

# E

# Manual Template Instantiation

---

This appendix describes how to use compiler options and pragmas to manually control instantiation of templates. It contains the following sections:

§E.1: *Overview*

§E.2: *Preventing Duplicate Function Definitions with Toggle Auto\_func\_instantiation*

§E.3: *Class Templates and Duplicate Definitions*

§E.4: *Selective Template Instantiation*

§E.5: *Checking Template Usage: Toggle Print\_template\_usage*

---

**Note:** If you are not using the MetaWare linker, you must manually instantiate templates as described in this appendix, in order to avoid the single-copy problem. You must specify option **-Hnocomdat** when you use this solution. For background information about the single-copy problem, see §8.8.1: *The Single-Copy Problem*.

---

---

## E.1 Overview

*Locally defined templates* If you are using templates defined locally within a single source module, there is no single-copy problem and no special care is needed. Each source module has its own templates and instances, as shown here:

*File 1.cpp:*

```
template <class T> T min(T x, T y) {...}
...
int i = min(1,2) + min(3,4);
```

File 2.cpp:

```
template <class T> T max(T x, T y) {...}
...
int j = max(1,2) + max(3,4);
```

In this example, each module contains an instance of a template function: `min` for 1.cpp and `max` for 2.cpp.

*Duplicate definitions* The single-copy problem arises only when you are using the same template in separate source modules, as shown here:

File `hdr.h`:

```
template < class T > T min(T x, T y) {...}
template < class T > T max(T x, T y) {...}
```

File 1.cpp:

```
#include "hdr.h"
int i = min(1,2) + max(3,4);
```

File 2.cpp:

```
#include "hdr.h"
int j = min(1,2) + max(3,4);
```

In this example, the object modules for 1.cpp and 2.cpp both contain definitions of `min(int,int)` and `max(int,int)`, and the linker will complain about a duplicate definition.

---

## E.2 Preventing Duplicate Function Definitions with Toggle `Auto_func_instantiation`

A way to prevent duplicate definitions is to turn Off automatic template instantiations when compiling. You turn them Off for template functions by turning Off `Auto_func_instantiation` as shown here:

File 1.cpp:

```
#pragma Off(Auto_func_instantiation)
#include "hdr.h"
int i = min(1,2) + max(3,4);
```

*File 2.cpp:*

```
#pragma Off(Auto_func_instantiation)
#include "hdr.h"
int j = min(1,2) + max(3,4);
```

The toggle must be turned `Off` in the definition of the function template — not at the reference to the function template. Essentially, the compiler records the setting of the toggle when it encounters the function template. For a non-inline function template encountered with the toggle `Off`, there are no instantiations of the function template bodies in the source code.

But now you have two modules, both of which make external references to `min(int,int)` and `max(int,int)`, and the linker will complain about missing functions.

You can provide the necessary functions as follows. In a single new source module, include calls (with the appropriate arguments) to the function templates, and keep the `Auto_func_instantiation` toggle `On`. The object code for this new module contains the required instantiated functions.

We recommend you place these calls to the function templates in the module in such a way that the functions are never actually called at run time, and the code for calling them is never actually generated by the compiler.

The following example shows a convenient way to do this with macros:

*File fti.cpp:*

```
#include "hdr.h"
#define DONTCALL(f) (0 && f)
static void dummy() {
    DONTCALL(min(1,1));
    DONTCALL(max(1,1));
};
```

Because of the conditional expression in the macro, the body for `dummy()` is empty. Yet the object module for this source module contains the definitions of `min(int,int)` and `max(int,int)`.

## E.3 Class Templates and Duplicate Definitions

A class template must always be instantiated when it is used — because the resultant type is required for further processing — but the single-copy problem applies to the non-inline class member functions and static data members. If you use a class template only locally to a source module there is no problem; the problem arises when you share an instantiation of a member function or data member.

As with function templates, the compiler by default always generates instances of class members when the class is used in a source module, and also when the definitions of those members are presented to the compiler. You can omit the definitions if you want.

You can turn off this automatic instantiation by turning Off toggle `Auto_class_member_instantiation`. Then you must provide another source module (with the toggle On), in which you define the class members, as shown in Example E.1.

### Example E.1 Turning OFF Automatic Instantiation

*File stack.h:*

```
template < class T > class stack {
    T *v, *p; int sz;
public:
    stack(int);
    ~stack() { delete[] v; }
    void push(T);
    T pop() { return *--p; }
};

template<class T> void stack<T>::push(T a) {
    *p++ = a;
}

template<class T>      stack<T>::stack(int s) {
    v = p = new T[sz=s];
}
```

File 3.cpp:

```
#pragma Off(Auto_class_member_instantiation)
#include "stack.h"
f() {
    stack<int> I(10);
    stack<float> F(20);
    I.push(1);      F.push(1.0);
    I.pop();        F.pop();
}
```

File 4.cpp:

```
#pragma Off(Auto_class_member_instantiation)
#include "stack.h"
f() {
    stack<int> I(10);
    stack<float> F(20);
    I.push(2);      F.push(2.0);
    I.pop();        F.pop();
}
```

---

In Example E.1, both modules contain external references to the following:

```
stack<int>::push(int)
stack<int>::stack(int)
stack<float>::push(float)
stack<float>::stack(float)
```

Note that there are no references to the **inline** members (such as `pop`) of `stack<int>` and `stack<float>`. These **inline** members are in fact generated by the compiler and inlined. The external references are made only to the non-**inline** functions and static data members.

You can provide the `push( )` and constructor functions by compiling the following module, which simply refers to the two stack types:

File `cti.cpp`:

```
#include "stack.h"
typedef stack<int>    dummy1;
typedef stack<float> dummy2;
```

## E.4 Selective Template Instantiation

Regardless of how toggle `Auto_class_member_instantiation` is set, you can ensure or suppress the instantiation of selected template types with pragmas `Ensure_instantiation(T1, T2, ... Tn)` and `Exclude_instantiation(T1, T2, ... Tn)`, where `T1, T2, ... Tn` are names of specific template types.

*Ensure\_instantiation* When you use pragma `Ensure_instantiation`, the class members for each template type in its argument list are instantiated. Here is an example:

```
#pragma Off(Auto_class_member_instantiation)
template <class T> struct Q {
    int f();
};
template <class T> Q<T>::f() {
    return sizeof(T);
}
typedef Q<int> x;
typedef Q<double> y;
typedef Q<Q<int> > z;
// Make sure Q<double> and Q<Q<int> > are
// instantiated
#pragma Ensure_instantiation(Q<double>, Q<Q<int> >)
// Q<int> will NOT be instantiated, because it is not
// listed
```

*Exclude\_instantiation* When you use pragma `Exclude_instantiation`, you suppress instantiation of the class members for each template type in its argument list.

You can use pragma `Exclude_instantiation` to resolve the single-copy problem in programs compiled from two or more source modules. If you find, after linking the program, that a template has been instantiated in two or more object files, add pragma `Exclude_instantiation` to all but one of the source files and recompile.

In the following example, *File 1* and *File 2* both contain an instantiation of `Q<int>`:

*File x.h:*

```
template <class T> struct Q {
    int f();
}
template <class T> Q<T>::f() {
    ...
}
```

*File 1:*

```
#include "x.h"
Q<int> x;
```

*File 2:*

```
#include "x.h"
Q<int> y;
```

To resolve the problem, modify *File 2* as follows, and recompile:

*File 2:*

```
#include "x.h"
Q<int> y;
#pragma Exclude_instantiation(Q<int>)
```

*Forcing instantiation of template types* You can use toggle `Ensure_instantiation` to force instantiation of template types. Any template used within the scope of this toggle gets instantiated. Here is an example:

```
#pragma Off(Auto_class_member_instantiation)
template <class T> struct Q {
    int f();
}
template <class T> Q<T>::f() {
    return sizeof(T);
}
typedef Q<int> x;
typedef Q<double> y;
#pragma On(Ensure_instantiation)
// Q<int> and Q<Q<int>> will both be instantiated
typedef Q<Q<int>> z;
#pragma Off(Ensure_instantiation)
```

With toggle `Ensure_instantiation`, it is harder to predict exactly what will be instantiated than with `pragma Ensure_instantiation`, because one template usage can indirectly cause other templates to be used. In the last example, the use of `Q<int>` causes the expansion of `Q`'s class body, in which any other template types used are then instantiated.

---

## E.5 Checking Template Usage: Toggle `Print_template_usage`

To get an idea of which templates were used in a compilation, turn On toggle `Print_template_usage`. This toggle prints both function templates and class templates. It also prints the members of a class template, indented below the class template. A single character in the leftmost column indicates the following:

- `I` means the compiler instantiated this
- `e` means undefined (it is external) and you must supply it
- `U` means the user defined a special version of a function, class member, or class template

The following output shows the result of compiling `3.cpp` (see Example E.1):

```
Template usage:
I    stack<int>                (at "3.cpp",L5/C7)
e    stack(int)                (at "stack.h",L4/C11)
e    void push(int)            (at "stack.h",L6/C16)
I    stack<float>              (at "3.cpp",L5/C25)
e    stack(int)                (at "stack.h",L4/C11)
e    void push(float)          (at "stack.h",L6/C16)
```

This output shows that the two `stack` classes are instantiated (`I`) but the two member functions are external (`e`).



When you compile `cti.cpp` (see discussion following Example E.1), you see the two `stack` classes instantiated:

```
Template usage:
I      stack<int>                (at "cti.cpp",L2/C11)
I      stack(int s)              (at "stack.h",L10/C26)
I      void push(int a)          (at "stack.h",L9/C26)
I      stack<float>              (at "cti.cpp",L3/C11)
I      stack(int s)              (at "stack.h",L10/C26)
I      void push(float a)        (at "stack.h",L9/C26)
```

