C++ portability guide

version 0.7
by David Williams
27 March 1998

Updated and maintained by Scott Collins and Christopher Blizzard

What follows is a set of rules, guidelines, and tips that we have found to be useful in making C++ code portable across many machines and compilers.

This information is the result of porting large amounts of code across about 25 different machines, and at least a dozen different C++ compilers. Some of these things will frustrate you and make you want to throw your hands up and say, ``well, that's just a stupid compiler if it doesn't do <*insert favorite C++ feature*>." But this is the reality of portable code. If you play by the rules, your code will seamlessly work on all of the Mozilla platforms and will be easy to port to newer machines.

We will endeavor to keep the information up to date (for example, sometimes a new compiler revision will lift a restriction). If you have updates on any of these tips, more information, more ideas, please forward them to Christopher Blizzard or Scott Collins.

If you find code in Mozilla that violates any of these rules, please report it as a bug. You can use bonsai to find the author.

C++ portability rules.

C++ portability rules.

1. **Don't use C++ templates.** (*)

   Don't use the C++ template feature. This feature is still not implemented by all compilers, and even when it is implemented, there is great variation. Most of the interesting things that you would want to do with templates (type safe container classes, etc.) can be implemented with macros and casting, even though you do lose the type safety (pity). Often times subclassing can easily achieve the same result.

   (*) There is a an exception to this rule: nsCOMPtr.

   However, this does not mean "Open Season" for template code. The "Don't use C++ templates" rule still applies. nsCOMPtr is allowed because the authors spent a lot of time making sure their use of templates does not break poor compilers.

   It is very likely that other "simple" template code will break some poor compilers which we need to support.

2. **Don't use static constructors.**

   Non-portable example:

```
FooBarClass static_object(87, 92);

void
bar()
{
  if (static_object.count > 15) {
     ...
```

```
    }
  }
```

Static constructors don't work reliably either. A static initialized object is an object which is instanciated at startup time (just before `main()` is called). Usually there are two components to these objects. First there is the data segment which is static data loaded into the global data segment of the program. The second part is a initializer function that is called by the loader before `main()` is called. We've found that many compilers do not reliably implement the initializer function. So you get the object data, but it is never initialized. One workaround for this limitation is to write a wrapper function that creates a single instance of an object, and replace all references to the static initialized object with a call to the wrapper function:

Portable example:

```
  static FooBarClass* static_object;

  FooBarClass*
  getStaticObject()
  {
    if (!static_object)
      static_object =
        new FooBarClass(87, 92);
    return static_object;
  }

  void
  bar()
  {
    if (getStaticObject()->count > 15) {
      ...
    }
  }
```

3. **Don't use exceptions.**

   Exceptions are another C++ feature which is not very widely implemented, and as such, their use is not portable C++ code. Don't use them. Unfortunately, there is no good workaround that produces similar functionality.

   One exception to this rule (don't say it) is that it's probably ok, and may be necessary to use exceptions in some machine specific code. If you do use exceptions in machine specific code you must catch all exceptions there because you can't throw the exception across XP (cross platform) code.

4. **Don't use Run-time Type Information.**

   Run-time type information (RTTI) is a relatively new C++ feature, and not supported in many compilers. Don't use it.

   If you need runtime typing, you can achieve a similar result by adding a `classOf()` virtual member function to the base class of your hierarchy and overriding that member function in each subclass. If `classOf()` returns a unique value for each class in the hierarchy, you'll be able to do type comparisons at runtime.

5. **Don't use namespace facility.**

   Support of namespaces (through the `namespace` and `using` keywords) is a relatively new C++ feature, and not supported in many compilers. Don't use it.

6. **`main()` must be in a C++ file.**

   The first C++ compiler, Cfront, was in fact a very fancy preprocessor for a C compiler. Cfront reads the C++ code, and generates C code that would do the same thing. C++ startup is slightly different to C startup (for example static constructor functions must be called for C++), and Cfront implements this special startup by noticing the function called "`main()`", converting it to something else (like "`__cpp__main()`"), adding another `main()` that does the special C++ startup things and then calls the original function. Of course for all this to work, Cfront needs to *see* the `main()` function, hence `main()` must be in a C++ file. Most compilers lifted this restriction years ago, and deal with the C++ special initialization duties as a linker issue. But there are a few commercial compilers shipping that are still based on Cfront: HP, and SCO, are examples.

So the workaround is quite simple. Make sure that `main()` is in a C++ file. On the Unix version of Mozilla, we did this by adding a new C++ file which has only a few lines of code, and calls the main `main()` which is actually in a C file.

7. **Use the common denominator between members of a C/C++ compiler family.**

   For many of the compiler families we use, the implementation of the C and C++ compilers are completely different, sometimes this means that there are things you can do in the C language, that you cannot do in the C++ language on the same machine. One example is the 'long long' type. On some systems (IBM's compiler used to be one, but I think it's better now), the C compiler supports long long, while the C++ compiler does not. This can make porting a pain, as often times these types are in header files shared between C and C++ files. The only thing you can do is to go with the common denominator that both compilers support. In the special case of long long, we developed a set of macros for supporting 64 bit integers when the long long type is not available. We have to use these macros if either the C or the C++ compiler does not support the special 64 bit type.

8. **Don't put C++ comments in C code.**

   The quickest way to raise the blood pressure of a Netscape Unix engineer is to put C++ comments (`//` comments) into C files. Yes, this might work on your Microsoft Visual C compiler, but it's wrong, and is not supported by the vast majority of C compilers in the world. **Just do not go there.**

   Many header files will be included by C files and included by C++ files. We think it's a good idea to apply this same rule to those headers. Don't put C++ comments in header files included in C files. You might argue that you could use C++ style comments inside `#ifdef __cplusplus` blocks, but we are not convinced that is always going to work (some compilers have weird interactions between comment stripping and pre-processing), and it hardly seems worth the effort. Just stick to C style `/**/` comments for any header file that is ever likely to be included by a C file.

9. **Don't put carriage returns in XP code.**

   While this is not specific to C++, we have seen this as more of an issue with C++ compilers, see Use the common denominator between members of a C/C++ compiler family.

   On unix systems, the standard end of line character is new line (`'\n'`). The standard on many PC editors is carriage return (`'\r'`). The PC compilers seem to be happy either way, but some Unix compilers just choke when they see a carriage return (they do not recognize the character as white space). So, we have a rule that you cannot check in carriage returns into any cross platform code. This rule is not enforced on the Windows front end code, as that code is only ever compiled on a PC. The Mac compilers seem to be happy either way, but the same rule applies as for the PC - no carriage returns in cross platform code.

10. **Put a new line at end-of-file.**

    Not having a new-line char at end-of-file breaks some compilers (Solaris).

11. **Don't put extra top-level semi-colons in code.**

    Non-portable example:

    ```
    int
    A::foo()
    {
    };
    ```

    This is another problem that seems to show up more on C++ than C code. This is problem really a bit of a drag. That extra little semi-colon at the end of the function is ignored by most compilers, but it makes some compilers very unhappy (IBM's AIX compiler doesn't like extra top-level semi-colons). Don't do it.

    Portable example:

    ```
    int
    A::foo()
    {
    }
    ```

12. **C++ filename extension is `.cpp`.**

    This one is another plain annoying problem. What's the name of a C++ file? `file.cpp`, `file.cc`, `file.C`, `file.cxx`, `file.c++`, `file.C++`? Most compilers could care less, but some are very particular. We have not been able to find one file extension which we can use on all the platforms we have ported Mozilla code to. For no great reason, we've settled on `file.cpp`, probably because the first C++ code in Mozilla code was checked in with that extension. Well, it's done. The extension we use is `.cpp`. This extension seems to make most compilers happy, but there are some which do not like it. On those systems we have to create a wrapper for the compiler (see `STRICT_CPLUSPLUS_SUFFIX` in `ns/config/rules.mk` and `ns/build/*`), which actually copies the `file.cpp` file to another file with the correct extension, compiles the new file, then deletes it. If in porting to a new system, you have to do something like this, make sure you use the `#line` directive so that the compiler generates debug information relative to the original `.cpp` file.

13. **Don't mix varargs and inlines.**

    Non-portable example:

    ```
    class FooBar {
      void va_inline(char* p, ...) {
        // something
      }
    };
    ```

    The subject says it all, varargs and inline functions do not seem to mix very well. If you must use varargs (which can cause portability problems on their own), then ensure that the vararg member function is a non-inline function.

    Portable example:

    ```
    // foobar.h
    class FooBar {
        void
          va_non_inline(char* p, ...);
    };

    // foobar.cpp
    void
    FooBar::va_non_inline(char* p, ...)
    {
           // something
    }
    ```

14. **Don't use initializer lists with objects.**

    Non-portable example:

    ```
    FooClass myFoo = {10, 20};
    ```

    Some compilers won't allow this syntax for objects (HP-UX won't), actually only some will allow it. So don't do it. Again, use a wrapper function, see Don't use static constructors.

15. **Always have a default constructor.**

    Always have a default constructor, even if it doesn't make sense in terms of the object structure/hierarchy. HP-UX will barf on statically initialized objects that don't have default constructors.

16. **Don't put constructors in header files.**

    The Visual C++ 1.5 compiler for windows is really flaky, and putting constructors into the headers seems to be one of the causes of mysterious internal compiler errors.

17. **Be careful with inner-classes.**

    Some compilers (HP-UX) generally require that types (classes, enums, etc.) declared inside of another class should be referred to with their fully scoped form (e.g., `Foo::kListMaxLen` versus `kListMaxLen`).

18. **Be careful of variable declarations that require construction or initialization.**

    Non-portable example:

```
void
A::foo(int c)
{
  switch(c) {
  case FOOBAR_1:
    XyzClass buf(100);
    // stuff
    break;
  }
}
```

Be careful with variable placement around if blocks and switch statements. Some compilers (HP-UX) require that any variable requiring a constructor/initializer to be run, needs to be at the start of the method -- it won't compile code when a variable is declared inside a switch statement and needs a default constructor to run.

Portable example:

```
void
A::foo(int c)
{
  XyzClass buf(100);

  switch(c) {
  case FOOBAR_1:
    // stuff
    break;
  }
}
```

19. **Make header files compatible with C and C++.**

Non-portable example:

```
/*oldCheader.h*/
int existingCfunction(char*);
int anotherExistingCfunction(char*);

/* oldCfile.c */
#include "oldCheader.h"
...

// new file.cpp
extern "C" {
#include "oldCheader.h"
};
...
```

If you make new header files with exposed C interfaces, make the header files work correctly when they are included by both C and C++ files. If you start including an existing C header in new C++ files, fix the C header file to support C++ (as well as C), don't just `extern "C" {}` the old header file. Do this:

Portable example:

```
/*oldCheader.h*/
#ifdef __cplusplus
extern "C" {
#endif
int existingCfunction(char*);
int anotherExistingCfunction(char*);
#ifdef __cplusplus
}
#endif
```

```
/* oldCfile.c */
#include "oldCheader.h"
...

// new file.cpp
#include "oldCheader.h"
...
```

There are number of reasons for doing this, other than just good style. For one thing, you are making life easier for everyone else, doing the work in one common place (the header file) instead of all the C++ files that include it. Also, by making the C header safe for C++, you document that "hey, this file is now being included in C++". That's a good thing. You also avoid a big portability nightmare that is nasty to fix...

Some systems include C++ in system header files that are designed to be included by C or C++. Not just `extern "C" {}` guarding, but actual C++ code, usually in the form of inline functions that serve as "optimizations". While we question the wisdom of vendors doing this, there is nothing we can do about it. Changing system header files, is not a path we wish to take. Anyway, so why is this a problem? Take for example the following code fragment:

Non-portable example:

```
/*system.h*/
#ifdef __cplusplus
  /* optimization */
inline int sqr(int x) {return(x*x);}
#endif

/*header.h*/
#include <system.h>
int existingCfunction(char*);

// file.cpp
extern "C" {
#include "header.h"
}
```

What's going to happen? When the C++ compiler finds the `extern "C"` declaration in `file.cpp`, it will switch dialects to C, because it's assumed all the code inside is C code, and C's type free name rules need to be applied. But the __cplusplus pre-processor macro is still defined (that's seen by the pre-processor, not the compiler). In the system header file the C++ code inside the `#ifdef __cplusplus` block will be seen by the compiler (now running in C mode). Syntax Errors galore! If instead the `extern "C"` was done in the header file, the C functions can be correctly guarded, leaving the systems header file out of the equation. This works:

Portable example:

```
/*system.h*/
#ifdef __cplusplus
  /* optimization */
inline int sqr(int x) {return(x*x);}
#endif

/*header.h*/
#include <system.h>
extern "C" {
int existingCfunction(char*);
}

// file.cpp
#include "header.h"
```

One more thing before we leave the `extern "C"` segment of the program. Sometimes you're going to have to `extern "C"` system files. This is because you need to include C system header files that do not have `extern "C"` guarding themselves. Most vendors have updated all their headers to support C++, but there are still a few out there that won't grok C++. You might have to do this only for some platforms, not for others (using `#ifdef SYSTEM_X`). The safest place to do `extern "C"` a system header file (in fact the safest place to include a system header file) is at the lowest place possible in the header file inclusion hierarchy. That is, push all this stuff down to the header files closer to the system code, don't do this stuff in the mail header files. Ideally the best place to do this is in the NSPR or XP header files **-** which sit directly on the system code.

20. **Be careful of the scoping of variables declared inside `for()` statements.**

    Non-portable example:

    ```
    void
    A::foo()
    {
        for (int i = 0; i < 10; i++) {
          // do something
        }
        // i might get referenced
        //  after the loop.
        ...
    }
    ```

    This is actually an issue that comes about because the C++ standard has changed over time. The original C++ specification would scope the **i** as part of the outer block (in this case function `A::foo()`). The standard changed so that now the **i** in is scoped within the `for() {}` block. Most compilers use the new standard. Some compilers (for example, HP-UX) still use the old standard. Some other compilers (for example, gcc) use the new rules, but will tolerate the old. If **i** was referenced later in the `for() {}` block, gcc will allow the construct, but give a warning about use of an "obsolete binding". So, while the code above is valid, it would become ambiguous if **i** was used later in the function. It's probably better to be on the safe side and declare the iterator variable outside of the `for()` loop. Then you'll know what you are getting on all platforms:

    Portable example:

    ```
    void
    A::foo()
    {
      int i;
      for (i = 0; i < 10; i++) {
        // do something
      }
      // i might get referenced
      //  after the loop.
      ...
    }
    ```

21. **Declare local initialized aggregates as static.**

    Non-portable example:

    ```
    void
    A:: func_foo()
    {
      char* foo_int[] = {"1", "2", "C"};
      ...
    }
    ```

    This seemingly innocent piece of code will generate a "loader error" using the HP-UX compiler/linker. If you really meant for the array to be static data, say so:

    Portable example:

    ```
    void
    A:: func_foo()
    {
    ```

```
    static char *foo_int[] = {"1", "2", "C"};
    ...
  }
```

Otherwise you can keep the array as an automatic, and initialize by hand:

Portable example:

```
  void
  A:: func_foo()
  {
    char *foo_int[3];

    foo_int[0] = XP_STRDUP("1");
    foo_int[1] = XP_STRDUP("2");
    foo_int[2] = XP_STRDUP("C");
    // or something equally Byzantine...
    ...
  }
```

22. **Expect complex inlines to be non-portable.**

    Non-portable example:

```
  class FooClass {
    ...
    int fooMethod(char* p) {
      if (p[0] == '\0')
        return -1;

      doSomething();
      return 0;
    }
    ...
  };
```

It's surprising, but many C++ compilers do a very bad job of handling inline member functions. Cfront based compilers (like those on SCO and HP-UX) are prone to give up on all but the most simple inline functions, with the error message "sorry, unimplemented". Often times the source of this problem is an inline with multiple return statements. The fix for this is to resolve the returns into a single point at the end of the function. But there are other constructs which will result in "not implemented". For this reason, you'll see that most of the C++ code in Mozilla does not use inline functions. We don't want to legislate inline functions away, but you should be aware that there is some danger in using them, so do so only when there is some measurable gain (not just a random hope of performance win). **Maybe you should just not go there.**

Portable example:

```
  class FooClass {
    ...
    int fooMethod(char* p) {
      int return_value;

        if (p[0] == '\0') {
           return_value = -1;
        } else {
           doSomething();
           return_value = 0;
        }
        return return_value;
    }
    ...
  };
```

Or

Portable example:

```
class FooClass {
  ...
  int fooMethod(char* p);
  ...
};

int FooClass::fooMethod(char* p)
{
  if (p[0] == '\0')
    return -1;

  doSomething();
  return 0;
}
```

23. **Don't use return statements that have an inline function in the return expression.**

    For the same reason as the previous tip, don't use return statements that have an inline function in the return expression. You'll get that same "sorry, unimplemented" error. Store the return value in a temporary, then pass that back.

24. **Be careful with the include depth of files and file size.**

    Be careful with the include depth of files and file size. The Microsoft Visual C++1.5 compiler will generate internal compiler errors if you have a large include depth or large file size. Be careful to limit the include depth of your header files as well as your file size.

25. **Use virtual declaration on all subclass virtual member functions.**

    Non-portable example:

```
class A {
  virtual void foobar(char*);
};

class B : public A {
  void foobar(char*);
};
```

    Another drag. In the class declarations above, `A::foobar()` is declared as virtual. C++ says that all implementations of void `foobar(char*)` in subclasses will also be virtual (once virtual, always virtual). This code is really fine, but some compilers want the virtual declaration also used on overloaded functions of the virtual in subclasses. If you don't do it, you get warnings. While this is not a hard error, because this stuff tends to be in headers files, you'll get so many warnings that's you'll go nuts. Better to silence the compiler warnings, by including the virtual declaration in the subclasses. It's also better documentation:

    Portable example:

```
class A {
  virtual void foobar(char*);
};

class B : public A {
  virtual void foobar(char*);
};
```

26. **Always declare a copy constructor and assignment operator.**

    One feature of C++ that can be problematic is the use of copy constructors. Because a class's copy constructor defines what it means to pass and return objects by value (or if you prefer, pass by value means call the copy constructor), it's important to get this right. There are times when the compiler will silently generate a call to a copy constructor, that maybe you do not want. For example, when a you pass an object by value as a function parameter, a temporary copy is made, which gets passed, then destroyed on return from the function. Maybe you don't want this to happen, maybe you'd always like instances of your class to be passed by reference. If you do not define a copy constructor the C++ compiler will generate one for you (the default copy constructor), and this automatically generated copy constructor might, well, suck. So you have a situation where the compiler is going to silently generate calls to a piece of code that might not be the greatest code for the job (it may be wrong).

Ok, you say, "no problem, I know when I'm calling the copy constructor, and I know I'm not doing it". But what about other people using your class? The safe bet is to do one of two things: if you want your class to support pass by value, then write a good copy constructor for your class. If you see no reason to support pass by value on your class, then you should explicitly prohibit this, don't let the compiler's default copy constructor do it for you. The way to enforce your policy is to declare the copy constructor as private, and not supply a definition. While your at it, do the same for the assignment operator used for assignment of objects of the same class. Example:

```
class foo {
  ...
  private:
  // These are not supported
  // and are not implemented!
  foo(const foo& x);
  foo& operator=(const foo& x);
};
```

When you do this, you ensure that code that implicitly calls the copy constructor will not compile and link. That way nothing happens in the dark. When a user's code won't compile, they'll see that they were passing by value, when they meant to pass by reference (oops).

27. **Be careful of overloaded methods with like signatures.**

    It's best to avoid overloading methods when the type signature of the methods differs only by 1 "abstract" type (e.g. `PR_Int32` or `int32`). What you will find as you move that code to different platforms, is suddenly on the Foo2000 compiler your overloaded methods will have the same type-signature.

28. **Type scalar constants to avoid unexpected ambiguities.**

    Non-portable code:

```
class FooClass {
  // having such similar signatures
  // is a bad idea in the first place.
  void doit(long);
  void doit(short);
};

void
B::foo(FooClass* xyz)
{
  xyz->doit(45);
}
```

Be sure to type your scalar constants, e.g., PR_INT32(10) or 10L. Otherwise, you can produce ambiguous function calls which potentially could resolve to multiple methods, particularly if you haven't followed (2) above. Not all of the compilers will flag ambiguous method calls.

Portable code:

```
class FooClass {
  // having such similar signatures
  // is a bad idea in the first place.
  void doit(long);
  void doit(short);
};

void
```

```
B::foo(FooClass* xyz)
{
  xyz->doit(45L);
}
```

29. **Type scalar constants to avoid unexpected ambiguities.**

    Some platforms (e.g. Linux) have native definitions of types like Bool which sometimes conflict with definitions in XP code. Always use PRBool (PR_TRUE, PR_FALSE) or XP_Bool (TRUE, FALSE).

30. **Use macros for C++ style casts.**

    Not all C++ compilers support C++ style casts:

    `static_cast<type>(expression)` (C++ style)

    `(type) expression` (C style)

    The header nscore.h defines portable cast macros that use C++ style casts on compilers that support them, and regualar casts otherwise.

    These macros are defined as follows:

    ```
    #define NS_STATIC_CAST(__type, __ptr)      static_cast<__type>(__ptr)
    #define NS_CONST_CAST(__type, __ptr)       const_cast<__type>(__ptr)
    #define NS_REINTERPRET_CAST(__type, __ptr) reinterpret_cast<__type>(__ptr)
    ```

    Note that the semantics of `dynamic_cast` cannot be duplicated, so we dont use it. See Chris Waterson's detailed explanation on why this is so.

    Example:

    Instead of:

    ```
    foo_t * x = static_cast<foo_t *>(client_data);
    bar_t * nonConstX = const_cast<bar_t *>(this);
    ```

    You should use:

    ```
    foo_t * x = NS_STATIC_CAST(foo_t *,client_data);
    bar_t * nonConstX = NS_CONST_CAST(bar_t *,this);
    ```

31. **Don't use mutable.**

    Not all C++ compilers support the `mutable` keyword:

    You'll have to use the "fake this" approach to cast away the constness of a data member:

    ```
    void MyClass::MyConstMethod() const
    {
      MyClass * fakeThis = NS_CONST_CAST(MyClass *,this);

      // Treat mFoo as mutable
      fakeThis->mFoo = 99;
    }
    ```

32. **Use nsCOMPtr in XPCOM code.**

    Mozilla has recently adopted the use of nsCOMPtr in XPCOM code.

    See the nsCOMPtr User Manual for usage details.

Stuff that is good to do for C or C++.

1. **Always use the nspr types for intrinsic types.**

Always use the nspr types for intrinsic integer types. The only exception to this rule is when writing machine dependent code that is called from xp code. In this case you will probably need to bridge the type systems and cast from an nspr type to a native type.

2. **Do not wrap include statements with an `#ifdef`.**

   Do not wrap include statements with an `#ifdef`. The reason is that when the symbol is not defined, other compiler symbols will not be defined and it will be hard to test the code on all platforms. An example of what not to do:

   Bad code example:

   ```
   // don't do this
   #ifdef X
   #include "foo.h"
   #endif
   ```

   The exception to this rule is when you are including different system files for different machines. In that case, you may need to have a `#ifdef SYSTEM_X` include.

3. **`#include` statements should include only simple filenames.**

   Non-portable example:

   ```
   #include "directory/filename.h"
   ```

   Mac compilers handle `#include` path names in a different manner to other systems. Consequently `#include` statements should contain just simple file names. Change the directories that the compiler searches to get the result you need, but if you follow the Mozilla module and directory scheme, this should not be required.

   Portable example:

   ```
   #include "filename.h"
   ```

4. **Macs complain about assignments in boolean expressions.**

   Another example of code that will generate warnings on a Mac:

   Generates warnings code:

   ```
   if ((a = b) == c) ...
   ```

   Macs don't like assignments in **if** statements, even if you properly wrap them in parentheses.

   More portable example:

   ```
   a=b;
   if (a == c) ...
   ```

5. **Every source file must have a unique name.**

   Non-portable file tree:

   ```
   feature_x
       private.h
       x.cpp
   feature_y
       private.h
       y.cpp
   ```

   For Mac compilers, every has to have a unique name. Don't assume that just because your file is only used locally that it's OK to use the same name as a header file elsewhere. It's not ok. Every filename must be different.

   Portable file tree:

   ```
   feature_x
   ```

```
        xprivate.h
        x.cpp
    feature_y
        yprivate.h
        y.cpp
```

6. **Use `#if 0` rather than comments to temporarily kill blocks of code.**

   Non-portable example:

```
int
foo()
{
  ...
  a = b + c;
  /*
   * Not doing this right now.
  a += 87;
  if (a > b) (* have to check for the
                candy factor *)
    c++;
   */
  ...
}
```

   This is a bad idea, because you always end up wanting to kill code blocks that include comments already. No, you can't rely on comments nesting properly. That's far from portable. You have to do something crazy like changing `/**/` pairs to `(**)` pairs. You'll forget. And don't try using `#ifdef NOTUSED`, the day you do that, the next day someone will quietly start defining `NOTUSED` somewhere. It's much better to block the code out with a `#if 0`, `#endif` pair, and a good comment at the top. Of course, this kind of thing should always be a temporary thing, unless the blocked out code fulfills some amazing documentation purpose.

   Portable example:

```
int
foo()
{
  ...
  a = b + c;
#if 0
  /* Not doing this right now. */
  a += 87;
  if (a > b) /* have to check for the
                candy factor */
    c++;
#endif
  ...
}
```

7. **Turn on warnings for your compiler, and then write warning free code.**

   This might be the most important tip. Beware lenient compilers! What generates a warning on one platform will generate errors on another. Turn warnings on. Write warning free code. It's good for you.

Revision History.

- 0.5 Initial Revision. 3-27-1998 David Williams
- 0.6 Added "C++ Style casts" and "mutable" entries. 12-24-1998 Ramiro Estrugo
- 0.7 Added "nsCOMPtr" entry and mozillaZine resource link. 12-02-1999 Ramiro Estrugo

---

Further reading:

Here are some books and pages which provide further good advice on how to write portable C++ code.

- *Scott Meyers*, Effective C++ : 50 Specific Ways to Improve Your Programs and Designs
- *Robert B. Murray*, C++ Strategies and Tactics
- mozillaZine has a list of books on C++, Anti-C++, OOP (and other buzzwords). This list was compiled from the suggestions of Mozilla developers.
- others?
