtual VA { int dva; }; // From virtual VA
class DVB: virtual VB { int dvb; }; // From virtual VB
class DVB2: virtual VB { int dvb2; }; // From virtual VB
class D: virtual DVA,
    virtual DVB,
    virtual VB,
    DVB2 { int d; };
```

The following output shows the class lattice that the compiler produces for this example. (See toggle `Print_lattices` in Chapter 4: *Using Compiler Toggles* for more information about printing class lattices):

```
D {d}
-----
v:DVA {dva} v:DVB {dvb} v:VB {vb} DVB2 {dvb2}
-----
v:VA {va}    v:VB {vb}                v:VB {vb}
```

In this example, there are two pointers to virtual base VB: one within DVB and one within DVB2. When an object of class D is constructed, both are initialized to point to the single occurrence of VB; there would have been yet another pointer to VB from D, but it is optimized away. A total of five virtual-base pointers are initialized when a D object is constructed.

Figure 10.5 shows the object layout for an instance of D.



**Figure 10.5** Object Layout for an Instance of Class D

