C++ FAQ

Matthew D. Peavy
www.GiveMeFish.com

Last Updated: 2009-07-01

[Note: This is FAQ is not yet completed.]

Summary of Contents

Contents of FAQ

0 FAQ about this FAQ

0.1 Who maintains this FAQ?

This FAQ is maintained by Matthew Peavy of Give Me Fish, LLC. You may find more information about the company at www.GiveMeFish.com

0.2 Is this FAQ copyrighted?

Yes, this FAQ is copyrighted by Matthew D. Peavy.

0.3 May I use the source code? How is it licensed?

Yes, you may use the source code for whatever purpose you would like. It is licensed under a FreeBSD license. This gives you the right to use it in commercial (proprietary) or open-source / free software.

All the examples in this FAQ have source code provided. Builds for several platforms are provided.

The source code is available here:

http://www.givemefish.com/Downloads/Downloads.php

0.4 May I link to this FAQ?

Yes, by all means. If you are quoting or paraphrasing parts of this FAQ, it is your responsibility to reference this FAQ. Please include a reference or a hyper-link.

0.5 Why is this FAQ written?

Because I was tired of repeating answers to these issues on programming user-group and forum web-sites. It's much easier to say, “see the FAQ, section 2” than to repeat the answer time after time. It's part of the “teach me to fish” initiative.

0.6 May I contribute, correct, or make a suggestion?

Yes. Please send any contribution, corrections, or suggestions to the FAQ email address: faq@GiveMeFish.com. Contributors will be recognized in the Contributors section at the end of this FAQ. Thanks in advance.

0.7 Is there any guarantee or warranty offered with this information?

No. This information is offered as is, with no warranty expressed or implied. Use this information (as well as all information) at your own risk.

0.8 What other C++ FAQs are available?

There are many. I list a few here:

Title	**Link**	**Comments**
Stroustrup FAQ	http://www.research.att.com/~bs/bs_faq.html	Bjarne Stroustrup is the original author of C++.
FAQ Lite	http://www.parashift.com/c++-faq-lite/	Marshall Cline's excellent FAQ
Comeau	http://www.comeaucomputing.com/techtalk/	Comeau – compiler writers who know their stuff

1 Language Specific

1.1 What should `main()` return?

1.1.1 Short Answer:

Either:

1) an `int`, with `0` indicating success and any other number indicating failure, or

2) `EXIT_SUCCESS` or `EXIT_FAILURE`, both defined in `<cstdlib>`.

1.1.2 Long Answer

The function `main()` has the prototype `int main(int argc, char** argv)`.

Thus the function prototype requires the returning of an integer. However, the body of the function doesn't actually have to return a value. The lack of a user-specified return value will default to `0`.

Some compilers allow the prototype to specify a return type of `void`, i.e.,

```
void main()        //This is an error
```

This is not standard-compliant and should be eschewed.

It is customary to return a `0` to indicate success and any other number to indicate failure. This may, at first blush, seem strange. Why should a value of `0` indicate success? Would a positive number not be a better choice?

First, the discussion centers on what is customary. There is nothing preventing you from returning `1` to indicate success. However, some programs that are run from the command line do check the return code and will follow the custom of interpreting non-0 values to indicate a program's failure. Thus it is recommended to follow the custom.

Second, the custom is historical. The return value was usually used to return error codes. Thus returning `0` indicated “no errors”, whereas `1` might indicate “one error” or “error type one” or something completely different.

The question has been posed whether negative numbers may be used. For portability reasons, it is not recommend. There are platforms that will truncate the low 8 bits from an `int`. Thus the “negative”ness of the number will not always be seen.

In addition, the standard provides two pre-defined values to indicate success or failure. Those are `EXIT_SUCCESS` and `EXIT_FAILURE`, both defined in `<cstdlib>`.

The values are portable and are extremely clear as to their meaning. Therefore this FAQ recommends the use of the pre-defined terms for most cases. The only exception would be if the user wanted to specify more explicit error return codes. In this case, a specialized numbering scheme could be employed (e.g., `1` = no data available, `2` = memory error, `3` = other error). The scheme could further be formalized as an enumeration.

Here is a properly formed Hello World program skeleton:

```
#include <cstdlib>
#include <iostream>

int main()
{
   std::cout << "Hello World!" << std::endl;
   return EXIT_SUCCESS;
}
```

1.2 What `#include` form should I use for standard library headers?

The `#include` pre-processor directive is seen in three different forms for standard library header files:

```
#include "math.h"
#include <math.h>
#include <cmath>
```

Which should you use? The short answer is the `<cmath>` form. As with all good questions, the complete answer takes a bit of explaining. There are several issues, and we'll take a look at each.

With the 1998 standardization of C++, four major things happened to the standard library header files.

1) The C++ specific header files were declared of the form `<header>`, dropping the earlier syntax of `<header.h>`
2) The C library files included in ANSI C++ were prefixed with the character '`c`' and the .h suffix was dropped
3) The C library files included in ANSI C++ usually implement their functions as templates rather than macros.
4) All names in both sets of these files were placed within the `std` namespace

For C++ programs, choose the non .h header files for standard library headers. This is a simple rule that's easy to follow. For example, choose `<iostream>` rather than `<iostream.h>`. Or choose `<cmath>` rather than `<math.h>`.

The old .h variety of the C++ header files are pre-standard (or pseudo-standard, as Meyers calls it in Effective C++) and are considered deprecated. This means you definitely should not use them (although compiler support may likely last for a long time to come). Examples of the deprecated header files are: `<complex.h>`, `<iostream.h>`, `<limits.h>`, etc.

All the functionality defined within the non .h standard library header files are declared within the `std` namespace. Therefore if you convert your program to use the C++ version of the C libraries (e.g., you change from `<math.h>` to `<cmath>`), the functions will be hidden within the `std` namespace. This will require you to either qualify the functions with `std::` or to place a `using namespace std;` directive at the top of your .cpp file (not within your .h file! See next item.)

There is one specific header file that can cause confusion. That is the family of `<string>`, `<string.h>`, and `<cstring>`. In summary, `<string>` is the C++ string class header file. The other two are the `char*` header files. `<string.h>` is the old C header file, whereas `<cstring>` is the modern `std` wrapped version. So both `<string>` and `<cstring>` can be used legitimately within a modern C++ program, each offering different library support. They are not mutually exclusive. However, `<string.h>` should be replaced with `<cstring>`.

The following table summarizes the header files that make up the C++ standard library, as well as the C Header files that are included in ANSI C++ with the new "`c`" prefix nomenclature and the original ANSI C file equivalents. Files in the last column should not be used in C++ programs.

C++ Standard Header Files		**C Header Files Included in ANSI C++**	**Equivalent File in ANSI C**
		Note: Functionality within `std` namespace	Note: Don't use these in C++ programs
`<algorithm>`		`<cassert>`	`<assert.h>`
`<bitset>`		`<cctype>`	`<ctype.h>`
`<complex>`		`<cerrno>`	`<errno.h>`
`<deque>`		`<cfloat>`	`<float.h>`

C++ Standard Header Files		C Header Files Included in ANSI C++	Equivalent File in ANSI C
`<exception>`		`<ciso646>`	`<iso646.h>`
`<fstream>`		`<climits>`	`<limits.h>`
`<functional>`		`<clocale>`	`<locale.h>`
`<iomanip>`		`<cmath>`	`<math.h>`
`<ios>`		`<csetjmp>`	`<setjmp.h>`
`<iosfwd>`		`<csignal>`	`<signal.h>`
`<iostream>`		`<cstdarg>`	`<stdarg.h>`
`<istream>`		`<cstddef>`	`<stddef.h>`
`<iterator>`		`<cstdio>`	`<stdio.h>`
`<limits>`		`<cstdlib>`	`<stdlib.h>`
`<list>`		`<cstring>`	`<string.h>`
`<locale>`		`<ctime>`	`<time.h>`
`<map>`		`<cwchar>`	`<wchar.h>`
`<memory>`		`<cwtype>`	`<wtype.h>`
`<new>`			
`<numeric>`			
`<ostream>`			
`<queue>`			
`<set>`			
`<sstream>`			
`<stack>`			
`<stdexcept>`			
`<streambuf>`			
`<string>`			
`<typeinfo>`			
`<utility>`			
`<valarray>`			
`<vector>`			

1.3 What does `using namespace xxx` mean? Why should I not use it in a header file?

The `using` declaration brings the stated namespace (in our example, `xxx`) into scope. This means that none of the names within that namespace has to be qualified. For example, the following two functions incorporate the standard `string` and `iostream` classes. The first takes advantage of a `using` declaration, the second doesn't. They are functionally equivalent.

```
void func()
{
   using namespace std;       //Acceptable use of "using"
   string s1("Hello World.");
   cout << s1 << endl;
}

void func2()
{
   //No using declaration, so must qualify with std::

   std::string s1("Hello World.");
   std::cout << s1 << std::endl;
}
```

While the `using` declaration has saved some typing and made things a bit more succinct, it is important not to include a `using` declaration within a header file. The reason for this is because the `using` declaration will be "in effect" for every file that includes the offending header file. In addition, every file that includes a header file that includes the offending header file will also be subjected to the `using` declaration.

In essence, such a `using` declaration increases scope greatly and may become global (or nearly so) for your project. The intent of namespaces is to hide names and isolate them from the global namespace. Inserting a `using` declaration in a header file undermines these objectives. Therefore don't include a `using` declaration within header files.

1.4 What is an anonymous (or unnamed) `namespace`?

The use of the `namespace` keyword followed by an open curly bracket (rather than by a name and then a curly bracket) creates an anonymous or unnamed namespace. The names that are declared within this namespace are in effect visible only to the current file. This is a technique to localize and hide functions that are only used within one source file.[1]

If the function is only used within a single source file, then it is usually a good idea to place it within an anonymous namespace. If it is used in more than one file, it cannot be placed within an unnamed namespace. This method is preferred to (and replaces) the use of the `static` keyword for specifying internal linkage.

If you are using one, it is common practice to place the anonymous namespace near the top of a source file and group all anonymous namespace functions within that single namespace. However it is permissible to have multiple unnamed namespaces throughout a translation unit. Every translation unit may contain its own anonymous namespace.

An example of an anonymous namespace within a .cpp file might look like this:

```
//Anonymous namespace
namespace {
  void foo();  // foo() is only visible within this translation unit
}
```

1.5 What is the difference between ++var and var++? Which should I prefer?

1.5.1 The short answer

There is no way to say that one form is always better. The use of one may be wrong in certain situations. However, it is correct to prefer the pre-increment `++var` form for efficiency if the post-fix behavior is not necessary [but please, please read the long answer.]

1.5.2 The long answer

Functionally, the two operators will differ in the precedence of operation, the parameter type, and often their return type. This may lead to different behavior between the prefix and postfix operators. A short example will help illustrate the difference.

```
1:     int x, y, z;
2:
3:     x = 1;      // x = 1
4:     y = ++x;    // x = 2, y = 2
```

1 This is accomplished in practice through the use of a unique name which the compiler assigns to the anonymous namespace. The names declared within that namespace are brought into scope automatically via a using declaration.

```
1:     int x, y, z;
5:     z = x++;     // x = 3, z = 2
6:     x++;         // x = 4
7:     ++x;         // x = 5
8:
9:     cout << x++;       // 5
10:    cout << x;         // 6
11:    cout << ++x;       // 7
12:    cout << x;         // 7
```

Clearly, the precedence does matter when used for assignment in lines **4** and **5**. Lines **6** and **7** are functionally interchangeable, since there is no concern for precedence.

In line **9**, the call to `cout` is passed the un-incremented value of `x`, whereas in line **11,** the value is first incremented before being passed.

You can remember the precedence order by associating prefix with "increment and fetch" and postfix as "fetch and increment." [Meyers97]

The two forms of the increment operator for class objects are actually different function calls. The difference is made by an `int` argument that is silently passed to the postfix operator. For example, here is a prototype of the two operator functions:

```
MyClass& operator++();              //Prefix,   ++var
const MyClass operator++(int);      //Postfix,  var++
```

It is important to note that the return types differ. Prefix often returns a non-`const` reference whereas postfix often returns a `const` object. While this is not always true, it is often true[1]. Returning a reference will execute faster than returning a copy of the object. Thus when the precedence issue is not important, prefer the pre-increment form.

For built-in types (e.g., `int`) and some standard types (e.g., `complex`), the different calls are identical in speed of execution. [Sutter00]

A common case where pre-increment should always be used is within standard for-loop iteration.

```
for(vector::iterator vi = vect.begin(); vi != vect.end(); ++vi)
{
      ...
}
```

Because an iterator is being used (which is an object), prefix will be more efficient than postfix.

1 There are good arguments for writing your classes so that they do so, but this another topic.

1.6 What is the difference between `NULL` and `0` for pointers? Which should I prefer?

In standard C++, `NULL` is defined as an integral constant with a `0` value. Often times, the standard library implementation will macro define `NULL` to be zero, as in:

```
#define NULL 0
```

In C, `NULL` was often defined (via a macro) as `0` <u>or</u> as `((void *)0)`, which are two different types. Problems could arise if the set of header files compiled and linked against are specific to C rather than C++. The assumption that `NULL` is equivalent to `0` may not be valid in this case.

Assuming you are using pure C++, you can thus use either `0` or `NULL` interchangeably when assigning to a pointer. Which should you prefer? The tendency has been moving towards using `0` rather than `NULL`. However, if a standards-compliant compiler is used, this becomes a style issue rather than a substantive issue. If you want a definite recommendation, Bjarne Stroustrup (the father of C++) recommends the use of `0` rather than `NULL` for pointers.

The upcoming C++0X standard will introduce the keyword `nullptr`.

1.7 Can I delete a null pointer safely?

Yes. The standard guarantees that a null pointer (one that equals 0) may be safely deleted. The pointer need not have ever been set to point to a dynamically allocated object.

However, remember that an uninitialized pointer will not, in general, point to `0`. Thus the deleting of an uninitialized pointer should always be considered an error.

Therefore it is recommended that pointers that are not initialized to a defined memory location be initialized to `0` to avoid accidental deletion of an uninitialized pointer.

1.8 What is the class initialization list and why should I use it?

The class initialization list is a list that is defined in the constructor (specifically just after the argument list but before the body of the constructor). It is used to initialize some or all of the class member variables (except static data members). In addition, it is used to initialize base classes, if applicable. The initialization may use whatever data is available at the time, including the parameters passed into the constructor.

The list should be used used whenever possible.

First, the list must be used in certain cases. Any `const` data variables must be initialized in the list. References must also be initialized in the list (whether `const` or not).

For example, give then class Example:

```
class Example {
public:
   Example(SomeClass& aRef, double aVal, const string& aString);

private:
   SomeClass&     m_ref;
   double         m_dblValue;
   const int      m_aConstInt;
   SomeClass*     m_aPointer;
   string         m_aString;
};
```

The constructor and initialization list might look like this:

```
Example::Example(SomeClass& aRef, double aVal, const string& aString)
: m_ref(aRef), m_dblValue(aVal), m_aConstInt(7),
  m_aPointer(0), m_aString(aString)
{
      //Other constructor activities happens here.
}
```

In this case, the reference and `const` value must be initialized using the initialization list. The other member variables do not require the use of the initialization list but should, none the less, use it for the second and third reasons listed.

Second, efficiency may be achieved through the initialization list. For the above `Example` class, if the initialization list is not used, then the `m_aString` variable will automatically be constructed once. It must then also be assigned in the body of the constructor. Whereas if the initialization list is used, the copy constructor is called only once.

Third, the list should be used to initialize data at the earliest possible moment. This can help avoid errors of using uninitialized member variables. If the constructor is considerable in length and the initialization list were not used, a programmer might inadvertently attempt to use an uninitialized member variable (including the use of uninitialized pointers). This is automatically avoided by sticking to the rule of using the initialization list wherever possible.

1.9 For `std::vector`, is `--myVector.end()` equivalent to `myVector.end()-1`? Which should I prefer?

These two are not equivalent. In fact, `--myVector.end()` will likely fail and should always be avoided. Depending on implementation, calling `myVector.end()` may return a temporary of a built-in type. C++ does not allow for the modification of temporaries of built-in types. Not all compilers will choose such an implementation, but the standard does not forbid it. Thus using `--myVector.end()` is not safe.

When you need to access the last (one less than `end`) element in a vector, you may use `myVector.end()-1` safely so long as the vector contains at least one element.

2 Compiler Specific Issues

2.1 Should I use `#pragma once` or `#ifndef / #define / #endif`?

While the `#pragma` preprocessor command is defined by the standard, it's implementation is compiler specific and therefore not portable. Some compilers (including Microsoft Visual C++) use `#pragma once` to guarantee that a header file is only included once. But others will issue an error if they encounter `#pragma once`.

The `#ifndef / #define / #endif` inclusion guard paradigm is portable and should therefore be used. The following is an example of a header file implementing the recommended method:

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
   MyClass();
};

#endif
```

2.2 Why should I not use _ (and especially not __) to prefix variable or function names?

2.2.1 Short Answer

To avoid accidental name conflicts.

2.2.2 Long Answer

Some underbar-prefixed names are reserved by the standard for compiler implementation. “For example, names with leading underscores are technically reserved only for nonmember names, so some would argue that this isn't a problem and that class member names with leading underscores are fine. That's not entirely true in practice, because some implementations `#define` macros with leading-underscore names, and macros don't respect scope. [Sutter00], Item 21, pg. 74.”

$EXPAND THIS

2.3 How do I call a system command?

Within `<cstdlib>`, the command `system` is available. The `system` call takes a `const char *`, which will be executed by a command processor on the target environment. It returns an `int`, which is the value returned by the command processor in response to the command.

Here is a short program that prints the current working directory contents.

```
#include <cstdlib>
#include <iostream>

int main() {
  system("ls -la");
  return EXIT_SUCCESS;
}
```

$EXPAND THIS – and add references.

2.4 Does `NULL` always mean the same thing?

Yes. Within a standards compliant C++ environment, you can be sure that NULL is an integral constant with 0 value. Be aware that C libraries may define NULL as `((void *)0)`. See item 1.6 in this FAQ for more details.

3 Style

3.1 Are these style issues mandatory, suggested, or merely preference?

These issues are all style issues and should not be considered mandatory. Code that contravenes these points may still be correct and fully functional. However, it is suggested that you follow these guidelines. There is fairly wide-spread consensus on most of these issues, though some software shops continue (for historical reasons or otherwise) to contravene some or all of the suggestions laid out here.

3.2 What is the difference between `func(void)` and `func()`? Which should I prefer?

Nothing is functionally different between the two. The newer, C++ style is the latter, and that's what I recommend using.

3.3 Should I `#include` files in the header or source file?

Include as many files as possible from the source file, and then include the remaining files from the header file. This can reduce build times considerably. It also may reduce file dependencies.

If you can get away with including files from the source file, do so. This means that if you are using only a pointer or reference to a class within the header file, you should use a forward reference to that class rather than a full `#include`.

In addition, many classes are only seen within the source file. For example, if you're doing some string manipulation, you may use the `stringstream` class within the cpp file. `stringstream` is not used within the interface at all. Then you should include `<sstream>` only in the source file.

3.4 Why should I avoid (or minimize) using `/* */` style comments?

The C-style comment blocks that use `/* */` should be avoided because they do not nest. For example, if you intend to comment out a function that is already using C-style comments, you will introduce an error.

```
/*
int calcSquareSum()
{
   int squareSum =0;               /* Used to sum */
   for(int i=0; i<10; ++i)
      squareSum += (i * i);
   return squareSum;
}
*/
```

Your editor's code colorization may help you to visualize what blocks of code are commented, making this type of error more obvious. And you might not want to type or delete the `//` on each line (although there are macros for doing this). It's best to minimize the use of C-style comments, and certainly avoid them for single line comments as shown in the example above.

3.5 Should I use Hungarian notation?

No. Hungarian notation is out-dated and obsolete. In today's modern, generic C++, it is often times impossible to denote type information.

For example, what is the type of a `vector<int>`? And how does that differ from a `vector<double>`?

C++ is a (mostly) strongly typed language, so compiler-enforcement of type will keep you from going astray (at least as far as type is concerned!)

In addition, the prefixes can become long-winded and a distraction. Typing `lpsz` before every “long pointer to a null-terminated ASCII string” becomes a real waste of time and screen space.

This practice still has its strong adherents, and many software shops continue to require the practice.

3.6 Should `public`, `protected`, or `private` declarations come first?

This is a matter of personal taste, and one that certainly doesn't have a right or wrong answer. I would recommend public declarations be placed before private. The reason being that the user of the class only has access to the public section of the class. Thus the private members and functions should be placed out of the way (i.e., at the end). Private information is an implementation detail that the class user should not be interested in nor burdened with. And `protected` falls naturally in the middle. I strongly dislike having multiple class access specifiers of the same type within a class (e.g., seeing `public`:, then `private`:, followed by a `public`: specifier.) I would rather group all my public functions at the top, and put my private data and functions below.

3.7 Should I avoid the `goto` statement?

Yes, in general. The `goto` statement exists within the C++ language, inherited from C. The statement can make programs incredibly difficult to understand and maintain. It is error prone. There is wide spread agreement in the programming community that `goto` should be avoided for general control, and many argue to avoid it in all cases.

One example given to justify the use of the `goto` statement is breaking out of deeply nested loops when a condition has been met. The use of `goto` in this case can simplify the look of the code. Within very this specific example, `goto` could be an acceptable control structure. As a general style rule, it should be avoided. Francis Glassborow summed it up succinctly on the newsgroup comp.lang.c++.moderated:

```
“I am not of the school that says 'never use goto' but I
cannot remember when I last found the need for one.”
```

3.8 Why should I avoid `public` data in classes?

One word: encapsulation. $EXPAND

3.9 Why should I avoid global variables?

Because global variables are error prone and needless. $EXPAND

3.10 Why should I avoid the use of the preprocessor for variable definitions?

Despite the nearly universal recommendations from C++ authorities on this issue, it is common practice to see modern code (including newly written code) that includes the use of preprocessor variable definitions. This is a result of ignorance of the dangers of the practice combined with current-practice inertia within some shops.

3.11 Why should I avoid the use of the preprocessor for pseudo macros?

For most of the same reasons as the previous FAQuestion. Instead of using error prone pseudo-macros, use a template function. You get all the functionality, but you also have the type-safety that you desire.

3.12 Should I avoid multiple inheritance?

Not necessarily, but use multiple inheritance (MI) carefully. MI, when used properly, works as advertised. It solves the problem of modeling a situation where an object logically "is a" two different base types.

However, MI can get you into trouble if you're not careful. This is especially true when considering the "diamond of death" inheritance hierarchy, whereby two two different child classes inherit from a parent class, and a fourth grand-child class inherits multiply from the two children classes.

Many shops have simply outlawed MI. This shouldn't be seen as a tragedy, as you should still be able to write clear and correct C++ without MI. But a blanket rule like this may also be overly stifling. See Scott Meyer's description in Effective C++ for more details.

3.13 Should I declare all the variables used in a function at the top of that function?

No. This violates the programming ideal of minimizing variable scope. You should strive to scope variables as tightly as possible. This means placing the variable at the tightest scope (inner most scope) as possible. In addition, variables should be declared only when needed.

There is no programmatic benefit to placing all the variables at the top of a function. Conversely, the benefits of tight scoping are clear.

It can be argued that placing variables at the top of a function allows a new user of the function to immediately see all the variables in one, concise are of code. However, that usually comes from programmers who are accustomed to this style and therefore desire to see all the variables in one place. Seeing the variables at the beginning of the function doesn't help the reader understand the function. In fact, it burdens the reader by declaring the variables too early, possibly forcing the reader to look back to the beginning of the function to find the variable's type when it is first used.

The two following functions illustrate the point:

```
void notTightlyScoped()
{
   int i, iSquared;
   string str;

   for(i =0; i<5; ++i) {
      str += "X";
      iSquared  = i * i;
      cout << "String: " << str << ". ";
      cout << "i squared = " << iSquared << endl;
   }
   cout << "String Final: " << str << endl;
}
```

```
void tightlyScoped()
{
   string str;
   for(int i =0; i<5; ++i) {
      str += "X";
      int iSquared  = i * i;
      cout << "String: " << str << ". ";
      cout << "i squared = " << iSquared << endl;
   }
   cout << "String Final: " << str << endl;
}
```

One might argue that iSquared should be placed between the two `cout` statements. Indeed, that would be tighter scoping. However, I include this example to illustrate that code clarity and flow may sometimes trump the scoping rules slightly. It makes sense to me to keep the two `cout` statements together.

3.14 Should temporary for-loop variables be declared within the loop statement?

Yes, if possible. For example:

```
for(int i=0; i<10; ++i)
   cout << i << endl;
```

is preferable to:

```
int i;
for(i=0; i<10; ++i)
   cout << i << endl;
```

if `i` is not needed later in the function.

The only reason not to include the variable declaration within the for-loop is if you need access to that value after the for-loop has ended.

Some argue that several for loops in one function benefit for only declaring one integer value and re-using that value. This misplaced zeal for optimization is not only <u>not</u> an optimization (at least with today's aggressive-optimizing compiles), but it is error prone to boot.

Note that Microsoft's Visual C++ compiler (through and including version 2005) contains a compiler switch that will recognize or "force" for-loop scope conformance. They have included this switch since so much legacy code written using Visual C++ violates the standard on for-loop scope conformance.

Unfortunately the switch is set by default to not conform with the standard. This leads many programmers to assume that their C++ is legal when it is not. One should always start out a project by setting the switch to conform to the standard for-scope rules. This can be accomplished by changing the Project properties -> C/C++ -> Language -> For Conformance in For Loop Scope, set to YES. This is the `/Zc` compiler switch.

3.15 What should I name the inclusion guard variable?

Inclusion guards are used to force the compiler to only view header files once. They are implemented using the preprocessor directives `#ifndef`, `#define`, `#endif`.

There are no hard-and-fast rules as to what the inclusion guards should be named. This is a style issue that should be decided on and applied consistently throughout a project. Although the style may differ from project to project, the style should remain consistent throughout a single project.

One common style is to use all-caps and name the variable as the header file, substituting and _ for the . of the file ending. For example, for the MyClass.h header file, I would use:

```
#ifndef MYCLASS_H
#define MYCLASS_H
```

If the project uses namespaces, I opt to include the namespace that the class resides in as well. For example,

```
#ifndef MYNAMESPACE_MYCLASS_H
#define MYNAMESPACE_MYCLASS_H
```

The goal is for uniqueness. The class and namespace combination should nearly always guarantee that uniqueness. For situations where there is more than one namespace at play in a header file, just develop your own strategy that will result in a unique pre-processor definition and stick to it.

4 Techniques

4.1 How do I convert a `string` into a numerical type (i.e., `int` or `double`)?

An easy way to convert a string into a number is using the stringstream class. This class takes care of most of the work for you. For example:

```
string numInString("96.7");
double number;
stringstream converter;
converter << numInString;
converter >> number;          //number now contains 96.7
```

This glosses over several issues such as precision (see the item on precision in this chapter for more details) and, more basically, verifying whether the string holds a number in the first place.

One of the disadvantages of this method is the introduction of the temporary variable `converter`. This can be avoided by using the boost lexical_cast library. The header file `boost/lexical_cast.hpp` must be included. The above could be condensed into the following line. This should be preferred if boost can be used in your project.

```
double number = boost::lexical_cast<double>(numInString);
```

A quality discussion of different converting and formatting options can be found here: http://www.gotw.ca/publications/mill19.htm

4.2 How do I convert a number to a `string`?

In the same way as in the previous question – using `stringstream` or `lexical_cast<string>`.

```
string numInString;
double number(96.7);
stringstream converter;
converter << number;
converter >> numInString;          //numInString now contains "96.7"
```

If you need to get to a character array (a C-style string):

```
char* cStyleString = numInString.c_str();
```

For `lexical_cast`, the following will work:

```
string numInString = boost::lexical_cast<string>(number);
```

4.3 How do I tell if a character is a digit or not?

Use the standard function `std::isdigit()` within the `<cctype>` header. As an example, this custom written function would accomplish the same thing.

```
//This function returns whether the char passed in is a digit or not.
//Consider using the standard isdigit() function.
bool isDigit(char c)
{
   return(c >= '0'  &&  c <= '9');
}
```

Within `<cctype>`, several other functions are made available for classifying and converting characters, such as `isspace()`, `toupper()`, etc.

4.4 How can I tell if a `string` contains a number (i.e., an `long`, or `double`)?

The functions `strtol()` and `strtod()` can accomplish this task. They can be found within the `<cstdlib>` header. They are of the form:

```
long strtol(const char* s, char** endptr, int base);
double strtod(const char* s, char** endptr);
```

A `char` array containing the value to be tested is passed to the functions. If the conversion attempt fails, the second argument will be set equal to that of the first. Thus to test whether a C-string value contains a `long` or `double`, call the conversion function and check whether `(s != endptr)`. Note that in testing this way, a string that contains a `double` will test `true` for `long`, since the `long` value is present within the string.

Another option is to use the boost `lexical_cast` template function. If the conversion is successful, then the `string` tested contains a value of the template parameter type used in the `lexical_cast`. If the test fails, a `bad_lexical_cast` object is thrown. Therefore this test must by within a `try` / `catch` block. A simple function can help to isolate this test.

```
template<typename T>
bool lexTest(const string& s)
{
  try {
    boost::lexical_cast<T>(s);
    return true;
  }
  catch(boost::bad_lexical_cast&) {
    return false;
  }
}
```

Note that a call to `lexTest` with a `string` containing a `double` will not return `true` if the template parameter is `long` (which is in contrast to the behavior of the `strtod()` function above).

4.5 How do I trim extra spaces off the beginning or end of a string?

One method is to use the `string` member function `find_last_not_of` in order to find the last character that is not a space. Knowing this position, take a sub-string starting from position 0. Use the position of the `find_last_not_of +1` as the number of characters for the sub-string function. If no space was found (resulting in a position of `npos`), simply do nothing with the string `line`.

```
string line("Example line of text.  ");

string::size_type pos = line.find_last_not_of(' ');
if(pos != string::npos)
   line = line.substr(0, pos+1);
```

Another source of excellent string processing functions is the Boost String Algorithms Library. This library contains similar trim functions and much more.

4.6 How do I use the command-line arguments passed to `main`?

Command-line arguments are passed to `main` each time the program is executed. Remember, the prototype of `main` that takes arguments is:

```
int main(int argc, char** argv)
```

Alternatively, this could be written as:

```
int main(int argc, char* argv[])
```

The first argument (`argc`) denotes how many command-line arguments are being passed, while the second argument (`argv`) actually holds the arguments. `argv` is a pointer to an array of `char` pointers.

The number of command-line arguments will always be at least one. The first character string in `argv` will contain the name of the application, regardless of whether any command-line arguments were specified. If the number of arguments indicated by `argc` is greater than one, then command-line arguments were specified when running the application. They will be stored in `argv` starting at `argv[1]`.

I find it preferable to use a vector of strings in place of the error-prone array of character arrays inherited from C. Thus I usually incorporate the following function to parse the command-line arguments for me.

```
std::vector<std::string> parseCommandLine(int argc, char** argv)
{
   //Parse out command-line args.  Ignore the first arg, which is
   // the application name.  If you don't want to ignore this arg,
   // loop from i=00 to i<argc rather than from i=1 to i<argc.
   vector<string> commandLineArgs;
   for(int i=1; i<argc; ++i)
      commandLineArgs.push_back(argv[i]);
   return commandLineArgs;
}
```

In this form, the programmer may do with the arguments what he or she would like. They are readily accessible. Note that this function will ignore the first argument, which is guaranteed to be the application name. If this argument is needed, simply alter the for-loop to start at `i=0` rather than `i=1`.

A more succinct version was offered by Igor Mikushkin as:

```
std::vector<std::string> parseCommandLine(int argc, char** argv)
{
   return vector<string>(argv+1, argv + argc);
}
```

Arguments may be safely ignored. In fact, the programmer may explicitly ignore all command-line arguments by writing:

```
int main()
{
      ...
}
```

Another good choice is to use the Boost Program_options library. This library contains functions for command-line and other program options handling.

4.7 How do I create and use dynamic multi-dimensional arrays?

There are many methods from which to choose, and the choice will depend on the exact needs for a multi-dimensional array. For mathematics, many numerical class libraries exist which handle arrays of any dimension. For example, see the TNT library, the Matrix Template Library (MTL), the Boost uBLAS library, or several others at ooNumerics. [Links are provided in the bibliography.]

Another solution is to write a multi-dimension array class which implements your needs exactly. However, this should only be done if absolutely necessary (i.e., if none of the present libraries does what you need).

The Boost.MultiArray library provides a very well written library that may accomplish what you're looking for. It's not specifically a mathematical library. It handles multi-dimensional arrays of various sizes. It is designed to behave similarly to STL containers and is extremely efficient. This should be a candidate for your multi-dimensional needs.

For very simple needs, however, I will recommend the following: a vector of vectors. This has fulfilled my needs countless times and is extremely easy to implement. You also don't need to obtain or link external libraries.

In the following example, I'll implement a 3x3 matrix that contains the integers 0 to 8.

```
vector< vector<int> >  myArray;   //Note space between > >
for(int row=0; row<3; ++row) {

   //Add another row
   myArray.push_back(vector<int>());

   for(int col=0; col<3; ++col) {
      int value = row*3 + col;
      myArray[row].push_back(value);
   }
}

//Print out the values in the array
cout << "The values contained in the array: " << endl << endl;
for(int row=0; row<3; ++row) {
   for(int col=0; col<3; ++col) {
      cout << myArray[row][col] << " ";
   }
   cout << endl;
}

//Now print out the value at the 1,2 position. Note vectors are base 0.
cout << endl << "The (1,2) cell of the array contains: " <<
   myArray[1][2] << endl;
```

A couple points of interest here. First, the declaration of `myArray` must include a space between the two `>` characters. Otherwise the compiler will interpret this as an extractor (`>>`) operator. [This may change in the upcoming C++ 0x standard due out by the end of the decade.]

Another is that the array does not need to be square, or even rectangular. In fact, each row in the array can be as long or as short as desired (including 0 length).

Accessing values may be accomplished via `operator[]` or through iterators.

Finally, C++ standard vectors are base 0, so indexing begins with 0.

4.8 How do I tell if a number is even or odd?

```
bool isEven(int i)
{
   return (i%2 == 0);
}
```

or:

```
bool isOdd(int  i)
{
   return (i%2 != 0);
}
```

4.9 How do I copy one `stringstream` to another?

The copying of `stringstream` objects is specifically disallowed. The reason for this is the lack of proper semantics for certain copy functions.

However, the contents of one `stringstream` can easily be copied to another using the following:

```
//Copy from stringstream s1 to s2, assuming s2 is empty
s2 << s1.str();
```

4.10 How do I tell if a `stringstream` is empty?

You can use this function (perhaps in an anonymous namespace within the source file that needs it) in order to determine if a `stringstream` is empty.

```
//This function returns whether a stringstream object is empty.
template <class S>
bool isEmpty(S& a)
{
   return(S::traits_type::eq_int_type(a.rdbuf()->sgetc(),
      S::traits_type::eof())
   );
}
```

4.11 How do I generate random numbers?

Random numbers are the subject of a wide body of literature. There are manifold techniques for generating “pseudo-random” numbers. A full discussion of the theory and implementation of random numbers is beyond the scope of this FAQ. C++ also includes a simple random number generator which is covered below.

4.11.1 Strongly Random Numbers

The Boost libraries contains a well documented, well written, free random numbers library. These classes should be acceptable for strongly random generation.

There are many libraries available at OO Numerics (www.oonumerics.org).

Finally, a search of web will results numerous additional random number libraries.

4.11.2 Simple Random Numbers

For simple cases (and understand this to mean "not acceptable for scientific or technical programming requiring a true degree of randomness"), the `rand()` function within the `<cstdlib>` header should do.

The `rand()` function return an integer between `0` and `RAND_MAX`. `RAND_MAX` is an implementation-specific variable (thus it may vary between compilers and / or platforms.) You may test against or use `RAND_MAX` if you need.

As an example, the following values were obtained for `RAND_MAX`:

- Microsoft Visual C++ 2003 on WinXP = 32767.
- Microsoft Visual C++ 2005 on WinXP = 32767.
- Cygwin (Pentium 4, Windows XP) using gcc 3.3.3 = 2147483647.
- Linux (Pentium 4, Mandrake 10.1) using gcc 3.4.1 = 2147483647.
- Linux (Pentium M, Debian) using gcc 4.3.3 = 2147483647.

Note that `RAND_MAX` may be as low as 32767, but will be no lower.

The following example illustrates the use of `rand` and `RAND_MAX`:

```
void item_4_11_maxRandomNumber()
{
   cout << "The largest possible random number is: " << RAND_MAX
        << endl;
}

void item_4_11_generateRandomNumbersInRange()
{
   //Seed the random number generator
   srand( static_cast<unsigned>(time(0)) );

   //Print out 10 random numbers
   for(int i=0; i<10; ++i)
```

```
      cout << i << ": " << rand() << endl;
}
```

It is important to know that random numbers generated by `rand()` will eventually repeat. In order to initialize the random number generator (often referred to as "seeding" the generator), one may use the `srand(unsigned)` function.

Any number may be used to seed `srand()`. However, seeding the generator with a hard-coded value makes little sense. This will cause your program to use the same "random" sequence each time it is run. A practical way around this is to seed the number with a value obtain based on the system time. The following line illustrates `srand`'s usage:

```
srand( static_cast<unsigned>(time(0)) );
```

The `time(0)` function will return a value (in seconds) from a specific date. It should be cast into an `unsigned int`.

4.11.3 Simple Random Numbers within a Range

Given the `int` values `lowerBound` and range, you can use the following expression to obtain a random number in the range [`lowerBound`, `lowerBound + range`). Note that the notation **[ )** means that the range includes the value `lowerBound` (i.e., that `lowerBound` may be selected), and includes all numbers up to, but not including, `lowerBound + range`. For example, choosing `lowerBound` as 50 and `range` as 430, you can expect any number between and including 50 to 479 to be selected.

```
int randNum = lowerBound + static_cast<int>(range * (rand() /
    (RAND_MAX + 1.0)));
```

One interesting note is that the second set of parentheses (just preceding `rand()`) was necessary for the correct program execution when compiled under gcc and run on Linux. However, the example ran fine without that set of parentheses when compiled under Microsoft Visual Studio 2005 and run on Windows XP.

4.12 How do I print a number with a certain precision?

The `stringstream` class is robust and covers a lot of programming needs. This example shows only how to set the precision for printing of an `int` or `double` value.

The I/O manipulators are held in `<iomanip>`, so don't forget to `#include <iomanip>`, nor `std::` or `using namespace std;` as necessary.

The fixed format flag must be set before setting the width of precision. This only needs to be done once, even if the precision is reset many times.

Set width only (for `int` values):

```
stream << std::fixed << std::setw(5) << intVal;
```

Set width and precision (for `double` values) In this case, we're setting the width of the number to 15 and the precision to 5:

```
stream << fixed << setw(15) << setprecision(5) << doubleVal;
```

Note that `std::fixed` is different than `std::ios::fix`. The former is a flag to be called with the stream's `setf()` function. The latter is a manipulator that can be “inserted” into the stream via the `<<` inserter.

4.13 How do I print columns that align?

Sometimes you need to print columns of numbers and text that must align in a certain way. For example, the left column is an integer that should align right, the next two columns are doubles that should align their decimal points, followed by a column of left-aligned text. The desired output might look like this:

```
  8     12.4556   8553.3    axw
 12    244.5454    102.4    qropc
 54      8.1017    599.3    aszv
102     11.5555  10112.3    pyhkmm
```

You will want to set the stream to fixed, so do this before you enter the loop. The width must be set each time you use an inserter or extractor. It doesn't stay set (or maintain “state”). The precision, which is used just for the doubles, does maintain state. But since we want the two columns to have different precision, it will have to also be set anew when printing each column. And we'll set the left and right justification as appropriate. Justification defaults to right, but we'll need to explicitly set it to right in the loop, since it maintains its state of `right` coming into the 2nd iteration of the loop.

```
out << fixed;  //Set the float field to fixed using a manipulator

//Iterate over the vector of Data objects
for(vector<Data>::const_iterator dataIter = data.begin();
     dataIter != data.end(); ++dataIter) {

   //Must set the width before each extractor / inserter call.
   //Also, start out with right justified.
   out << right << setw(5) << dataIter->intData();
   out << setprecision(4) << setw(13) << dataIter->doubleData1();
   out << setprecision(1) << setw(9) << dataIter->doubleData2();
   //Need to leave space between right and left justified columns
   // otherwise they would touch
   out << "   ";
   out << left << dataIter->stringData() << endl;  //Add new-line
}
```

4.14 How do I return more than one value?

Functions are designed to return either `void` or a single entity (which could be an atomic type, an object, a reference, or a pointer). However, a programmer would sometimes like to return more than one value.

One common way around this issue is for the programmer to choose to pass in non-`const` references to the function, planning to alter the value of the underlying variable during the course of the function. Using this method, however, it is easy to overlook that you are passing in references to a function. The prototype looks very similar to pass-by-value, differing only by a `&` character. Non-`const` references are used more seldom than `const` references, and the lack of `const` may cause one to forget about the reference symbol. This can become a problem when multiple programmers are working on a piece of code.

In addition, it is more natural for functions that are designed to calculate values to return an argument containing those values. It is more semantically proper to return values from a "calc" type function than it is for the function to set references that are passed in.

Programmers have often solved this problem by writing dedicated `struct`s to accomplish the task. The `struct` works for returning aggregated data from a single function call. This technique avoids the criticisms leveled above, but it adds overhead to the programmer's job since an additional type must be written.

The pre-processor was sometimes used to automate this task. But that gets us back to the bad old days of un-type safe macros.

Note that the above discussion does not take into account the efficiency gains associated with passing references versus passing arguments by value. This can certainly be a concern (perhaps the over-riding concern), and should be properly considered when deciding whether to pass a non-`const` reference or not.

With the advent of template programming, the type-safe tuple became readily available. This allows for n-sized collections of un-related types to be passed as an object.

The STL implements a `pair` type which takes a pair of template arguments. As its name implies, this works well for pairs. In addition, boost offers an n-sized collection (currently written for up to 10 parameters, but easily extended to any size). The `pair` construct is primarily used in the `map` container, but may be used independently of `map`.

The `pair` class uses `first` and `second` functions to access the values it holds. The `tuple` class accesses its elements with the expression:

```
obj.get<N>()
```

or

```
get<N>(obj)
```

where `obj` is the tuple object and `N` is a constant integral corresponding to the index of the element in the tuple (base-0). This function is within the `boost::tuples` namespace.

`pair` is found in the `<utility>` header and the `std` namespace. `tuple` is found in the `boost/tuple/tuple.hpp` header and in the `boost` namespace.

The following is an example using both the `pair` and `tuple` types:

```
pair<int, string> ageAndName = getAgeAndName();
cout <<  ageAndName.second << " is age " << ageAndName.first << ".";

// Age = 33, Average Score = 96.7, Name = "Matthew D. Peavy"
boost::tuple<int, double, string> studentInfo;
get<0>(studentInfo) = 33;
get<1>(studentInfo) = 96.7;
get<2>(studentInfo) = "Matthew D. Peavy";
```

You may create tuples of `tuple`s and pairs of `pair`s, though nesting at this level becomes confusing and error prone. If you find yourself nesting tuples or pairs, consider using a `struct` or `class` instead.

4.15 How do I create and time and date stamp?

Time and date stamps are effective ways to order provide information and to name files. If the stamp is given in the format: yyyymmdd-hhmmss, then all files will be automatically sorted chronologically.

The following function returns a time/date stamp according to that format. Note that the headers `<ctime>, <sstream>, and <iomanip>` are required for this function.

```
   time_t now = time(0);
```

```
tm testTime = *localtime(&now);

//Year is only a 2 digit number, assuming we start at 1900.
//Month is base 0 (i.e., 0-11), so add 1
stringstream ss;
ss << testTime.tm_year +1900;     //Make a full 4 digit year

//The following two lines will fill in a 0 before a single digit
ss.fill('0');
ss.setf(ios::right, ios::adjustfield);
//The width must be set to 2 before each use of the inserter.
ss << setw(2) << testTime.tm_mon +1
   << setw(2) << testTime.tm_mday << "-"
   << setw(2) << testTime.tm_hour
   << setw(2) << testTime.tm_min
   << setw(2) << testTime.tm_sec;
```

4.16 How do I print the contents of a vector in one line?

You can print the contents of a `vector` to a stream in a single line by using the `copy` algorithm found in `<algorithm>`. Here we assume a `vector` of `string` objects being copied to `cout` using a single space delimeter between the objects:

```
copy(vec.begin(), vec.end(), ostream_iterator<string>(cout, " "));
```

Modify the template parameter type of the iterator, the output stream, and delimiter as necessary. Use `"\n"` to print the contents one-per-line, for example. Or use `ostream_iterator<int>` to print a `vector` of `int`s. In the sample code, this function is used to print the command-line arguments, which are parsed into a `vector` of `string` objects.

5 Bibliography

Meyers97: Scott Meyers, *Effective C++, Second Eddition*. Addison-Wesley, 1997.
Sutter00: Herb Sutter, *Exceptional C++*. Addison-Wesley, 2000.

Links:

FAQ Article	**Link**	**Library**
4.7	http://math.nist.gov/tnt/download.html	TNT
4.7	http://www.oonumerics.org	Various
4.11.1		
4.7	http://www.osl.iu.edu/research/mtl/	Matrix Template Library
4.1	http://www.boost.org/	lexical_cast
4.5		String Algorithms
4.6		Program_options
4.7		MultiArray, uBLAS
4.11.1		random
4.13		tuple
N/A	http://www.trumphurst.com/ cpplibs/cpplibs.phtml	A long list of available C++ libraries

6 Contributors

I would like to recognize and thank those who have contributed to this FAQ. If you have a contribution, please email me at faq@GiveMeFish.com. Include any contact information you would like included in this Contributors list.

Name	Contact	Contribution
Mark Jablin	jablin@pair.com	Various grammatical corrections.
Seungbeom Kim		Technical corrections to Ch. 1, 3, and 4.
Igor Mikushkin		Suggestion of alternative command line argument parsing function.