
GCC Porting Guide

[Tools > Compiling & Debugging](#)



2006-10-03



Apple Inc.
© 2005, 2006 Apple Computer, 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, Carbon, eMac, Mac, Mac OS, Macintosh, Objective-C, Velocity Engine, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

PowerPC and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

UNIX is a registered trademark of The Open Group

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 to GCC Porting Guide 7

Organization of This Document 7
See Also 7

General Guidelines for Using GCC 9

Making Your Transition Easier 9
Auto-Vectorization 9
Creating Universal Binaries From the Command Line 10
Migrating CodeWarrior Projects 10
Working With the C++ Runtime Environment 10

Porting from GCC 3.3 to GCC 4.0 13

Justifying the Migration 13
 Performance 14
 Code Optimization 14
 Better Warnings 14
 Better C++ Conformance 15
 Compiler Stability 15
 Compiler Support 15
Making the Switch to GCC 4.0 16
Step 1: Switch Your Project to Build Using GCC 4.0 16
 Modifying Xcode Targets 16
 Modifying Makefiles 17
 Changing Compiler and Linker Flags 17
 Check Your Preprocessor Usage 18
 Use Framework Include Files 19
Step 2: Compile Your Code in GCC 4.0 19
 Changes for C++ Templates 19
 Changes to lvalue Assignments 20
 Searching for Framework Headers 21
 Conflicting Declarations 22
 Kernel Extensions 23
 Warnings About Incorrect Code 24
 Thread Safety and Static Local Variable Initialization 27
Step 3: Address Link Errors 27
 Examine System Library Usage 27
 Check Any Linked C++ Libraries 27
 Check Your C++ Library Exports 28
Step 4: Validate Your Code 28

- Known Code Changes 29
- Optimization Changes for Inline Functions 29
- Forcibly Inlining Functions 30
- Data Type Size Differences 30
- Step 5: Attack Warnings 30
 - Objective-C and Selectors 30
 - Objective-C Instance Variables 31
 - Things You Should Not Do 31

Document Revision History 33

Tables and Listings

Porting from GCC 3.3 to GCC 4.0 13

Table 1	Changing Xcode targets to use GCC 4.0	17
Table 2	Problematic compiler options in GCC 4.0	18
Table 3	Common name lookup errors in GCC 4.0	20
Listing 1	Unqualified names in an inherited template	20

Introduction to GCC Porting Guide

Mac OS X v10.4 introduced a new version of the GCC compiler: version 4.0. This new compiler provided significant improvements for the compilation of C, C++, Objective-C, and Objective-C++. With these improvements, though, came stricter rules and better conformance to the C and C++ standards. As a result, developers may have encountered errors when compiling code that had previously compiled without error under GCC 3.3.

This document provides advice for how to modify your code in ways that make it more compatible with the latest versions of GCC. Because of its improved support for uniform coding standards, compiling your code under GCC 4.0 or later should make it easier to develop code that compiles successfully on other platforms and with other compilers.

Important: This document offers only guidance and tips on how to move your code to the newest revisions of the GCC compiler and is not the definitive reference for the compiler itself. If you need more information about GCC than is provided in this document, see *GNU C/C++/Objective-C 4.0.1 Compiler User Guide*. For a summary of compiler options, you can also consult the `gcc` man page.

Organization of This Document

This document includes the following articles:

- [“General Guidelines for Using GCC”](#) (page 9) offers general porting advice for developers coming to GCC 4.0 for the first time. It also discusses issues related to universal binaries and CodeWarrior migration.
- [“Porting from GCC 3.3 to GCC 4.0”](#) (page 13) provides specific information on how to migrate your code from GCC 3.3 to GCC 4.0.

See Also

For reference information on the GCC compiler, see the following documents:

- *GNU C/C++/Objective-C 4.0.1 Compiler User Guide*
- `gcc` man page

General Guidelines for Using GCC

The following sections provide guidance for developers who are coming to the GCC compiler (and possibly Xcode) for the first time. These guidelines are intended to help you find the information you need to modify your existing projects to support GCC 4.0 or later.

Making Your Transition Easier

When you compile existing code for the first time with the GCC 4.0 compiler, you may be shocked to see a lot of warnings in code that had previously compiled cleanly. Don't panic. This is not an unusual occurrence. The GCC 4.0 compiler is much more strict about code compliance than its predecessors and many of its peers.

Although the best way to remove warnings is to fix your code, if you are just starting your transition this may seem like a daunting task. Luckily, GCC supports a full range of options to suppress warnings. During the initial stages of your transition, you might want to use these options to hide warnings until you can fix the legitimate errors reported by the compiler. Once your executable builds without errors, though, you should enable warnings again and begin to fix those problems too. Starting with version 4.0, the GCC compiler is much better at warning you of potential problems. You should heed these warnings and fix the potential problems.

For information about the warnings and errors you are most likely to see under GCC 4.0, see ["Step 2: Compile Your Code in GCC 4.0"](#) (page 19). For a list of compiler options you can use to disable warnings, see *GNU C/C++/Objective-C 4.0.1 Compiler User Guide* or the `gcc` man page.

Auto-Vectorization

A feature introduced in GCC 4.0.1 is the ability to automatically generate AltiVec (Velocity Engine) or SSE instructions for some types of scalar code. When auto-vectorization is enabled, the compiler looks for loops that might benefit from the use of vector instructions. Loops that process scalar values sequentially are very inefficient compared to the same loop coded with vectors. Using vectors, a loop can process multiple values simultaneously during a single loop iteration. This reduces the total number of iterations (and thus CPU cycles) needed to process the same amount of data.

If you were already considering rewriting some of your scalar code to use vectors, you should try auto-vectorization first:

- If you are using Xcode 2.2 or later, you enable auto-vectorization by modifying the build settings for your project. (Note that auto-vectorization is generally more appropriate for deployment configurations because it also causes compiler optimizations to be enabled.) Select the desired configuration and enable the "Auto-vectorization" code generation option.

- If you are using GCC directly from the command line, add `-ftree-vectorize` to your compiler options. Auto-vectorization also requires at least the `-O2` or `-Os` optimization level. If you are compiling code for the PowerPC architecture, the target CPU must also be set to G4 or higher (`-mcpu=G4` or `-mcpu=G5`). For Intel-based Macintosh computers, vectorization is available for all supported CPU types.

Note: Auto-vectorization occurs only on code that does not already contain vector instructions. If your software contains hand-tuned AltiVec or SSE code, the auto-vectorization feature does not attempt to optimize your code further.

For information on how to port your AltiVec code to SSE, see *AltiVec/SSE Migration Guide*. For additional resources and support on how to transition your software so that it runs on Intel-based Macintosh computers, see Apple's Developer Transition Resource Center at <http://developer.apple.com/transition/>.

Creating Universal Binaries From the Command Line

Although it is easiest to build universal binaries from Xcode 2.1 or later, you can also build them from the command line using GCC 4.0 or later. To build a version of your binary for Intel-based Macintosh computers, you must set the `-arch` option of the GCC compiler to `i386` and build a separate version of your binary away from your PowerPC object files and binary. Once you have compiled binaries for both architectures, you can use the `lipo` tool to merge them into a single executable file.

For fundamental information on how to create universal binaries, *Universal Binary Programming Guidelines, Second Edition* is required reading. For additional information on developing executables for multiple architectures and multiple versions of Mac OS X, see *Cross-Development Programming Guide*. For reference information about the GCC compiler, including command-line options for different architectures, see the `gcc` man page or *GNU C/C++/Objective-C 4.0.1 Compiler User Guide*. Finally, for information about how to merge two or more binaries into a single Mach-O executable, see the `lipo` man page.

Migrating CodeWarrior Projects

If you are currently using CodeWarrior to build your projects, you must convert your CodeWarrior project to Xcode before you can use GCC 4.0. Xcode makes the process of converting CodeWarrior projects easier by providing automated support for the import process. However, you may still need to make some changes to your project for it to be imported cleanly.

For detailed information on how to migrate your CodeWarrior projects to Xcode, see *Porting CodeWarrior Projects to Xcode*.

Working With the C++ Runtime Environment

For projects containing C++ source code, you need to be aware that the runtime environment for C++ changed between Mac OS X v10.3.8 and v10.3.9. Prior to Mac OS X v10.3.9, the standard C++ library was provided as a static library that you would then link directly into your executables. In Mac OS X v10.3.9 and later, this library was changed to be a dynamic shared library.

New programs built using GCC 4.0 are automatically configured to use the new dynamic shared library version of the C++ runtime. Using the dynamic shared library is advantageous as it generally results in smaller application size and faster application load times. The new library also provides general improvements to the C++ runtime and better compatibility.

If you are migrating a project from an earlier version of GCC to GCC 4.0 or later, you must make sure that you remove any references to `libstdc++.a` from your project. (In Xcode, look for this library in the External Frameworks and Libraries group.) The `libstdc++.a` file is the older, static version of the C++ library. If this file is still in your project, you may encounter link errors due to the presence of two copies of the C++ standard library.

For additional information about changes to the C++ runtime, see *C++ Runtime Environment Programming Guide*.

Porting from GCC 3.3 to GCC 4.0

In Mac OS X 10.4, GCC 4.0 is the default compiler for all new projects. If you are creating new projects on the platform, you should naturally be using GCC 4.0 to compile those projects. However, if you are building existing projects using the GCC 3.3 compiler (the default compiler in Mac OS X 10.3), there are also many reasons to upgrade to GCC 4.0, including the following:

- Better compile times
- Better C++ language conformance
- Smaller C++ binaries
- Faster C++ compiles
- Better optimization machinery
- Better error checking and diagnosis
- GCC 3.3 is not supported in Intel-based Macs

Before you upgrade though, you should understand the changes that have gone into GCC 4.0 and how they might affect your code. In particular, code that compiled cleanly using GCC 3.3 may now generate warnings and errors when compiled using GCC 4.0. This is not intended to discourage you from updating to GCC 4.0, however. Upgrading may help you find subtle bugs in your code and bring your code into better conformance with existing standards.

This document provides information and guidance on how to migrate your code to GCC 4.0. For additional information, particularly regarding changes in C++ support, see the following GCC release notes:

- <http://gcc.gnu.org/gcc-3.4/changes.html>
- <http://gcc.gnu.org/gcc-4.0/changes.html>

Note: Apple's release of GCC 4.0 occurred before the Free Software Foundation (FSF) declared the official release. Apple's GCC 4.0 Release Note contains an excellent summary of the changes in Apple's version of the compiler.

Justifying the Migration

GCC 4.0 represents a significant improvement over GCC 3.3. Upgrading is highly recommended for the majority of developers. Not only is the performance of the new compiler better, but it provides stricter conformance to existing standards, thus ensuring that your code is more correct.

Important: GCC 4.0 uses the dynamic library implementations of libgcc and libstdc++ that were introduced in Mac OS X v10.3.9. If you migrate to GCC 4.0, the binaries you create can run only in Mac OS X 10.3.9 and later. For more information on the impacts these dynamic libraries may have on your code, see *C++ Runtime Environment Programming Guide*.

Performance

GCC 4.0 offers better compile times, especially for C++ code. Compile times for C++ programs have improved by as much as 30%. Compile times for C and Objective-C programs have improved by as much as 5%.

Code Optimization

For optimizing code, GCC 4.0 now uses a model called **static single assignment** or SSA. SSA makes your program faster and more efficient by permitting better and more optimizations. The SSA optimizer does a much better job of finding places where functions can be inlined. In future revisions of GCC, it will be possible to have even better optimizations as developers extend the SSA optimizations.

Better Warnings

Warnings often identify potential bugs in your code. GCC 4.0 performs many more checks (regardless of whether optimizations are enabled and disabled) and reports more warnings because of these checks. Identifying potential problems can help you fix your code in a way that improves overall stability. For example, GCC 4.0 will now warn about potentially ambiguous implicit casts, which could lead to manipulations on an unknown or unexpected object.

GCC 4.0 also promotes some warnings to full-fledged errors. For example, GCC 4.0 now reports an error if you try to use a return value from an Objective-C method whose return type is `void`, as shown in the following example:

```
#import <Foundation/Foundation.h>

@interface Foo : NSObject
{
    int _i;
};

- (void) i;
@end

@implementation Foo
- (void) i
{
    return _i;
}
@end

int main(int argc, char **argv)
{
    Foo *f=[[Foo alloc] init];
```

```
    int value = [f i]; // this is a hard error in GCC 4.0.  
}
```

Another place where GCC 4.0 now reports an error is when you have a statement in C++ where one array value is assigned to another array value, as shown below:

```
#include <string.h>  
  
main(int argc, char **argv)  
{  
    int ARRAY_SIZE = 5;  
    int a[ARRAY_SIZE], b[ARRAY_SIZE];  
  
    b[0] = 1;  
    a = b; // ERROR with GCC 4.0; Warning with GCC 3.3.  
    memcpy(a,b,sizeof(a)); // This is the correct way to copy an array in GCC  
4.0.  
}
```

When the preceding code is compiled, GCC reports an error similar to the following:

```
/tmp/foo.C:9: error: incompatible types in assignment
```

Better C++ Conformance

GCC 4.0 now provides closer conformance to the C and C++ language standards, which are supported by most major compiler vendors. Better language conformance means your code is now more correct. It also means that it is more portable to other platforms. If your code deviates from the standards, the compiler issues warnings to let you know. If you are developing your code for use on multiple platforms, this feature alone should justify the upgrade.

Note: To detect potential incompatibilities between C++ code generated with GCC 3.3 and GCC 4.0, you can pass the `-Wabi` flag to GCC. When this flag is in effect, GCC issues a warning when it generates code that is probably not compatible with the vendor-neutral C++ ABI. For more information, see the `gcc` man page.

Compiler Stability

GCC 4.0 contains numerous bug fixes. If you are a C++ developer using GCC 3.3, you are likely to notice many new fixes in GCC 4.0, thanks in part to improved C++ support and a new C++ parser. If you have ever encountered crashes while parsing complex C++ code, you should definitely upgrade to GCC 4.0.

Compiler Support

GCC 4.0 is now the default compiler for Mac OS X v10.4. This means that current development efforts are focused on making GCC 4.0 a better compiler. Even if you do not need the new features or performance offered by the compiler, upgrading means that you will be using the most up-to-date compiler that incorporates all of the latest bug fixes.

Of course, it is important to note that Apple still supports GCC 3.3 and ships the GCC 3.3 compiler alongside GCC 4.0. However, work on the GCC 3.3 code base will be limited to critical bug fixes.

Making the Switch to GCC 4.0

The process for migrating to GCC 4.0 is relatively straightforward and can be summarized by the following steps:

1. Switch your project to build with GCC 4.0.
2. Compile your project and address any errors.
3. Address link errors.
4. Validate your code to ensure it works as before.
5. Eliminate warnings aggressively.

The sections that follow provide detailed information regarding each of these steps.

Important: Within the following sections, pay particular attention to warnings or important notes like this one. These notes call out misleading symptoms and error messages that might divert your attention from the real problem.

Step 1: Switch Your Project to Build Using GCC 4.0

The first thing you have to do to use GCC 4.0 is configure your project. Even if your projects use the default system compiler (now GCC 4.0), you may need to modify the options you pass to the compiler. The basic steps for upgrading to GCC 4.0 are as follows:

- Modify your Xcode targets or makefiles.
- Check your compiler and linker flags.
- Check your preprocessor usage.


The following sections provide detailed information about each of these steps.

Modifying Xcode Targets

If you are building your project with Xcode, switching to GCC 4.0 involves modifying the settings for each target in your project. How you modify the settings depends on the type of your target.

Table 1 Changing Xcode targets to use GCC 4.0

Target type	Modifications
Native	The icon for native targets looks like a box. For native targets, open the Inspector window for the target. Select the Build Rules tab. If the compiler shown in the System C Rule is not the desired compiler, add a new build rule by clicking the + button. Configure your new build rule to process “C source files” using GCC 4.0 (or the “GCC System Version (4.0)” of the compiler).
Classic (Jambase)	The icon for classic targets looks like a set of concentric circles (bullseye). For classic targets, double-click the target in the project window to open it in an editor. Choose Settings > Simple View > GCC Compiler Settings. Select the GCC 4.0 compiler from the Compiler version popup menu.

 **Warning:** Some developers try to hard-wire their compiler choice in Xcode by assigning values to the CC and CPLUSPLUS settings in the Xcode target expert view. When this happens, the compiler might complain about some options, such as headermaps, that are being used. The problem is that Xcode thinks it's using GCC 4.0 but is in fact still passing GCC 3.3 options to the compiler. To fix this problem, delete the values in CC and CPLUSPLUS. For classic (Jambase) targets, look for suspicious entries referring to the compiler version in the target settings. For native targets, look in the Build tab for these settings.

In some cases, you may want to see which compiler is being used to compile your code and verify the options that are being passed to it. To get this information in Xcode, open the Build Results dialog and show the “build log” pane. (This section is located between the list of errors and the source code pane and is not always visible.) You can also use this pane to view linker error and warning messages. For more information on using this build log, see *Xcode 2.0 User Guide*.

Modifying Makefiles

If your project is built using Makefiles or custom build scripts, look for references to `/usr/bin/gcc-3.3` and change them to `/usr/bin/gcc` or `/usr/bin/gcc-4.0`. By default, invoking `/usr/bin/gcc` launches the default system compiler, which in Mac OS X v10.4 is GCC 4.0.

Changing Compiler and Linker Flags

Some projects may use flags that are supported in GCC 3.3 but were removed in GCC 4.0. You can locate these changes by explicitly checking your compiler options in each Xcode project, or you can simply build your project and wait for error messages regarding illegal options. Once you identify the incorrect flag, go to the settings for the offending target and remove the option. Table 2 lists some other compiler flags that you should think about changing.

Table 2 Problematic compiler options in GCC 4.0

Flag	Description
-Werror	This flag is still supported in GCC 4.0, but because it turns warnings into hard errors, you might want to disable it anyway. GCC 4.0 already generates significantly more warnings than before, so this flag is not as necessary while you are trying to get your code up and running on GCC 4.0. To disable it, disable the "Treat warnings as errors" build setting for your target.
-Wno-precomp	This flag has not been needed since GCC 3.1.
-fcoalesce, -fweak-coalesced, -fcoalesce-templates	There is no need to use these flags because the behavior specified by those flags is now the default. Those flags can safely be removed from any projects that use them.
-fno-coalesce, -fno-weak-coalesced, -fno-coalesce-templates	These flags can be safely removed. The projects that used these flags in earlier compiler versions usually did so as workarounds for compiler bugs that no longer exist. In the very rare cases where there is a real need to disable C++ "vague linkage", use the <code>-fno-weak</code> flag.
--param max-inline-insns, --param min-inline-insns	These flags are no longer used. The values used by the rest of the inlining parameters have changed their meaning between GCC 3.3 and GCC 4.0. Parameters used for GCC 3.3 will cause slower compile times and larger binaries in GCC 4.0. You may want to remove all inlining flags, analyze which portions of your code are slow, and then set the inlining parameters so that the slow functions are inlined.

Check Your Preprocessor Usage

Do a quick check for bad preprocessor usage. If you have conditional code based on which compiler you're using (as in GCC vs. CodeWarrior), don't forget to update any conditional code looking at the `__GNUC__` variable for the compiler version, as shown in the following code listing:

```
if (__GNUC__ == 3) // old

if (__GNUC__ == 3) || (__GNUC__ == 4) // new
```

Searching for `__GNUC__` in your project should locate all of the relevant code.

If you are migrating a project from CodeWarrior, do not wrap any Mac OS X-specific code using the `__MWERKS__` macro. Instead, use the following:

- Use `#ifdef __GNUC__` to wrap any GCC-specific code.
- Use `#ifdef __APPLE_CC__` to wrap any Mac OS X-specific code.

Use Framework Include Files

If you are migrating a Carbon application from CodeWarrior to Mac OS X, you should remove any references to universal header files and replace them with the appropriate framework header files for Carbon. Framework includes are the standard way to include system header files. For example, to include the header files for the Carbon framework, you would use the following `#include` statement:

```
#include <Carbon/Carbon.h>
```

If you encounter a large number of errors when including the Carbon framework in this manner, you can use the flat Carbon header files instead. The intent of the flat Carbon headers is to help ease the conversion away from universal headers. To include the flat Carbon headers, add `/Developer/Headers/FlatCarbon/` to the include path of your project.

Step 2: Compile Your Code in GCC 4.0

Once you start compiling your code, the most common causes of problems are likely to be the following:

- Invalid compiler and linker flags
- Changes to C++ template code
- Restrictions on what can be on the left hand side of assignment statements (lvalues)
- Problems finding header files
- Conflicting declarations (static vs. extern)
- Changes to kernel extension ("kext") source code
- Warnings about incorrect code

For more information about invalid compiler and linker flags, see [“Changing Compiler and Linker Flags”](#) (page 17). The remaining problems are explained in the sections that follow. Each section explains the class of the problem and discusses issues you may notice when running the resulting binary.

Changes for C++ Templates

The most common problem C++ programmers are likely to encounter revolve around changes to support C++ standards compliance. GCC 3.4 (which was shipped by the FSF but not by Apple) started to warn developers about incorrect C++ usage patterns. In GCC 4.0, those warnings are now errors.

One of the more common errors you’re likely to encounter is name lookup problems of the form “x isn’t defined in this scope.” You can find out more about this problem by searching for “name lookup” in the GCC 4.0 documentation. Four of the most common name lookup errors are as follows:

Table 3 Common name lookup errors in GCC 4.0

Error	Description
Dependent names in a template class	Names in function templates are either dependent or non-dependent. Dependent names depend lexically on a template parameter and are looked up when the template is instantiated. Non-dependent names are looked up when the template is parsed, which occurs before it is instantiated.
Unqualified names in a template	Names that are specific to a template parameter class must be qualified, either by explicitly saying that they are from the template (<code>Foo<T>::mArray</code>) or by using a <code>this</code> pointer to indicate they come from the template instance. If you explicitly state that they are from the template, the name is looked up at instantiation time; otherwise, if you use a <code>this</code> pointer, the name is looked up at template definition time.
Unqualified names in a template superclass	Names defined in a template superclass of the current template must be qualified as being from the superclass. Alternatively, you can also qualify those names by preceding them with <code>this-></code> .
Unqualified names in an inherited template class	If a template class inherits from another template class, members of the inherited template class must be qualified. For an example, see Listing 1.

Listing 1 shows an example of an unqualified name belonging to an inherited template class.

Listing 1 Unqualified names in an inherited template

```
template <typename T> struct Base
{
    int local_var;
    void f();
};

template <typename T> struct Derived : public Base<T>
{
    void g()
    {
        local_var++;    // ERROR: Name not found.
        f();           // ERROR: Name not found.
        this->local_var++;    // OK
        this->f();      // OK
        Derived::f();  // OK
        Base<T>::f();  // OK
    }
};
```

Changes to lvalue Assignments

The GCC compiler was previously extended to permit non-trivial expressions on the left hand side of an assignment (the **lvalue** of the assignment). This feature was removed from GCC 4.0 to better match the C and C++ standards and because it was not always clear what the programmer intended in those situations.

Currently, Apple's version of GCC 4.0 compiler only warns about some cases that the FSF's GCC 4.0 compiler would consider an error. Those cases that remain will be eliminated in a future version of Apple's compiler. What follows are some examples of code that may now generate errors or warnings.

In the following case, GCC 4.0 no longer permits the following conditional lvalue assignments.

```
(condition ? lvalue1 : lvalue2) = expr; // Warning: target of assignment not
really an lvalue (will be a hard error in future)
```

To avoid this problem, rewrite the expression so that the left-hand side contains only a variable.

Another case that generates an error involves taking the address of an lvalue variable, as shown here.

```
&foo = 1; // ERROR: invalid lvalue in assignment
```

Instead of this expression, create a temporary variable for the address of `foo`, and assign a value to that.

In the last case, the problem is caused by dereferencing first and then casting to the target type. Instead, cast the pointer to an appropriate pointer type and then dereference the value.

```
(u_int32_t)*(vcp->vc_outtok) = sp->sv_caps; // Warning: target of assignment
not really an lvalue (hard error in future)
```

```
*(u_int32_t*)(vcp->vc_outtok) = sp->sv_caps; // OK
```

Searching for Framework Headers

Several issues in framework and header searching changed between GCC 3.3 and GCC 4.0.

In GCC 3.3, the compiler would incorrectly search all framework paths when searching for a header in a framework. This is incorrect behavior because there are cases where headers could be retrieved from different installed versions of the framework. For example, some headers might be retrieved from a copy of the framework you are working on while others are retrieved from an officially installed version of the framework.

GCC 4.0 now uses a “first match wins” model for frameworks. With this model, the first time you include a framework header, the compiler makes a note of the location of that framework. Subsequent attempts to retrieve headers from the same framework automatically go to the same location.

Important: If the compiler reports that it cannot find a header that you know to exist in the framework, it could be due to this mismatched framework header problem. To fix this problem, reorder the framework paths so that the complete version of the framework is searched first.

In GCC 3.3, the `-I` and `-F` flags were treated interchangeably. This is incorrect behavior. In GCC 4.0, you must use the `-F` flag to get the proper search semantics for your framework directory.

Important: If the compiler reports that framework headers cannot be found, you should also check to make sure you are including the framework using the `-F` flag.

```
gcc -I/System/Library/MyFrameworks foo.m // worked in gcc-3.3, doesn't work in
gcc-4.0.
gcc -F/System/Library/MyFrameworks foo.m // OK
```

Conflicting Declarations

The following sections list some of the conflicting declarations you may encounter:

Static versus Extern

GCC 4.0 now warns you if your project defines the same variable as `extern` and `static`. In the past, the compiler used to respect whichever declaration it encountered last.

This problem occurs when you create a local static variable in one file and then declare a variable with the same name as `extern` in a different file. Don't do this. Instead, change the name of the static variable so that it doesn't match the global variable name.

```
// foo.h:
extern int _defaultWidth; // used by lots of compilation units

// foo.c:
static int _defaultWidth; // local copy -- ERROR: static declaration of
'_defaultWidth' follows non-static declaration
```

Forward Declaration of Static Functions

GCC 4.0 does not allow you to place a forward definition of a static function inside the body of another function. To correct this situation, move the forward declaration outside of the function (to the file namespace level).

```
int foo(int i) {
    static int bar(int j); // ERROR: invalid storage class for function 'bar
    bar(i+5);
}

static int bar(int j); // OK
int foo(int i) {
    bar(i+5);
}
```

C++ new

GCC 4.0 now enforces the rule that when allocating an object array, the class name in the new statement cannot be surrounded by parenthesis.

```
map = new (IOMemoryMap*)[WindowCount]; // ERROR: array bound forbidden after
parenthesized type-id
// note: try removing the parentheses around the type-id

map = new IOMemoryMap *[WindowCount]; // OK
```

Implicit Arguments to C++ Member Functions

The same implicit arguments (default parameters) cannot be specified in both the declaration and definition of a member function. A member function definition declared outside the class can give additional default arguments, but not the same ones as shown in the following example:

```

class A
{
    int i;
    int foo(int i, int j = 99);
};

int A::foo(int i, int j = 99) // ERROR: default argument given for parameter 1
of 'int A::foo(int)'
    // ERROR: after previous specification in 'int A::foo(int)'
{
    ...
}

int A::foo(int i=10, int j) { // OK, but visible only in this compilation unit
    ...
}

int A::foo(int i, int j) { // OK
    ...
}

```

Conflicting Declarations in Namespaces

When declaring a namespace, do not include the name of the namespace in the declaration:

```

namespace mynames
{
    void mynames::func(); // ERROR! Explicit qualification in declaration
    void func(); // OK.
};

```

Kernel Extensions

GCC 4.0 can compile kernel extensions (kexts) that should work with current and previous versions of Mac OS X (subject to all the conditions that GCC 3.3-compiled kexts followed). Regardless, you must make a few changes to your kext to get it to compile.

Kernel extensions cannot have statically initialized `auto` (function-local) variables that call C++ constructors. This is because the kernel does not provide locking routines around the initialization of `auto` variables. (Such locking routines could trigger nasty side effects in the kernel.) Although your code should compile fine, it will fail to load because the symbols for the missing locking code (`__cxa_guard_abort`, `__cxa_guard_acquire`, and `__cxa_guard_release`) are undefined. To correct the problem, you must move the local variables to the top level:

```

static A* globalArrayOfA = new A[10];

int A::foo(int i)
{
    static A* arrayOfA = new A[10];
    return arrayOfA[i].value(); // KEXT LOAD ERROR: __cxa_guard_abort undefined
    return globalArrayOfA[i].value(); // OK
}

```

Many kexts try to cast member function pointers to make them appear as if they are simply function pointers. This is done so that the member function can be registered as an interrupt handler. GCC 4.0 disallows such casts. Instead, you must use the `OSMemberCastFunction` macro to perform the cast correctly. This macro is available in Mac OS X v10.4 and should work identically with GCC 3.3 or GCC 4.0, and with the same behavior as the original code when compiled with GCC 3.3.

```
int A::bar(int i)
{
    handler = (IOInterruptAction) &A::interruptHandler; // ERROR: converting
from `int (A::*)(int)' to `int (*)(int)' in a kext. Use OSMemberFunctionCast()
instead.

    return (*handler)(5);
}

int B::bar(int i)
{
    handler = OSMemberFunctionCast(IOInterruptAction, this, &B::interruptHandler);
// OK
    return (*handler)(5);
}
```

Warnings About Incorrect Code

The following sections explain some of the other types of warnings and errors you may encounter during compilation.

Declaring Abstract Virtual Functions

Don't use NULL to define a virtual function as unimplemented. Instead, use the value 0, as shown in the following example:

```
class C
{
    virtual int PrepareCFBundle() = NULL; // ERROR: invalid initializer for
virtual method

    virtual int PrepareCFBundle() = 0; // OK
};
```

Labels

The ISO C specification says that labels must be followed by a statement. Thus, you cannot have a label at the end of a block:

```
foo: }
```

To fix this problem, simply add a semicolon after the label to create an empty statement:

```
foo: ; }
```


Static Data in Inline Functions

GCC 4.0 reports an error if you define a static array inside an inline function. The compiler cannot determine if it should use a single array or duplicate the array as it inlines the function. To fix this, move the array out of the function.

The `offsetof` Function

GCC 4.0 now requires that arguments to `offsetof` must be constant:

```
bufferSizeNeeded = offsetof(ATSUGlyphInfoArray, glyphs[numGlyphs]); // ERROR:

// Instead, do this.
bufferSizeNeeded = (char*)&(((ATSUGlyphInfoArray*)0)->glyphs[numGlyphs]) -
(char*)0;
```

Bitfields and the `typeof` Operator

Because of a change to the C compiler, attempting to get the type of a bitfield value generates an error in GCC 4.0. GCC 3.3 used to allow this operation for C code and it is still supported in C++ code in GCC 4.0.

```
// The following macro won't work in GCC 4.0 because the type of
// value is considered to be a bitfield that can't be used in typeof.
// In GCC 3.3, it would have been treated as an int.

#define STRUCT_VALUE(x) ((x->isSet ? x->value : (typeof(x->value))0))

struct a
{
    int isSet: 1;
    int value:15;
};

int foo(struct a *instance)
{
    return STRUCT_VALUE(instance);
}
```

In some situations, you can correct this problem in your C code. Specifically, if the bitfield value is a 2-bit integer, cast the value being used in the `typeof` statement to `int`. This should allow the code to compile.

Initializing Class Instance Variables

With GCC 4.0 and C++, static class instance variables can be initialized inside the class declaration only if the type is an integer or `enum` and only if the assigned value is a constant. If you need to initialize other types or if you need to initialize the value using an expression, you cannot do it from the class declaration. Instead, you must perform the assignment from the code in your class definition file.

Assigning Arrays

You cannot assign one array variable to another and expect the compiler to copy the array members for you. When copying arrays, you must use `bcopy` to explicitly copy the bytes of the array. GCC 3.3 issued a warning for this type of behavior, but in GCC 4.0, this now results in a hard error.

Sequence Points

You cannot modify variables as part of passing them to a function, as shown in the following example:

```
void myFunc()
{
    int i = 1;
    myOtherFunc(i, i++);    // Warning! Operation on 'i' may be undefined.
}
```

Instead, simply modify the variable outside of the function parameter list, as shown here:

```
void myFunc()
{
    int i = 1;
    myOtherFunc(i, i);
    i++;
}
```

Defining System Macros Which Change Header File Processing

Some open source programs have failed to build because of supposedly missing files. The problem is caused by the project defining an implementation-reserved name, such as `__FreeBSD__`, either in a source code file or in a compiler argument. Implementation-reserved names begin with two underscore characters or one underscore followed by a capital letter. These names often control which header files are used by the compiler. Defining the `__FreeBSD__` name, and not setting a value, can cause the BSD-derived header files to assume you are running on an earlier version of BSD than you actually are.

Important: The definition of implementation-reserved macros in your project should always be viewed suspiciously because they can alter the way in which the system parses header files. One of the symptoms of defining the `__FreeBSD__` name is that the compiler reports that the header file `machine/ansi.h` cannot be found.

In the case of the `__FreeBSD__` name, the header files assume you're running on a version of BSD prior to version 5. Versions of BSD prior to version 5 included the `machine/ansi.h` header file. Mac OS X is derived from BSD version 5, which does not include the header file. Thus, the compiler reports an error when it cannot find the file.

General Correctness

GCC 4.0 now generates errors in many cases where its predecessor (GCC 3.3) did not. Most of these errors should be obvious once you examine the code. Examples of these errors include the following:

- Accessing a private instance variable in C++ and Objective C
- An empty return value in a function that is expected to return a value
- Returning a value from a function whose return type is `void`
- Doing anything with the return value from a function whose return type is `void`
- Arbitrary casts. The compiler does not allow you to implicitly cast a `void` pointer to some other type. Instead, you must provide an explicit cast.

Thread Safety and Static Local Variable Initialization

To maximize thread safety, GCC 4.0 automatically adds locks around any code that initializes local static variables in C++. If you do not need this protection and want to reduce your code size slightly, you can disable the locking behavior by passing the `-fno-threadsafe-statics` option to the compiler.

Step 3: Address Link Errors

Once you sort through and solve any major compilation errors, you can begin to examine any linking issues. C++ developers in particular may discover that they have missing or multiply-defined symbols.

Examine System Library Usage

The first step to resolving link issues is to examine your system library usage. In Mac OS X v10.3.9 and later, both `libgcc` (the C support library) and `libstdc++` (the Standard C++ library) are implemented as dynamic libraries. These libraries are linked into your program automatically by the GCC 4.0 compiler (rather than `ld`). If you encounter multiply defined symbols from these libraries, you may be including the static versions of these libraries on your link line.

Check to see if your project links against the `libgcc.a`, `libstdc++.a`, or `libgcc_dynamic.a` static libraries. If it does, you should remove those libraries from your link commands.

If you compile your C++ application outside of Xcode, initiate the link phase using `g++`, and not `ld`, to ensure the inclusion of the correct libraries.

Check Any Linked C++ Libraries

Once you have resolved any link issues with the system libraries, look at the other libraries on your link line. If you link to any libraries that contain a C++ interface and were created using GCC 3.3, you must remove them from your link line.

The C++ application binary interface (ABI) changed between GCC 3.3 and GCC 4.0. Any application or library linking against a library that exports C++ functions must be built with the same compiler. Such problems are usually obvious when you check the list of undefined symbols. If the linker complains that any symbols starting with `_ZTI` are undefined, you are probably compiling a program using a mix of GCC 3.3 and GCC 4.0 binaries.

Note: The `_ZTI` prefix indicates the mangled type info for a specific C++ class.

In your own libraries, you can use the `-Wabi` flag to tell the GCC compiler to issue warnings when it generates code that is not compatible with the vendor-neutral C++ ABI. For more information about this flag, see the `gcc` man page.

Check Your C++ Library Exports

If you are building a C++ library and limit the symbols exported from your library, you should verify that your library exports the symbols you expect.

By default, GCC 4.0 marks most symbols as private to prevent them from being exported. This is a change from the GCC 3.3 behavior but has significant advantages, particularly for C++ developers. Setting the default symbol visibility to private reduces the size of the exported symbol tables. If your code makes extensive use of templates, this size reduction could be huge.

Important: If your project is a dynamic library and uses an export list, prepare to change the list to make sure the same symbols are exported from your library. Use the `nm` tool to check the list of symbols exported from both your GCC 3.3 binary and GCC 4.0 binary. If you export C++ interfaces, remember to export class info (`_ZTI*` symbols) for your exported classes.

If you don't want the space improvements triggered by making most symbols private, you can tell GCC 4.0 to export most symbols in the GCC 3.3 manner. To do so from Xcode, turn off the "symbols private by default" setting for your target. To do so from the GCC command line option, specify the `-fvisibility=default` option on the command line.

Important: In GCC 4.0 and Xcode, projects are built with all symbols marked `private extern` by default. To export symbols, you either need to turn off the global setting for hiding symbols or mark only those symbols that you want to be exported. Otherwise, if you use `nmedit` to strip unwanted symbols, the tool strips all `private extern` symbols, which may not be what you intended.

For more information regarding C++ symbol visibility and runtime behavior, see *C++ Runtime Environment Programming Guide*.

Step 4: Validate Your Code

When you reach the stage where your application builds successfully, you should start testing to make sure everything still works as intended. Run the code through your test suites to verify that the compiler did not change any of your application's behavior. Changing compilers can introduce new problems, so testing is an important step in verifying that the change went smoothly.

In addition to running your test suites, you should also check the size of your resulting binary file. With GCC 4.0, the compiler can do optimizations in places that it could not previously. Using the same compiler settings, you might find the resulting binary is quite different than ones previously built using GCC 3.3. New functions may be inlined, which could significantly increase the size of your binary. (For more information on this effect, see ["Optimization Changes for Inline Functions"](#) (page 29).)

If you are building a dynamic shared library, verify that the list of symbols exported by your library has not changed. For more information on fixing export problems, see ["Check Your C++ Library Exports"](#) (page 28).

Known Code Changes

The code optimizer in GCC 4.0 is very different from the ones in previous versions of GCC. If you discover incorrect behavior in your recompiled code, or if you encounter crashes, check for the following known issues:

- The optimizer in GCC 4.0 may change the order of computations for some mathematical operations (such as addition or multiplication).
- Casting a member-function pointer to a regular function pointer now works differently in GCC 4.0. Prior to GCC 4.0, such casts were resolved at runtime, which meant that the address of the function was based on the actual type of the object at the call site. For example, suppose you request the address of `A::MyFunction` from another method of `A`. If the actual type of the object at runtime is `B` (a subclass of `A`) and `B` defines its own version of `MyFunction`, you would actually get the address of `B::MyFunction`. In GCC 4.0, the address of these functions is now calculated at compile time. So now, if you ask for `&A::MyFunction`, you will always get the address of `A::MyFunction`.

For arithmetic operations, the C and C++ standards state that subexpression evaluation order is undefined; the compiler is free to evaluate the arguments in addition or multiplication expressions in any order it wants. There are exceptions to this rule, however. These exceptions include the following types of subexpressions, which are all evaluated from left to right:

- `&&` operators
- `||` operators
- `?:` operators
- `,` (comma) operators

If one of these subexpressions has side effects, you might notice different behavior in GCC 4.0, depending on how the optimizer has reordered the code. You can use the `-Wsequence-point` flag to find any code that may have undefined semantics.

Optimization Changes for Inline Functions

The inlining algorithms in GCC 4.0 are much better at deciding when to inline functions rather than call them. The side effects of this optimization, however, are larger binaries and faster code. In many cases, you should not see any significant differences unless you have been overriding the default parameters that control inlining in your project. If you currently use the inlining parameters `-finline-limit=xxx` or `--param`, do not continue to use the currently assigned values for those parameters. Instead, profile your code again and see what parameters offer the best balance between code size and speed for your application.

Internal testing has shown that projects using the `-finline-limit` option under GCC 3.3 would see code size increases of up to 30% and compile time increases of up to 50% under GCC 4.0. For `-finline-limit`, the default value is usually 600. If you use the `-Os` flag to optimize for size, the default value drops to 10. If you manually assign a limit of 200 or more to the `-finline-limit` flag, you are liable to see larger binaries than you did using the same settings under GCC 3.3.

Forcibly Inlining Functions

If you want to force a routine to be inlined all the time, do not assume the compiler will do it for you. Tell the compiler you explicitly want the routine inlined by adding the `always_inline` attribute to it, as shown in the following example:

```
void MyInlineFunction(int c) __attribute__((always_inline))
```

Data Type Size Differences

For many basic data types, such as `bool` and `int` compilers are at liberty to define the size of those types as they see fit. If you have code that expects these types to be a specific size, your code may not behave as expected. The transition to Macs that use Intel processors and differences between 32-bit and 64-bit binaries make it very important that you do not assume the size of some data types.

For string data, you should also not make assumptions about the size of character data types such as `wchar_t`. On Mac OS X, this type is typically 2 bytes wide, but on most UNIX operating systems it is 4 bytes. If you really need two-byte character data, manipulate your strings using CFString objects and examine the resulting characters using the `UniChar` data type. If your program retrieves strings from external sources, you can convert them using the `CFStringCreateFromExternalRepresentation` function. (You can also use the `iconv` function on UNIX; see `man 3 iconv` for details.)

Step 5: Attack Warnings

GCC 4.0 is much better about warning you of potentially incorrect code. Therefore, it is important that you vigorously pursue and eliminate warnings in your code. The newly reported warnings have already uncovered several potential bugs lurking in Mac OS X sources, and you are strongly advised to eliminate them from your own sources as well.

Once your code builds successfully under GCC 4.0, you should start examining the warnings produced by the compiler. Some warnings may reflect stylistic differences but they may also identify places where the compiler cannot infer enough information from the code to know what was originally intended. You should examine each of them to determine whether the warning is innocuous or identifies a potential bug. Removing the potential problems not only eliminates the warnings but should also add stability to your code.

The following sections describe some of the more common warnings you can expect to see in projects compiled with GCC 4.0.

Objective-C and Selectors

When a message is sent to a receiver of type `id`, the compiler looks at *all* matching selectors and warns about inconsistencies in argument types. GCC 4.0 is better at this than GCC 3.3 was and may reveal inconsistencies that were previously hidden. Although currently disabled, you can enable these warnings using the `-Wstrict-selector-match` flag. In the future, the compiler will warn about these inconsistencies by default.

Here are 3 examples of this warning that occurred in the Sketch example application:

```
NSNotification.h:40: warning: using `-(void)postNotificationName:(NSString *)aName object:(id)anObject' NSDistributedNotificationCenter.h:59: warning: also found `-(void)postNotificationName:(NSString *)aName object:(NSString *)anObject'
```

```
NSNotification.h:37: warning: using `-(void)addObserver:(id)observer selector:(SEL)aSelector name:(NSString *)aName object:(id)anObject' NSDistributedNotificationCenter.h:57: warning: also found `-(void)addObserver:(id)observer selector:(SEL)aSelector name:(NSString *)aName object:(NSString *)anObject'
```

```
NSView.h:115: warning: using `-(NSWindow *)window' NSAlert.h:125: warning: also found `-(id>window'
```

To fix these, use explicit casts or change the method definitions so that calls and definitions match.

Objective-C Instance Variables

In Objective C, instance variables are labeled as `@protected` by default. GCC 4.0 now explicitly warns about this, and future versions of the compiler will treat this as an error. To correct this problem, you should explicitly identify the scope of instance variables in your Objective-C interfaces.

Things You Should Not Do

There are cases where combinations of particularly ugly or strange casts can generate the following warning:

```
Pictures.c: In function 'StdOpcode':  
Pictures.c:1132: warning: function called through a non-compatible type  
Pictures.c:1132: note: if this code is reached, the program will abort
```

In these cases, the compiler really does insert an abort instruction rather than try to generate what it thinks is an incorrect function call. If you see this warning, you should absolutely fix it by removing or changing the cast so that the compiler can call the correct function.

Document Revision History

This table describes the changes to *GCC Porting Guide*.

Date	Notes
2006-10-03	Fixed the code that demonstrates the use of implicit arguments to C++ member functions.
2006-04-04	Improved the array assignment code example. Updated sequence point exceptions and documented additional compiler warnings that may aid in finding nonconforming code.
2005-11-09	Added high-level guidance for developers coming from CodeWarrior or making the transition to Intel-based Macs. Added more information about warnings and errors generated by GCC 4.0.
	Added information about the auto-vectorization feature in GCC 4.0.1.
	Changed title from "Porting to GCC 4.0 Release Notes".
2005-08-11	Added information about GCC 4.0 support for thread-safe local static variables.
2005-06-04	New document providing advice on how to port from GCC 3.3 to GCC 4.0.

