
C++ Runtime Environment Programming Guide

Compiler Tools



2009-10-09



Apple Inc.
© 2005, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Mac, Mac OS, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

iPhone and Numbers are trademarks of Apple Inc.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction 5

Organization of This Document 5

Overview of the C++ Runtime Environment 7

Targeting Mac OS X v10.3.8 and Earlier 7

Targeting Mac OS X v10.3.9 and Later 7

 The Static C++ Runtime 7

 The Dynamic C++ Runtime 8

Checking Version Numbers 9

Deploying Applications With the C++ Runtime 11

Caveats for Using the New C++ Runtimes 11

Deploying With the New Dynamic Runtime 12

Deploying With the New Static Runtime 12

 Additional Caveats for Using the New Static Runtime 12

Creating Compatible Libraries 15

Avoiding libstdc++ in Your Library Interfaces 15

Limiting the Scope of Callbacks 16

Controlling Symbol Visibility 17

Using GCC 4.0 to Mark Symbol Visibility 17

 Compiler Flags 17

 Visibility Attributes 18

 Pragmas 19

Reasons for Limiting Symbol Visibility 19

Reasons for Making Symbols Visible 20

Visibility of Inline Functions 20

Symbol Visibility and Objective-C 21

Document Revision History 23

Introduction

All nontrivial C++ programs must be linked with the standard C++ library, also known as the C++ runtime. This library includes the implementations for components such as I/O streams, STL container classes, the low-level exception handling runtime, and other low-level types and classes.

This document provides background information about the C++ runtime that you may find useful if you are developing C++ programs. It also offers information about Apple's C++ support and offers tips on how to write more compatible C++ libraries and programs.

Organization of This Document

Information about the C++ runtime environment is provided in the following articles:

- [“Overview of the C++ Runtime Environment”](#) (page 7) describes the state of the C++ runtime environment, including issues surrounding binary compatibility and general support.
- [“Deploying Applications With the C++ Runtime”](#) (page 11) provides guidelines for deploying applications using either the static or dynamic C++ standard library.
- [“Creating Compatible Libraries”](#) (page 15) provides tips on how to make sure your own dynamic shared libraries retain their binary compatibility even when changes occur to the C++ runtime.
- [“Controlling Symbol Visibility”](#) (page 17) describes new tools and techniques for controlling the symbols exported by your C++ code.

Overview of the C++ Runtime Environment

The C++ runtime environment has evolved over the course of Mac OS X development. Early versions of the library were shipped as a static archive file while the latest version is delivered as a dynamic shared library. The following is a summary of the history of the runtime:

- **Mac OS X v10.3.8 and earlier** - the C++ standard library is available only as a static archive file. It can be compiled using GCC 3.3.
- **Mac OS X v10.3.9 and later** - the runtime is delivered as a dynamic shared library. System frameworks are modified to use the dynamic library but the static library is also available.

The following sections provide more detailed information about the dynamic and static C++ runtimes. An additional section provides information about checking the version numbers of the runtime.

Targeting Mac OS X v10.3.8 and Earlier

In Mac OS X v10.3.8 and earlier, the standard C++ library is packaged as the static archive file `libstdc++.a`. This library is designed with multiple clients in mind and provides syncing of I/O streams as well as hidden symbols to alleviate namespace conflicts. Symbols in the library are marked `__private_extern__` to prevent them from being exported to other code modules. Versions of this library are available in all versions of Mac OS X but programs must be built using the GCC 3.3 compiler.

Targeting Mac OS X v10.3.9 and Later

In Mac OS X v10.3.9 and later, you have the option of linking against a static or dynamic version of the C++ runtime.

The Static C++ Runtime

With the introduction of Xcode Tools 2.3, a new version of the standard C++ library is made available in the static archive file `libstdc++-static.a`. This new static library is closer in nature to the dynamic shared library introduced in Mac OS X v10.3.9 than to the previous static library. It conforms to the Itanium C++ ABI and requires the use of GCC 4.0 for compilation. Programs linking to the library must run in Mac OS X v10.3.9 or later. For more information, see [“Deploying With the New Static Runtime”](#) (page 12).

The Dynamic C++ Runtime

In Mac OS X v10.3.9 and later, the C++ runtime is available as a dynamic shared library `libstdc++.dylib`. This change in packaging brings the C++ runtime in line with the C runtime, which has always been packaged as part of the dynamic shared library `libSystem.dylib`. The dynamic library conforms to the Itanium C++ ABI, which is a standard for compiled C++ code that provides better link-compatibility between C++ binaries. Because it is shared, the namespace limitations present with static versions of the library are gone and symbols are no longer marked `__private_extern__`. For information about designing and using C++-based dynamic libraries, see *Dynamic Library Programming Topics*.

Note: To build programs that link to `libstdc++.dylib`, you must have GCC 4.0, which is provided with Xcode Tools in Mac OS X v10.4. You use this compiler along with the SDKs provided by Apple to build your binary for 10.3.9. For more information, see “[Deploying Applications With the C++ Runtime](#)” (page 11).

Advantages of a Shared C++ Runtime

Packaging the standard C++ library as a dynamic shared library is important for ensuring performance and correctness of your program. If your executable links to `libstdc++.a` and to several dynamic shared libraries, and one or more of those shared libraries also links to `libstdc++.a`, then multiple copies of the library are loaded into memory at runtime.

For correctness, all components of a program should use the exact same copy of the standard C++ library. The reason is that some data in the C++ runtime must be shared by all components or unexpected results may occur. For example, if two components both use the C++ I/O mechanism to write to standard output, the results can become garbled if they use different buffers.

For performance, having multiple copies of the same library leads to increased disk activity as each copy is read into memory. The extra copies can also lead to increased paging and cache misses due to the increased memory footprint of the application. Eliminating the duplicated libraries can reduce the footprint of your application dramatically. For example, consider the following “hello world” program compiled with `libstdc++.a` and GCC 3.3:

```
int main ()
{
    std::cout << "Hello, World!\n" << std::endl;
    return 0;
}
```

The size of the resulting binary on a Mac OS X v10.4 system is 650 KB. The same program compiled on the same system using GCC 4.0 and `libstdc++.dylib` is 17 KB.

Binary Compatibility

GCC 4.0 conforms to the Itanium C++ ABI, which is a standard for compiled C++ code. The specifications for this standard are maintained by a multi-vendor consortium and cover issues such as name mangling, class member layout, virtual method invocation protocols, exception handling, and runtime type information (RTTI) formats. You can find the latest version of this specification at <http://www.codesourcery.com/cxx-abi/abi.html>.

Because GCC 4.0 conforms to the Itanium C++ ABI, C++ objects are link-compatible with objects built by other Mac OS X compilers that conform to this specification. Apple guarantees that future releases of GCC for Mac OS X will also conform to the Itanium C++ ABI. This means that developers may safely ship dynamic shared libraries whose interfaces involve C++ classes, albeit with some caveats:

- Apple guarantees ABI stability only for core language features. It does not guarantee stability for library classes, including `std::string`, `std::map<T>`, and `std::ostream` among others.
- Apple does not guarantee binary compatibility between different major versions of `libc++`. GCC 4.0 ships with version 6.0.3 of the library. If a new major version (version 7.0.0) is released in the future, that library is not guaranteed to be compatible with 6.x versions.
- Binary compatibility between different versions of a third-party dynamic shared library also depends on the design of the library, not just on the compiler support for a standard ABI. You must ensure that you do not introduce compatibility issues.

If you are designing a dynamic shared library for distribution, it is still your responsibility to ensure that you do not create binary compatibility problems. For example, you should not introduce member variables or virtual methods to a base class. Doing so causes a fragile base class problem and requires clients of the library to be recompiled.

For more information on binary compatibility, see [“Creating Compatible Libraries”](#) (page 15).

Checking Version Numbers

Traditionally, Xcode has shipped with matching versions of the GCC compiler and the C++ runtime. However, Xcode 3.0 broke with this tradition by shipping version 4.2 of GCC with version 4.0 of the runtime. You may need to check for this configuration to write portable code. The compiler macros `__GNUC__` and `__GNUC_MINOR__` are defined as the major and minor version numbers of the compiler itself. In the SDKs included in Xcode 3.2 and later, the macros `__GNUC_LIBSTD__` and `__GNUC_LIBSTD_MINOR__` are defined as the major and minor version numbers of the runtime. All four macros are defined at compile time only, not at run time.

The following example shows how to use macros to check the version of the C++ runtime:

```
#if (__GNUC_LIBSTD__ > 4) || ((__GNUC_LIBSTD__ == 4) && (__GNUC_LIBSTD_MINOR__
    >= 2))
#include <ext/atomicity.h>
#else
#include <bits/atomicity.h>
#endif
```


Deploying Applications With the C++ Runtime

In most situations, you should never have to worry about linking against the C++ runtime environment. When you build your C++ code, GCC chooses the most appropriate version of the C++ standard library and links to it. For applications compiled using GCC 3.3, the compiler links to the static C++ standard library `libstdc++.a`. For applications compiled using GCC 4.0, the compiler links to the dynamic C++ standard library `libstdc++.dylib` by default (`libstdc++.a` is also available as an option).

Note: Whenever you build C++ programs for earlier versions of Mac OS X, be sure to use the SDKs that come with XCode Tools. For more information on using these SDKs, see *SDK Compatibility Guide*.

The following sections describe issues you may encounter when linking against the dynamic runtime or the new static runtime. There are no known issues worth noting for deploying code using the original static C++ runtime.

Caveats for Using the New C++ Runtimes

The following caveats apply to the new static and dynamic C++ runtimes.

- If your program must run on versions of Mac OS X prior to 10.3.9, as well as Mac OS X v10.3.9 and later, you must use the GCC 3.3 compiler and build your program using the original static C++ standard library (`libstdc++.a`).
- Applications that run on Mac OS X 10.3.9 must not try to use the `long double` type or any of the functions associated with it. Support for the 128-bit `long double` type was not introduced until Mac OS X 10.4. Similarly, full support for `sinl`, `expl`, and all other `long double` math functions defined by the C standard is available only in 10.4 and later.
- When overriding the `new` and `delete` operators, your overrides will be visible across shared library boundaries (application wide) unless you specifically takes steps to make them hidden (such as with an (un)export list). Even uses of `new/delete` in system frameworks, in the dynamic C++ runtime (even if you are only indirectly linked to it) or in other modules (possibly linked statically or dynamically to the C++ runtime) will use your overrides, unless that module also has overridden `new/delete` operators.

Unless you are sure that code modules are using the same operator `new` and `delete`, do not transfer memory ownership across shared library boundaries.

- Multithreaded applications which use the standard stream (`std::cin`, `std::cout`, and so on) must explicitly synchronize their use of these global objects if there is any chance that two threads may simultaneously access one of them. Functions that set global state (such as `set_terminate`, `set_unexpected`, and so on) must also be explicitly synchronized if multiple threads can call them at once. In general:
 1. It is safe to simultaneously call `const` and `non-const` methods from different threads to distinct objects.

2. It is safe to simultaneously call `const` methods, and methods otherwise guaranteed not to alter the state of an object (or invalidate outstanding references and iterators of a container) from different threads to the same object.
3. It is not safe for different threads to simultaneously access the same object when at least one thread calls `non-const` methods, or methods that invalidate outstanding references or iterators to the object. The programmer is responsible for using thread synchronization primitives to avoid such situations.

Deploying With the New Dynamic Runtime

This is the library used by default when using GCC 4.0 and is appropriate when targeting Mac OS X v10.3.9 and later. For a list of caveats for using this library, see [“Caveats for Using the New C++ Runtimes”](#) (page 11).

Deploying With the New Static Runtime

In Xcode 2.3, a new version of the static C++ standard library `libstdc++-static.a` was introduced. You can link against the new static C++ standard library using Xcode 2.3 or the command line. To link against the library using Xcode, simply set “C++ Standard Library Type” in the build settings to “Static”. To link against this library on the command line, use `gcc` instead of `g++` and append the following two commands to the end of the command: `-shared-libgcc` and `-lstdc++-static`. For example:

```
gcc one.o two.o -o my_application -shared-libgcc -lstdc++-static
```

Note: The extra options to specify the static C++ runtime must appear at the end of the command.

For a list of caveats for using this library, see [“Caveats for Using the New C++ Runtimes”](#) (page 11) and the next section.

Additional Caveats for Using the New Static Runtime

The new static C++ standard library marks all of its symbols as “hidden”, which means those symbols are not exported by an application that links to the library. It is still possible, however, for symbols from the C++ library to be exported by your application. For example, your application might include symbols from the library in header files or other visible locations. To prevent symbols from being accidentally exported, you must use an export list for any binaries that link against the static C++ library. (For more information about the use of export lists, see [“Reducing the Number of Exported Symbols”](#) in *Xcode Build System Guide*. For information on using visibility pragmas and visibility switches to control symbol visibility in your own code, see [“Controlling Symbol Visibility”](#) (page 17).)



Warning: Remember that the system frameworks continue to use the dynamic version of the C++ standard library. If you link your binary against the static C++ library and one or more system frameworks, you may run into namespace conflicts. You must ensure that no C++ symbols (symbols starting with `_Z`) are exported from binaries that use the static C++ standard library. If it does export such symbols, your binary may potentially stop working after the installation of a system upgrade, software update, or security update.

Callback functions (such as registered with `std::set_terminate`) are only effective when triggered within the same code module.

This static library has the same size disadvantages as the one used with `gcc 3.3`. For details about this issue, see [“Advantages of a Shared C++ Runtime”](#) (page 8).

Creating Compatible Libraries

The GCC 4.0 compiler ships with version 6.0.3 of the dynamic C++ runtime. Future minor versions of the library are guaranteed to retain binary compatibility with version 6.0.3; however, major versions are not guaranteed to be compatible, and developers should assume that a future compiler release will include an incompatible version of `libstdc++.dylib`.

If you distribute dynamic shared libraries that use the dynamic C++ runtime, changes to that runtime could potentially break clients of your library. For example, this can happen if your library's interface includes types or classes defined by the dynamic C++ runtime. The following sections explain ways to remain compatible between major updates to the dynamic C++ runtime.

Avoiding `libstdc++` in Your Library Interfaces

Because any component of the dynamic C++ runtime may change between major versions, your own dynamic shared libraries must avoid using classes, templates, and structures of the dynamic C++ runtime in their exported interfaces. The size and layout of standard C++ classes may change between different versions of the dynamic C++ standard library. If that happens and your library exports interfaces that rely on the current class information, clients of your library will break and require recompilation with an updated version of your library.

For any symbols your library exports to clients, you should follow these rules.

- Avoid inheriting from classes in the dynamic C++ standard library, such as `std::ostream`.
- In your class definition, avoid member variables (even private members) whose type is a class defined by the dynamic C++ runtime. For example, do not include member variables of type `std::auto_ptr`.
- Avoid using classes defined in the dynamic C++ runtime as arguments or return values of functions or methods.
- If your function or method throws exceptions, be sure your exception classes do not derive from dynamic C++ runtime classes, such as `std::runtime_error`.

Keep in mind that the prohibitions on using dynamic C++ runtime classes applies only to your exported symbols. Within your library's internal implementation, you are free to use the classes of `libstdc++` as you choose. Also keep in mind that these rules are to prevent clients of your library from breaking when the dynamic C++ runtime is updated. Your own library might still require recompilation.

Note: You do not need to avoid all C++ features in your public interfaces, but you do need to avoid using any classes declared in `libstdc++.dylib`.

Remember that a dynamic shared library's exported interface consists of all public symbols, not just classes and functions declared in the library's header files. In particular, template instantiations are public by default. So, if you instantiate a template with the declaration `std::vector<my_type>`, your library exposes a public instance of `std::vector` and will experience problems with any major version changes to the dynamic C++ runtime.

As you create your library, you should be mindful of which interfaces you want to make public and mark them appropriately. For information on how to limit the exported symbols in your library, see [“Controlling Symbol Visibility”](#) (page 17).

Limiting the Scope of Callbacks

In certain cases, a C++ application can supply code that is used by the dynamic C++ runtime itself. The most relevant example of this is when an application replaces the global `new` and `delete` operators. Rarer examples include I/O stream callbacks defined through `std::ios_base::register_callback` and user-defined facets imbued into the global locale.

If you implement your own version of `operator new`, that version is used by all other libraries that link with the same version of `libstdc++.dylib`. If this is not what you intended with your implementation, you should mark your version of the method with the `__private_extern__` tag to prevent it from ever being seen by other libraries.

For more information on controlling the visibility of symbols in your libraries, see [“Controlling Symbol Visibility”](#) (page 17).

Controlling Symbol Visibility

In ordinary C, if you want to limit the visibility of a function or variable to the current file, you apply the `static` keyword to it. In a shared library containing many files, though, if you want a symbol to be available in several files inside the library, but not available outside the library, hiding that symbol is more difficult. Most linkers provide convenient ways to hide or show all symbols in a module, but if you want to be more selective, it takes a lot more work.

Prior to Mac OS X v10.4, there were two mechanisms for controlling symbol visibility. The first technique was to declare individual symbols as private to the library but external to the current file using the `__private_extern__` keyword. This keyword could be used in the same places you would use either the `static` or `extern` keywords. The second technique was to use an export list.

An export list is a file containing the names of symbols you explicitly want to hide or show. Although symbol names in C are easily determined (by prepending an underscore character to the name), determining symbol names in C++ is far more complicated. Because of classes and namespaces, compilers must include more information to identify each symbol uniquely, and so compilers create what is known as a mangled name for each symbol. This mangled name is often compiler-dependent, difficult to deduce, and difficult to find within a large list of symbols defined by your library.

Luckily, GCC 4.0 provides some new ways to change the visibility of symbols. The following sections describe these new techniques along with reasons why this might be important to you.

Using GCC 4.0 to Mark Symbol Visibility

Beginning with Mac OS X v10.4, hiding C++ symbol names is much easier. The GCC 4.0 compiler supports new options for hiding or showing symbols and also supports a new pragma and compiler attributes for changing the visibility of symbols in your code.

Note: The following features are available only in GCC 4.0 and later. For information on how to use these features with Xcode, see *Executing Mach-O Files* in *Mach-O Programming Topics*. For general information about symbol definition and method implementation, see *Dynamic Library Design Guidelines* in *Dynamic Library Programming Topics*.

Compiler Flags

GCC 4.0 supports a new flag for setting the default visibility of symbols in a file. The `-fvisibility=vis` compiler option lets you set the visibility for symbols in the current compilation. The value for this flag can be either `default` or `hidden`. When set to `default`, symbols not explicitly marked as hidden are made visible. When set to `hidden`, symbols not explicitly marked as visible are hidden. If you do not specify the `-fvisibility` flag during compilation, the compiler assumes `default` visibility.

Note: The name `default` does not refer to compiler defaults. Like the name `hidden`, it comes from visibility names defined by the ELF format. A symbol with `default` visibility has the kind of visibility that all symbols do if no special mechanisms are used—that is, it is exported as part of the public interface.

The compiler also supports the `-fvisibility-inlines-hidden` flag for forcing all inline functions to be hidden. You might use this flag in situations where you want to use default visibility for most items but still want to hide all inline functions. For more information why this might be necessary for inline functions, see [“Visibility of Inline Functions”](#) (page 20).

Visibility Attributes

If you are compiling your code with GCC 4.0, you can mark individual symbols as default or hidden using the visibility attribute:

```
__attribute__((visibility("default"))) void MyFunction1() {}
__attribute__((visibility("hidden"))) void MyFunction2() {}
```

Visibility attributes override the value specified with the `-fvisibility` flag at compile-time. Thus, adding the `default` visibility attribute causes a symbol to be exported in all cases, whereas adding the `hidden` visibility attribute hides it.

Visibility attributes may be applied to functions, variables, templates, and C++ classes. If a class is marked as hidden, all of its member functions, static member variables, and compiler-generated metadata, such as virtual function tables and RTTI information, are also hidden.

Note: Although template declarations can be marked with the visibility attribute, template instantiations cannot. This is a known limitation and may be fixed in a future version of GCC.

To demonstrate how these attributes work at compile-time, take a look at the following declarations:

```
int a(int n) {return n;}

__attribute__((visibility("hidden"))) int b(int n) {return n;}

__attribute__((visibility("default"))) int c(int n) {return n;}

class X
{
public:
    virtual ~X();
};

class __attribute__((visibility("hidden"))) Y
{
public:
    virtual ~Y();
};

class __attribute__((visibility("default"))) Z
{
public:
    virtual ~Z();
};
```

```
};

X::~X() { }
Y::~Y() { }
Z::~Z() { }
```

Compiling this code with the `-fvisibility=default` flag would cause the symbols for functions `a` and `c` and classes `X` and `Z` to be exported by the library. Compiling this code with the `-fvisibility=hidden` flag would cause the symbols for the function `c` and the class `Z` to be exported.

Using the visibility attribute to mark symbols as visible or hidden is better practice than using the `__private_extern__` keyword to hide individual symbols. Using the `__private_extern__` keyword takes the approach of exposing all symbols by default and then selectively hiding ones that are private. In a large shared library, the reverse approach is usually better. Thus, it is usually better to hide all symbols and then selectively expose the ones you want clients to use.

To simplify the task of marking symbols for export, you might also want to define a macro with the `default` visibility attribute set, such as in the following example:

```
#define EXPORT __attribute__((visibility("default")))

// Always export the following function.
EXPORT int MyFunction1();
```

The advantage of using a macro is that if your code is also compiled on other platforms, you can change the macro to the appropriate keywords for the compilers on the other platforms.

Pragmas

Another way to mark symbols as default or hidden is with a new pragma in GCC 4.0. The GCC visibility pragma has the advantage of being able to mark a block of functions quickly, without the need to apply the visibility attribute to each one. The use of this pragma is as follows:

```
void f() { }

#pragma GCC visibility push(default)
void g() { }
void h() { }
#pragma GCC visibility pop
```

In this example, the functions `g` and `h` are marked as default, and are therefore exported regardless of the `-fvisibility` flag, while the function `f` conforms to whatever value is set for the `-fvisibility` flag. As the names `push` and `pop` suggest, this pragma can be nested.

Reasons for Limiting Symbol Visibility

It is good practice to export as few symbols as possible from your dynamic shared libraries. Exporting a limited set of symbols improves program modularity and hides implementation details. Reducing the number of symbols in your libraries also decreases the footprint of your library and reduces the amount of work that must be done by the dynamic linker. With fewer symbols to load and resolve, the dynamic linker is able to get your program up and running more quickly.

Reasons for Making Symbols Visible

Although it is likely that most C++ symbols in your shared library do not need to be visible, there are some situations where you do need to export them:

- If your library exports a C++ interface, the symbols associated with that interface must be visible.
- If your symbol uses runtime type identification (RTTI) information, exceptions, or dynamic casts for an object that is defined in another library, your symbol must be visible if it expects to handle requests initiated by the other library. For example, if you define a catch handler for a type in the C++ standard library, and you want to catch exceptions of that type thrown by the C++ standard library, you must make sure that your `TypeInfo` object is visible.
- If you expect the address of an inline function used in different code modules to be the same for each module, the function must be exported from each code module.
- If your inline function contains a static object and you expect there to be only one copy of that object, your symbol for that static object must be visible.

Visibility of Inline Functions

You might think that the visibility of inline functions is not an issue, but it is. Inline functions are normally expanded at the call site, and thus never emitted as symbols in the object file at all. In a number of cases, however, the compiler may emit the body of the function, and therefore generate a symbol for it, for some very good reasons. In the most common case, the compiler may decide not to respect the inline optimization if all optimizations are disabled. In more rare cases, the function may be too big to inline or the address of the function might be used elsewhere and thus require a symbol.

Although you can apply the visibility attribute (see “[Visibility Attributes](#)” (page 18)) to inline functions in C++ just as you can any other symbol, it is usually better to hide all inline functions. Some complex issues arise when you export inline functions from dynamic shared libraries. Because there are several variables involved in the compiler’s decision to emit a function or inline it, you may run into errors when building clients for different builds of your shared library.

It is also important to remember that there are subtle differences between the inline function semantics for C and C++. In C programs, only one source file may provide an out-of-line definition for an inline function. This means that C programmers have precise control over where out-of-line copies reside. So for a C-based dynamic shared library, it is possible to export only one copy of an inline function. For C++, the definition of an inline function must be included in every translation unit that uses the function. So, if the compiler does emit an out-of-line copy, there can potentially be several copies of the function residing in different translation units.

In the end, if you want to hide all inline functions (but not necessarily all of your other code), you can use the `-fvisibility-inlines-hidden` flag when compiling your code. If you are already passing the `-fvisibility=hidden` flag to the compiler, use of the `-fvisibility-inlines-hidden` flag is unnecessary.

Symbol Visibility and Objective-C

Objective-C is a strict superset of C, and Objective-C++ is a strict superset of C++. This means that all of the discussion regarding symbol visibility in C and C++ applies to Objective-C and Objective-C++ too. You can use the compiler flags, visibility attributes, and the visibility pragma to hide C and C++ code in your Objective-C code files.

In a 32-bit Mac OS X project, these visibility controls apply only to the C or C++ subset of your code. They do not apply to Objective-C classes and methods. Objective-C class and message names are bound by the Objective-C runtime, not by the linker, so the notion of visibility does not apply to them. There is no mechanism for hiding an Objective-C class or method defined in a dynamic library from the clients of that library.

When building for x86_64 Mac OS X or for iPhone OS, symbol visibility *does* affect objective-C classes. Hiding a class is not a security panacea—enterprising developers can access any class with objective-C runtime calls—but if you directly reference a class whose visibility is hidden in a library you link to, you will get a linker error. This means that if a given class is intended to be usable outside the library or executable it's defined in, you need to ensure proper symbol visibility.

Document Revision History

This table describes the changes to *C++ Runtime Environment Programming Guide*.

Date	Notes
2009-10-09	Added new information about symbol visibility in Objective-C.
2009-01-22	Added information about checking the version of the C++ runtime library.
2006-06-28	Updated caveats for using the static runtime.
2006-05-23	Added information related to the libstdc++-static library included with Xcode Tools 2.3; made editorial changes.
	Added information related to the libstdc++-static library included with Xcode Tools 2.3.
	Added list of reasons why you would want to export C++ symbols from your binary.
2005-08-11	Updated advice on how to deploy C++ programs running under Mac OS X v10.3.8 and earlier.
2005-04-29	New document that describes how to create and deploy C++ binaries.

