

# 8

## C++-Specific Issues

---

This chapter presents information specific to compiling and linking C++ programs. It contains the following sections:

§8.1: *Compiling and Linking C and C++ Modules*

§8.2: *Using the Built-In Inliner Versus the Stand-Alone Inliner*

§8.3: *Using the Name Unmangler*

§8.4: *Using Exception Handling*

§8.5: *Generating Run-Time Type Information (RTTI)*

§8.6: *Switching to New-Style Type Cast Notation: Toggle Try\_static\_cast*

§8.7: *Limitations in Pointer-to-Member Comparison*

§8.8: *Using Templates*

---

### 8.1 Compiling and Linking C and C++ Modules

High C/C++ includes both a C compiler and a C++ compiler in one executable. C++ has its own set of standard header files that are distinct from those of C. C++ also has its own run-time library.

The driver checks the file extension of the source file you specify on the command line to determine whether it should be compiled as C or C++ code. See §2.2.4: *Driver Input Files* for more information.

*Symbol CPLUS* The CPLUS symbol in the configuration (.cnf) file tells the High C/C++ compiler to compile and link C++ files. The following happens when CPLUS is set:

- The C++ header files, such as `iostream.h`, are placed on the compiler's search path.
- The C++ library files are included when a program is linked.

---

**Note:** The `CPLUS` symbol must be set when you compile a C++ program. If it is not set, the compilation or the link might fail unless you supply option `-Icpp_header_directory` to access the C++ header files (if you need them). To gain access to any of the C++ library functions, you must also specifically include the C++ libraries in the link.

---

`CPLUS` is set by default when you install the High C/C++ compiler. If you have unset it for any reason, you can reset it on the command line with option `-Hcplus` (see Chapter 3: *Using Compiler Options*). You can also set `CPLUS` by assigning it a non-zero value in the configuration (`.cnf`) file.

*Do not use these identifiers in C* The C++ library files contain identifiers that intrude upon the name space of ANSI Standard C programs. Identifiers with these names should *not* be used in C if you are going to mix C with C++:

`cin`   `cout`   `cerr`   `clog`

*Symbolic constants*   *\_\_CPLUSPLUS\_\_*   *and*   *\_\_cplusplus* Symbolic constants `__CPLUSPLUS__` and `__cplusplus` are defined when you compile C++ modules. You can use these constants in your programs to conditionally compile C or C++ code.

---

## 8.2 Using the Built-In Inliner Versus the Stand-Alone Inliner

The High C/C++ compiler has an optional optimization phase known as the *function inliner*, or *inliner* for short. This stand-alone inliner is described in Appendix C: *Using the Optional Inliner*.

---

### 8.2.1 Optional Stand-Alone Function Inliner

You can use the stand-alone inliner to inline a function anywhere, even if the function is called prior to its own definition. This inlining is especially helpful with functions in template classes, because the expansion of separately defined class template functions must be delayed until the end of compilation, in case you have overridden a template function with a specific

version of your own. Thus, calls to the template function are emitted by the compiler before the function definition is seen.

*When to use the stand-alone inliner*

In most cases, you will not need to use the stand-alone inliner for C++ functions. Use the stand-alone inliner only if you want to inline the following:

- arbitrary functions that are not necessarily designated as **inline**
- functions whose bodies are presented after the compiler sees calls to those functions
- functions across compilation units (via “wide inlining”)

---

## 8.2.2 Built-in C++ Inliner

The High C/C++ compiler also includes a built-in inliner that provides most of the functionality of the stand-alone inliner. The inlining of C++ **inline** functions is performed automatically, so you do not need to invoke the stand-alone inliner to inline such functions.

*Size limit for inlining C++ functions*

The compiler limits the size of functions that are automatically inlined, to limit code size expansion in compiled code. You can change the maximum size of functions to inline with command-line option **-Hinlsize**. See Chapter 3: *Using Compiler Options*.

If you want a class function to be inlined but plan to present the body later, specify **inline** in both the class declaration and in the function definition.

```
struct s {
    inline int fnc();
};
main () {
    s x;
    x.fnc();
}
inline s::fnc() { return 0; }
```

---

## 8.3 Using the Name Unmangler

The High C/C++ compiler converts the name of each function to a *mangled name* that encodes the function type in the text of the name. This encoding

is necessary in C++ to support overloading multiple functions with the same name.

*Type-safe linkage* Name mangling makes type-safe linkage possible. This means you can safely call a function that is defined in another module, and not worry whether that function is declared with the same parameter types in the other module.

The High C/C++ name unmangler, **unmangle**, converts the mangled function and variable names generated by the C++ compiler into a form similar to the C++ declaration syntax. You can use **unmangle** as a filter to read linker output or assembler output. You can also use **unmangle** to help you perform other essential debugging-related tasks.

**unmangle** deciphers names generated by the MetaWare High C++ compiler. **unmangle** looks for certain key sequences in a name to determine whether it is a mangled name.

For example, names containing embedded `__of` or `@` strings are treated as mangled names. There might be other key sequences, which vary from system to system.

---

### 8.3.1 Invoking the Name Unmangler

You can invoke the name unmangler from the command line in two ways:

**Method 1** *Invoke the name unmangler with arguments.*

The command-line arguments are unmangled. You can supply any number of arguments; the unmangled form of each argument is printed to `stdout`, each on a separate line.

**Method 2** *Invoke the name unmangler without arguments.*

If there are no command-line arguments, **unmangle** reads data from `stdin`, parses the data into words (delimited by whitespace — spaces, tabs, or newlines), and attempts to unmangle each word. This technique allows **unmangle** to be used as a filter.

---

### 8.3.2 Examples of Name Unmangling

Given these C++ function declarations:

```
int fnc( int i1, int i2 );  
void fnc( short i1, short i2 );
```

the High C/C++ compiler mangles the function names like this:

```
_fnc__FiT1  
_fnc__FST1
```

and **unmangle** unmangles them like this:

```
fnc(int,int)  
fnc(short,short)
```

The unmangled names are fully qualified with class membership scope, if applicable.

For example, this command:

```
unmangle _j__lXFi
```

writes this to stdout:

```
X::j(int)
```

---

**Note:** **unmangle** does not try to convert names that do not match the format of a mangled name. Such names are simply echoed without modification.

---

**Caution:** **unmangle** expects mangled names to be correctly constructed. If there is an error in the mangled name, **unmangle** might get confused and produce unexpected output. This feature makes command-line usage hazardous, because a typographical error on the command line could result in an incorrect mangled name.

---

---

## 8.4 Using Exception Handling

This section describes how to compile classes that are exception-aware, and how to use them with existing classes that are not exception-aware.

---

### 8.4.1 Generating Exception-Aware Classes

An important part of exception handling is the clean-up of partially constructed or destructed objects. To properly achieve this clean-up in the MetaWare implementation of exception handling, constructors increment a global “subobject count” by one and destructors decrement the count by one. For example, after construction of an object with a total of three subobjects, the subobject count is incremented by three.

*Definition of exception-aware* A class is *exception-aware* if its constructors increment the global subobject count and the destructor decrements the count.

So that code with exception-handling capability can properly interoperate with earlier code, the High C++ compiler provides a way to declare exactly which classes are exception-aware (see §8.4.2).

---

### 8.4.2 Making a Class Exception-Aware with `Toggle Exception_aware_class`

By default, a class is not exception-aware. You make it so by turning On `toggle Exception_aware_class`. Exception-aware classes can be intermixed with non-exception-aware classes within a class hierarchy.

#### 8.4.2.1 Completely Constructed Objects

*Exception unaware* For a completely constructed object, no portion of which is exception-aware, no destructors are run during clean-up. This is an advantage, because it saves time during clean-up.

*Somewhat exception aware* For a completely constructed object, a portion of which is exception-aware, all destructors are run during clean-up.

### 8.4.2.2 Partially Constructed Objects

For partially constructed objects, the presence of non-exception-aware classes introduces imprecision in the clean-up.

*When a partially constructed object is fully constructed*

A partially constructed object is considered to be fully constructed if all its exception-aware subobjects have been constructed. All destructors are run, even though the object is partially constructed.

For example, if all but the most-derived constructor completes, and the most-derived class is not exception-aware, the object appears fully constructed.

```
#pragma On(Exception_aware_class)
struct A { ~A(); };
struct B { ~B(); };
#pragma Pop(Exception_aware_class)
struct C:A, B { ~C(); C() { throw 1; } };
C x;
```

Even though C's constructor throws the exception, x is taken as fully constructed and the C subobject is fully destructed.

*Ensuring a fully constructed object*

If you are concerned about a destructor running on a non-exception-aware class, be sure that the most-derived class is exception-aware. In that way, you ensure that if all the exception-aware objects are constructed, the object is fully constructed.

For a partially constructed object with some exception-aware subobjects not constructed, a non-exception-aware subobject is not destructed unless its construction was followed by the successful construction of an exception-aware subobject.

For example:

```
#pragma On(Exception_aware_class)
struct A { ~A(); };
#pragma Pop(Exception_aware_class)
struct B { ~B(); };
struct C { A a1; B b; A a2; };
C x;
```

If the construction of the second A-type class, a2, throws an exception, the b subobject is not destructed, even though it was fully constructed.

- When to make a class exception-aware* You should make a class exception-aware if either of the following is true:
- You do not want the destructor to run on an unconstructed object; for example, if the destructor accesses a pointer allocated during construction.
  - You want the destructor always to run on a constructed object; for example, if the destructor frees storage allocated during construction.

---

## 8.4.3 Interacting with Previously Compiled Classes

Because exception handling is new, there exist classes (such as those found in third-party libraries) that are not exception-aware. This is why none of the standard classes in the High C++ run-time library (such as `iostreams`) are exception-aware by default. To make them exception-aware would introduce problems of binary incompatibility with any third-party libraries that might make use of those classes.

*Option -Hexcept* Option **-Hexcept** makes it safer to use toggle `Exception_aware_class`, and also enables you to choose between the exception-aware version of the run-time library and the version that is not exception-aware. Specifying **-Hexcept** turns on toggle `Exception_aware_class`

### 8.4.3.1 Wrapping Header Files

Header files for third-party libraries that do not contain exception-aware classes *must not* be compiled with toggle `Exception_aware_class` turned On (whether or not you use option **-Hexcept**).

You must protect these header files by *wrapping* them with other include files that turn Off and On toggle `Exception_aware_class`. To see how this works, specify **-Hon=list** when you compile.

*Option -Hnonexcept\_wrap* To direct the compiler to wrap an include file, insert a character string `xxxx` in the file's name, then specify option **-Hnonexcept\_wrap=xxxx** when you compile. For example, given this command line:

```
hc fnc.cpp -Hexcept -Hnonexcept_wrap=nonexc
```

any include file with the string "nonexc" in its name is wrapped with **#pragma Off**(`Exception_aware_class`) at the beginning and **#pragma Pop**(`Exception_aware_class`) at the end.



Option `-Hnonexcept_wrap` behaves just like option `-Hc_wrap`, except that option `-Hnonexcept_wrap` uses wrapper files called `exc_off.h` and `exc_pop.h`.

---

## 8.5 Generating Run-Time Type Information (RTTI)

*Non-class types* For a non-class type, use `typeid` to generate RTTI.

*Class types* For a class type, turn On toggle `Run_time_type_info` to ensure that RTTI is generated and stored in the class's virtual-function tables. (The default for toggle `Run_time_type_info` is On).

For such a class *D*, polymorphic `typeid`s on a subobject of *D* return the type object for *D*, and `dynamic_cast` works for such subobjects. Also, classes satisfying the single-copy heuristic have just one copy of the RTTI. For a discussion of the single-copy problem, see §8.8.1: *The Single-Copy Problem*.

When toggle `Run_time_type_info` is Off, RTTI is generated only as needed by `typeid`, so you cannot guarantee the presence of a pointer to the RTTI in the virtual-function tables. Thus, a polymorphic `typeid` can return a NULL reference.

In particular, a compilation unit that generates virtual-function tables for the class, but lacks a reference to `typeid`, also lacks a pointer to the RTTI in the virtual-function table.

*When not to use toggle Run\_time\_type\_info* It is best leave toggle `Run_time_type_info` On for all your classes. Be aware, however, of two things:

- You should not turn On toggle `Run_time_type_info` globally for header files from third parties.  
Because RTTI is a new feature, third-party libraries might not contain RTTI. If you turn the toggle On around such headers, the compiler might generate a reference to the RTTI for a class satisfying the single-copy heuristic when it generates the RTTI for one of your classes, and your program link will fail.
- You might want to turn Off toggle `Run_time_type_info` around classes with virtual functions that do not satisfy the single-copy heuristic. For such classes, the RTTI and virtual-function tables are generated for every compilation unit containing the definition of the class, even if the

class is not used. For these classes, the compiler warns that each compilation unit will have a copy of the virtual-function tables.

For a discussion of the single-copy problem, see §8.8.1: *The Single-Copy Problem*.

---

## 8.6 Switching to New-Style Type Cast Notation: Toggle Try\_static\_cast

The proposal of the X3J16 C++ standard for the new-style High C/C++ cast notation syntax suggests the following: When moving from the old-style casts to the new, replace all old-style casts with the new notation **static\_cast**, and see what diagnostics you get when you recompile.

To facilitate this replacement process, High C/C++ provides the toggle `Try_static_cast` (default `Off`). The effect of this toggle is to treat all C++ casts of the form `(Type) expr` as **static\_cast**, and warn at each line containing a cast (so you can find them all). An error might also be generated if the **static\_cast** is not valid.

---

## 8.7 Limitations in Pointer-to-Member Comparison

When a pointed-to member function is a virtual function, a comparison between two pointer-to-member functions might not yield the results you expect.

If one source file sets the value and a different source file does the comparison, the comparison fails because different function “thunks” are used to implement the call to the virtual function.

*File* `hdr.h`:

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

File 1.cc:

```
#include "hdr.h"
extern void (s::*p)();  extern sub();
main () {
    sub();               // Set the pointer
    if (p==s::fnc) printf("OK\n"); // Will not print OK
}
```

File 2.cc:

```
#include "hdr.h"
void (s::*p)();
sub() { p = s::fnc; }
```

---

## 8.8 Using Templates

Templates are a powerful feature of C++. With templates you can define generic functions or classes. Unlike macros, which are sometimes used to do the same thing (for example, `generic.h`), instantiations of templates are accompanied by full type checking.

The **Annotated C++ Reference Manual** makes reference to “function templates” and “template functions.”

*Function templates*    *A template function* is an instantiation of a function template, so we use the  
                                 versus    term “instantiation” or “instance” rather than “template function.”  
*template functions*

*Class templates*    The terms “class templates” and “template classes” are related the same  
                                 versus    way. *A template class* is an instantiation of a class template, so we use the  
*template classes*    term “instantiation” or “instance” rather than “template class.”

The rest of this section explains how to use templates, discusses different types of templates, and suggests methods to control various problems you might encounter.

---

### 8.8.1 The Single-Copy Problem

There is a problem with using templates: you must ensure that for each distinct set of parameters to a template, there is only a single copy of an

instantiated function or static class data member. We call this the “single-copy problem.”

The default operational mode of the High C/C++ compiler is to always generate instances of function templates and classes when they are used in a source module — one instance in the source module is generated per set of parameters to the template.

High C/C++ currently supports the following solutions to the single-copy problem:

- Merging of global template instantiations, also referred to as the COMDAT solution. This is the default solution for template instantiation and virtual function table generation when you use the MetaWare linker. This solution is discussed below in §8.8.2: *Using COMDAT Sections to Merge Global Instantiations*.
- Controlling instantiations at compile time. This is an alternate method for template instantiation. You should use this solution if you are not linking with a MetaWare linker. You should specify option `-Hnocomdat` when you use this solution. See Appendix E: *Manual Template Instantiation* for more information.

---

## 8.8.2 Using COMDAT Sections to Merge Global Instantiations

The COMDAT solution is the default solution for template instantiation and virtual function table generation when you use the MetaWare linker. If you plan to use a third-party linker, you should adopt the manual instantiation techniques outlined in Appendix E: *Manual Template Instantiation*.

Some object module formats (OMFs) allow the linker to choose one of the multiple copies of a structure and discard the others, so that only one copy appears in the final linked program. This is what is known as the *COMDAT solution* to the single-copy problem.

---

**Note:** MetaWare has made extensions to ELF to support COMDAT for all High C/C++ compilers targeting the Power PC.

---

*Limitations of COMDAT* COMDAT does not allow you to do the following:

1. Declare apocryphal some class template *T* in Module 1 and provide definitions for its member functions; then
2. Declare class template *T* in Module 2 *without* member function definitions, and instantiate template *T* in Module 2.

---

### 8.8.3 Problems with Matching `const` Parameters

Instantiating a function template requires an exact match of the function formal-parameter types and the actual-parameter types. This requirement can be a nuisance. For example, suppose you generalize the ANSI Standard `C bsearch()` routine with a template:

```
template <class T> T *bsearch(const T key,
                             const T *base,
                             int nmemb);
```

Because you used `const`, your generic search routine “promises” never to clobber the elements being searched, nor the key. This is a valuable promise. But when you try to call `bsearch()`:

```
extern int a[10];
int *solution = bsearch(123, a, 10);
```

the call fails, because 123 is not a `const`, and `a` is not of type pointer-to-`const`. In fact, you must write this:

```
extern int a[10];
int *solution = bsearch((const int)123,
                       (const int *) a, 10);
```

to successfully call `bsearch()`.

One way to alleviate this problem is to define an additional `inline` template that allows non-`const` parameters, but which calls the “real” one:

```
template <class T> inline T *bsearch(T key,
                                     T *base,
                                     int nmemb) {
    bsearch((const T)key, (const T*)base, nmemb);
}
```

This allows the call to `bsearch(123, a, 10)` to succeed. The second template is called, which turns around and calls the first. Because the second template is inlined, there is no performance penalty.

To permit a non-**const** parameter to be “matched” with a **const** parameter in a function template, turn On the High C/C++ toggle `Template_trivial_conversions`. Doing this allows so-called trivial conversions (in the parlance of the **Annotated C++ Reference Manual**) when arguments are matched to function templates. This conversion allows the call to `bsearch(123, a, 10)` to work without introducing the intermediate **inline** template.