

# C, C++ programming tips

Vishal Patil

Summer 2003

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Managing multi-file C/C++ project</b>	<b>4</b>
2.1	Good software engineering practices . . . . .	4
<b>3</b>	<b>Using Automake and Autoconf</b>	<b>7</b>
3.1	Project directory structure . . . . .	7
3.2	Enabling Portability . . . . .	7
3.3	Configuration files in brief . . . . .	8
3.4	A brief example . . . . .	8
3.5	Understanding the configure.ac file . . . . .	8
3.6	Understanding the Makefile.am file . . . . .	9
3.6.1	For each program . . . . .	10
3.6.2	For each library . . . . .	11
<b>4</b>	<b>Generating build scripts</b>	<b>12</b>
<b>5</b>	<b>Build options</b>	<b>12</b>
<b>6</b>	<b>Avoiding common errors in C and C++</b>	<b>13</b>
6.1	Identifier clashes between source files . . . . .	13
6.2	Multiply defined symbols . . . . .	13
6.3	Redefinitions, redeclarations, conflicting types . . . . .	14
<b>7</b>	<b>Effective C++ programming</b>	<b>15</b>
7.1	Put the constant on the left in a conditional . . . . .	15
7.2	Handle errors and not bugs . . . . .	15
7.3	Use asserts in debug builds . . . . .	16

7.4	Use exceptions . . . . .	17
7.5	Virtual functions . . . . .	18
7.6	Don't ignore API function return values . . . . .	18
7.7	Be consistent . . . . .	18
7.8	Make your code const correct . . . . .	18
7.8.1	The many faces of const . . . . .	18
7.8.2	Understanding the const_cast operator . . . . .	19
7.8.3	const and data hiding . . . . .	20
<b>8</b>	<b>References</b>	<b>21</b>

# 1 Introduction

This document gives a quick idea about the various aspects of software development using C and C++. The document is in an adhoc form and lacks a proper structure and flow of information. However I shall be revising the structure from time to time and as I add new information to it.

NOTE: I HOLD NO RESPONSIBILITY FOR ANY DAMAGE OR FAILURE CAUSED BY USING THIS DOCUMENT. IT IS VERY LIKELY THAT THE CONTENTS OF THE DOCUMENT MIGHT CHANGE FROM TIME TO TIME AND MIGHT EVEN BE MISTAKEN AT SOME PLACES. USE THIS DOCUMENT AS PER YOUR OWN DISCRETION.

## 2 Managing multi-file C/C++ project

### 2.1 Good software engineering practices

The key to better software engineering is to focus away from developing monolithic applications that do only one job, and focus on developing libraries. One way to think of libraries is as a program with multiple entry points. Every library you write becomes a legacy that you can pass on to other developers. Just like in mathematics you develop little theorems and use the little theorems to hide the complexity in proving bigger theorems, in software engineering you develop libraries to take care of low-level details once and for all so that they are out of the way everytime you make a different implementation for a variation of the problem.

On a higher level you still don't create just one application. You create many little applications that work together. The centralized all-in-one approach in my experience is far less flexible than the decentralized approach in which a set of applications work together as a team to accomplish the goal. In fact this is the fundamental principle behind the design of the Unix operating system. Of course, it is still important to glue together the various components to do the job. This you can do either with scripting or with actually building a suite of specialized monolithic applications derived from the underlying tools.

The name of the game is like this: Break down the program to parts. And the parts to smaller parts, until you get down to simple subproblems that can be easily tested, and from which you can construct variations of the original problem. Implement each one of these as a library, write test code for each library and make sure that the library works. It is very important for your library to have a complete test suite, a collection of programs that are supposed to run silently and return normally (`exit(0);`) if they execute successfully, and return abnormally (`assert(false); exit(1);`) if they fail. The purpose of the test suite is to detect bugs in the library, and to convince you, the developer, that the library works. The best time to write a test program is as soon as it is possible! Don't be lazy. Don't just keep throwing in code after code after code. The minute there is enough new code in there to put together some kind of test program, just do it! I can not emphasize that enough. When you write new code you have the illusion that you are producing work, only to find out tomorrow that you need an entire week to debug it. As a rule, internalize the reality that you know you have produced new work everytime you write a working test program for the new features, and not a minute before. Another time when you should definitely write a

test suite is when you find a bug while ordinarily using the library. Then, before you even fix the bug, write a test program that detects the bug. Then go fix it. This way, as you add new features to your libraries you have insurance that they won't reawaken old bugs.

Please keep documentation up to date as you go. The best time to write documentation is right after you get a few new test programs working. You might feel that you are too busy to write documentation, but the truth of the matter is that you will always be too busy. After long hours debugging these seg faults, think of it as a celebration of triumph to fire up the editor and document your brand-spanking new cool features.

Please make sure that computational code is completely separated from I/O code so that someone else can reuse your computational code without being forced to also follow your I/O model. Then write programs that invoke your collection of libraries to solve various problems. By dividing and conquering the problem library by library with a test suite for each step along the way, you can write good and robust code. Also, if you are developing numerical software, please don't expect that other users of your code will be getting a high while entering data for your input files. Instead write an interactive utility that will allow users to configure input files in a user friendly way. Granted, this is too much work in Fortran. Then again, you do know more powerful languages, don't you?

Examples of useful libraries are things like linear algebra libraries, general ODE solvers, interpolation algorithms, and so on. As a result you end up with two packages. A package of libraries complete with a test suite, and a package of applications that invoke the libraries. The package of libraries is well-tested code that can be passed down to future developers. It is code that won't have to be rewritten if it's treated with respect. The package of applications is something that each developer will probably rewrite since different people will probably want to solve different problems. The effect of having a package of libraries is that C++ is elevated to a Very High Level Language that's closer to the problems you are solving. In fact a good rule of thumb is to make the libraries sufficiently sophisticated so that each executable that you produce can be expressed in one source file. All this may sound like common sense, but you will be surprised at how many scientific developers maintain just one does-everything-program that they perpetually hack until it becomes impossible to maintain. And then you will be even more surprised when you find that some professors don't understand why a "simple mathematical modification" of someone else's code is taking you so long.

Every library must have its own directory and Makefile. So a library

package will have many subdirectories, each directory being one library. And perhaps if you have too many of them, you might want to group them even further down. Then, there's the applications. If you've done everything right, there should be enough stuff in your libraries to enable you to have one source file per application. Which means that all the source files can probably go down under the same directory.

Very often you will come to a situation where there's something that your libraries to-date can't do, so you implement it and stick it along in your source file for the application. If you find yourself cut and pasting that implementation to other source files, then this means that you have to put this in a library somewhere. And if it doesn't belong to any library you've written so far, maybe to a new library. When you are in a deadline crunch, there's a tendency not to do this since it's easier to cut and paste. The problem is that if you don't take action right then, eventually your code will degenerate to a hard-to-use mess. Keeping the entropy down is something that must be done on a daily basis.

## 3 Using Automake and Autoconf

In this section I shall explain managing multi file C and C++ projects using `automake` and `autoconf` tools. These tools enable the developer to get rid of the tedium of writing complicated `Makefiles` for large projects and also avail portability across various platforms. These tools have been specifically designed for managing GNU projects. Software developed using `automake` and `autoconf` needs to adhere to the GNU software engineering principles.

### 3.1 Project directory structure

The project directory is recommended to have the following subdirectories and files.

- `src` : Contains the actual source code that gets compiled. Every library should have its own subdirectory. Every executable should have its own subdirectory as well. If each executable needs only one or two source files it's sensible to keep all the source files in the same directory.
- `lib` : An optional directory in which you place portability code like implementations of system calls that are not available on certain platforms.
- `doc` : Directory containing documentation for your package.
- `m4` : A directory containing 'm4' files that your package may need to install. These files define new 'autoconf' macros that you should make available to other developers who want to use your libraries.
- `intl` : Portability source code which allows your program to talk in various languages.
- `po` : Directory containing message catalogs for your software package.

Automake makes it really easy to manage multidirectory source code packages so you shouldn't be shy taking advantage of it.

### 3.2 Enabling Portability

In order to make your C/C++ project portable across different platforms you need to add the following lines of code to all of the source files before any include statements

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

The `config.h` file is generated by the tools (or the configure script not quite sure!!) and the `HAVE_CONFIG_H` flag is passed along with the `-D` option of the compiler by the generated scripts at the time of building the project. Use the `autoheader` utility to generate the `config.h` file for the project.

### 3.3 Configuration files in brief

There are two main configuration files used by these tools.

- `configure.ac` : Used by the `autoconf` tool to generate platform specific `configure` script.
- `Makefile.am` : Used by the `automake` tool to generate the `Makefile` for the sources present in the directory containing the `Makefile.am` file.

*NOTE : There exists a single `configure.ac` for a project however you need to create a distinct `Makefile.am` for each sub directory in the project*

### 3.4 A brief example

I shall explain the use of these tools with the help of a small sample project. The project will involve creation of an executable file called `main` (source `main.c`) which will call functions from a user generated library `geom` (source `circle.c`) and `stats` (source `mean.c`). The code for the main executable is present in the `src` directory while that for the `geom` and `stats` libraries is present inside subdirectories `geom` and `stats` respectively, inside the `src` directory. Thus this project will cover compiling of source files, creating static libraries and linking the libraries to create the final executable.

### 3.5 Understanding the `configure.ac` file

This file is used by the `autoconf` tool and used to generate the platform specific `configure` script.

The `configure.ac` file for the example is shown below.

```
AC_INIT(reconf)
AM_CONFIG_HEADER(config.h)
```

```

AM_INIT_AUTOMAKE(test,0.1)
AC_PROG_CC
AC_PROG_RANLIB
AC_PROG_INSTALL
AC_CONFIG_FILES([Makefile
                  doc/Makefile
                  m4/Makefile
                  src/Makefile
                  src/geom/Makefile
                  src/stats/Makefile
                  lib/Makefile
])
AC_OUTPUT

```

- In the above sample `configure.ac` the parameters passed to the `AM_INIT_AUTOMAKE` function namely `test` and `0.1` represent the package name and version number respectively.
- The `AC_CONFIG_FILES` function needs to be passed the paths of the various `Makefiles` which need to be generated for the various subdirectories. *Please note that `Makefile.am` must be present in each sub directory under the project directory*

### 3.6 Understanding the `Makefile.am` file

A `Makefile.am` is a set of assignments. These assignments imply the `Makefile`, a set of targets, dependencies and rules, and the `Makefile` implies the execution of building. The first set of assignments look like this

```

INCLUDES = -I/geom -I/stats ....
LDFLAGS = -L/geom -L/stats ....
LDADD = -lgeom -lgeom ...

```

- The `INCLUDES` assignment is where you insert the `-I` flags that you need to pass to your compiler. If the stuff in this directory is dependent on a library in another directory of the same package, then the `-I` flag must point to that directory.
- The `LDFLAGS` assignment is where you insert the `-L` flags that are needed by the compiler when it links all the object files to an executable.

- The LDADD assignment is where you list a long set of installed libraries that you want to link in with all of your executables. Use the -l flag only for installed libraries. You can list libraries that have been built but not installed yet as well, but do this only by providing the full path to these libraries.

If your package contains subdirectories with libraries and you want to link these libraries in another subdirectory you need to put ‘-I’ and ‘-L’ flags in the two variables above. To express the path to these other subdirectories, use the \$(top\_srcdir) variable. For example if you want to access a library under ‘src/libfoo’ you can put something like:

```
INCLUDES = ... -I$(top_srcdir)/src/geom ...
LD_FLAGS = ... -L$(top_srcdir)/src/geom ...
```

on the ‘Makefile.am’ of every directory level that wants access to these libraries. Also, you must make sure that the libraries are built before the directory level is built. To guarantee that, list the library directories in ‘SUBDIRS’ before the directory levels that depend on it. One way to do this is to put all the library directories under a ‘lib’ directory and all the executable directories under a ‘bin’ directory and on the ‘Makefile.am’ for the directory level that contains ‘lib’ and ‘bin’ list them as:

```
SUBDIRS = lib bin
```

### 3.6.1 For each program

You need to declare the set of files that are sources of the program, the set of libraries that must be linked with the program and (optionally) a set of dependencies that need to be built before the program is built. These are declared in assignments that look like this:

```
main_SOURCES = main.c
main_LDADD = -lgeom -lstats
main_LDFLAGS = -L$(top_srcdir)/src/geom -L$(top_srcdir)/src/stats
main_DEPENDENCIES = geom stats
```

- main\_SOURCES : Here you list all the ‘\*.cc’ and ‘\*.h’ files that compose the source code of the program. The presence of a header file here doesn’t cause the file to be installed at ‘/prefix/include’ but it does cause it to be added to the distribution when you do make dist. To cause header files to be installed you must also put them in ‘include\_HEADERS’.

- `main_LDADD` : Here you add primarily the `-l` flags for linking whatever libraries are needed by your code. You may also list object files, which have been compiled in an exotic way, as well as paths to uninstalled yet libraries.
- `main_LDFLAGS` : Here you add the `-L` flags that are needed to resolve the libraries you passed in `'main_LDADD'`. Certain flags that need to be passed on every program can be expressed on a global basis by assigning them at `'LDFLAGS'`.
- `main_DEPENDENCIES` : If for any reason you want certain other targets to be built before building this program, you can list them here.

### 3.6.2 For each library

There's a total of four assignments that are relevant to building libraries. Eg for the `lib geom` these assignments can be specified as follows.

```
lib_LIBRARIES = libgeom.a
libgeom_a_SOURCES = circle.c circle.h
libgeom_a_LIBADD = circle.o
libgeom_a_DEPENDENCIES =
```

- `lib_LIBRARIES` : The library name
- `libgeom_a_SOURCES` : Just like with programs, here you list all the `'*.cc'` files as well as all the private header files that compose the library. By private header file we mean a header file that is used internally by the library and the maintainers of the library, but is not exported to the end-user. You can list public header files also if you like, and perhaps you should for documentation purposes, but if you mention them in `include_HEADERS` it is not required to repeat them a second time here.
- `libgeom_a_LIBADD` : If there are any other object files that you want to include in the library list them here. You might be tempted to list them as dependencies in `'libgeom_a_DEPENDENCIES'`, but that will not work. If you do that, the object files will be built before the library is built but they will not be included in the library! By listing an object file here, you are stating that you want it to be built and you want it to be included in the library.

- `libgeom_a_DEPENDENCIES` : If there are any other targets that need to be built before this library is built, list them here.

## 4 Generating build scripts

The following commands have to be executed in order to generate the build scripts for the project.

1. `libtoolize`
2. `aclocal`
3. `autoheader`
4. `autoconf`
5. `touch README AUTHORS NEWS ChangeLog` (Required for GNU software adherence)
6. `automake -a`

The execution of the above four commands generates the `configure` in the top directory and `Makefile` scripts in the top directory as well as each of the sub directories.

## 5 Build options

You need to run the `configure` script before building the project using `make`. After successfully running the `configure` script the following options are available for `make`

- `make` Builds the project and creates the executables and libraries.
- `make clean` Cleans the project i.e removes all the executables.
- `make install` Builds and installs the project i.e the executable is copied in the `/prefix/bin`, headers in `/prefix/include` and libraries in `/prefix/lib` where `prefix` is usually `/usr/local`.
- `make uninstall` Uninstalls the project i.e removes the files added to `/prefix/bin`, `/prefix/include` and `/prefix/lib` directories.
- `make dist` Creates a distribution of the project (`<package-name>-<version>.tar.gz` file) of the project.

## 6 Avoiding common errors in C and C++

### 6.1 Identifier clashes between source files

In C, variables and functions are by default public, so that any C source file may refer to global variables and functions from another C source file. This is true even if the file in question does not have a declaration or prototype for the variable or function. You must, therefore, ensure that the same symbol name is not used in two different files. If you don't do this you will get linker errors and possibly warnings during compilation.

One way of doing this is to prefix public symbols with some string which depends on the source file they appear in. For example, all the routines in `gfx.c` might begin with the prefix `'gfx_'`. If you are careful with the way you split up your program, use sensible function names, and don't go overboard with global variables, this shouldn't be a problem anyway.

To prevent a symbol from being visible from outside the source file it is defined in, prefix its definition with the keyword `'static'`. This is useful for small functions which are used internally by a file, and won't be needed by any other file.

### 6.2 Multiply defined symbols

A header file is literally substituted into your C code in place of the `#include` statement. Consequently, if the header file is included in more than one source file all the definitions in the header file will occur in both source files. This causes them to be defined more than once, which gives a linker error (see above).

Solution: don't define variables in header files. You only want to declare them in the header file, and define them (once only) in the appropriate C source file, which should `#include` the header file of course for type checking. The distinction between a declaration and a definition is easy to miss for beginners; a declaration tells the compiler that the named symbol should exist and should have the specified type, but it does not cause the compiler to allocate storage space for it, while a definition does allocate the space. To make a declaration rather than a definition, put the keyword `'extern'` before the definition.

So, if we have an integer called `'counter'` which we want to be publicly

available, we would define it in a source file (one only) as ‘int counter;’ at top level, and declare it in a header file as ‘extern int counter;’.

### 6.3 Redefinitions, redelaratations, conflicting types

Consider what happens if a C source file includes both a.h and b.h, and also a.h includes b.h (which is perfectly sensible; b.h might define some types that a.h needs). Now, the C source file includes b.h twice. So every #define in b.h occurs twice, every declaration occurs twice (not actually a problem), every typedef occurs twice, etc. In theory, since they are exact duplicates it shouldn’t matter, but in practice it is not valid C and you will probably get compiler errors or at least warnings.

The solution to this problem is to ensure that the body of each header file is included only once per source file. This is generally achieved using preprocessor directives. We will define a macro for each header file, as we enter the header file, and only use the body of the file if the macro is not already defined. In practice it is as simple as putting this at the start of each header file:

```
#ifndef FILENAME_H
#define FILENAME_H
```

and then putting this at the end of it:

```
#endif
```

replacing FILENAME\_H with the (capitalised) filename of the header file, using an underline instead of a dot. Some people like to put a comment after the #endif to remind them what it is referring to, e.g.

```
#endif /* #ifndef FILENAME_H */
```

Personally I don’t do that since it’s usually pretty obvious, but it is a matter of style.

You only need to do this trick to header files that generate the compiler errors, but it doesn’t hurt to do it to all header files.

## 7 Effective C++ programming

### 7.1 Put the constant on the left in a conditional

We've all experienced bugs like this:

```
while (continue = TRUE)
{
    // ...this loops forever!
}
```

This type of problem can be solved by putting the constant on the left, so if you leave out an = in a conditional, you will get a compiler error instead of a program bug (because constants are non lvalues, of course):

```
while (TRUE = continue)
{
    // compile error!
}
```

### 7.2 Handle errors and not bugs

There are lots of error conditions that happen in the normal life of a program. For instance, file not found, out of memory, or invalid user input. You should always handle these conditions gracefully (by re-prompting for a filename, by freeing memory or telling the user to quit other applications, or by telling the user there is an error in his input, respectively). However, there are other conditions which are not real error conditions, but are the result of bugs. For example, say you have a routine which copies a string into a buffer, and no one is supposed to pass in a NULL pointer to the routine. You do not want to do something like this:

```
void CopyString(char* szBuffer, int nBufSize)
{
    if (NULL == szBuffer)
        return;          // quietly fail if NULL pointer
    else
    {
        strncpy(szBuffer, "Hello", nBufSize);
    }
}
```

Also, don't do something like this (some extremely fault-tolerant systems may be able to justify this, but 99% of applications can't):

```
void CopyString(char* szBuffer, int nBufSize)
{
    if (NULL == szBuffer)
    {
        cerr << "Error: NULL pointer passed to CopyString" << endl;
    }
    else
    {
        strncpy(szBuffer, "Hello", nBufSize);
    }
}
```

Instead, do something like this:

```
void CopyString(char* szBuffer, int nBufSize)
{
    assert(szBuffer); // complains and aborts in debug builds
    strncpy(szBuffer, "Hello", nBufSize);
};
```

In a release build, if the user passes in a NULL pointer, this will crash. But rather than think "I never want my application to crash, therefore I will test all pointers for NULL", think "I never want my applications to crash, therefore I will put in asserts and find bugs that result in NULL pointers being passed to routines, so I can fix them before I ship this software".

### 7.3 Use asserts in debug builds

Use asserts liberally in debug builds. You normally don't want to put assert code in release builds, because you don't want the user to see your bug messages. ANSI C (that's ISO for you purists) provides assertion functions in `assert.h` - if the symbol `NDEBUG` is defined somewhere before `assert.h` is included, then `assert()` will have no effect. Otherwise, it will print out a diagnostic message and abort the program if its argument evaluates to `FALSE`. Since `0 == FALSE`, you can use `assert()` on pointers to test them for non-NULL:

```
void myFunction(char* szFoo)
{
    assert(szFoo); // same as assert(NULL != szFoo);
}
```

## 7.4 Use exceptions

Use the try/throw/catch mechanisms in C++ - they are very powerful. Many people implement an exception class, which they use for general error reporting throughout their program.

```
class ProgramException
{
    // pass in a pointer to string, make sure string still exists when
    // the PrintError() method is called
    ProgramException(const char* const szErrorMsg = NULL)
    {
        if (NULL == szErrorMsg)
            m_szMsg = "Unspecified error";
        else
            m_szMsg = szErrorMsg;
    };

    void PrintError()
    {
        cerr << m_szMsg << endl;
    };
};

void OpenDataFile(const char* const szFileName)
{
    assert(szFileName);
    if (NULL == fopen(szFileName, "r"))
        throw ProgramException("File not found");
    // ...
}

int main(void)
{
    try
    {
        OpenDataFile("foo.dat");
    }
    catch (ProgramException e)
    {
        e.PrintError();
    }
}
```

```
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

## 7.5 Virtual functions

In C++ virtual functions are used for enabling Polymorphism. Following two important points need to be noted while using virtual functions:

- A function declared as virtual in the base class should be declared as virtual in the derived class as well.
- A class which has a member function declared as virtual needs to have its destructor to be defined as virtual as well. This is required for proper calls to the destructor up the class hierarchy.

## 7.6 Don't ignore API function return values

Most API functions will return a particular value which represents an error. You should test for these values every time you call the API function. If you don't want to clutter your code with error-testing, then wrap the API call in another function (do this when you are thinking about portability, too) which tests the return value and either asserts, handles the problem, or throws an exception. The above example of `OpenDataFile` is a primitive way of wrapping `fopen` with error-checking code which throws an exception if `fopen` fails.

## 7.7 Be consistent

Be consistent in the way you write your code. Use the same indentation and bracketing style everywhere. If you put the constant on the left in a conditional, do it everywhere. If you assert on your pointers, do it everywhere. Use the same kind of comment style for the same kind of comments. If you are the type to go in for a naming convention (like Hungarian notation), then you have to stick to it everywhere. Don't do `int iCount` in one place and `int nCount` in another.

## 7.8 Make your code const correct

### 7.8.1 The many faces of const

```
const int x;          // constant int
```

```

x = 2;                // illegal - can't modify x

const int* pX;       // changeable pointer to constant int
*pX = 3;            // illegal - can't use pX to modify an int
pX = &someOtherIntVar; // legal - pX can point somewhere else

int* const pY;       // constant pointer to changeable int
*pY = 4;            // legal - can use pY to modify an int
pY = &someOtherIntVar; // illegal - can't make pY point anywhere else

const int* const pZ; // const pointer to const int
*pZ = 5;            // illegal - can't use pZ to modify an int
pZ = &someOtherIntVar; // illegal - can't make pZ point anywhere else

```

The `const` keyword is more involved when used with pointers. A pointer is itself a variable which holds a memory address of another variable - it can be used as a "handle" to the variable whose address it holds. Note that there is a difference between "a read-only handle to a changeable variable" and a "changeable handle to a read-only variable".

## 7.8.2 Understanding the `const_cast` operator

```

const int x = 4;      // x is const, it can't be modified
const int* pX = &x;  // you can't modify x through the pX pointer

cout << x << endl;   // prints "4"

int* pX2 = const_cast < int* > (pX); // explicitly cast pX as non-const

*pX2 = 3;            // result is undefined
cout << x << endl;   // who knows what it prints?

```

The `const_cast` operator is more specific than normal type-casts because it can only be used to remove the `const`-ness of a variable, and trying to change its type in other ways is a compile error. For instance, say that you changed `x` in the above example to an `double` and changed `pX` to `double*`. However the variable `pX2` is casted as

```
int* pX2 = ( int* ) (pX); // explicitly cast pX as non-const
```

The code would still compile, but `pX2` would be treating it as an `int`. It might not cause a problem (because `ints` and `doubles` are somewhat similar), but

the code would certainly be confusing. Also, if you were using user-defined classes instead of numeric types, the code would still compile, but it would almost certainly crash your program. If you use `const_cast`, you can be sure that the compiler will only let you change the `const`-ness of a variable, and never its type.

### 7.8.3 `const` and data hiding

In C++ when a member function returns a pointer which points to its member variable, there exists a possibility of the pointer address or the pointer value getting modified. This problem can be overcome by using the `const` keyword. An example illustrating this idea is given below

```
class Person
{
public:
    Person(char* szNewName)
    {
        // make a copy of the string
        m_szName = _strdup(szNewName);
    };

    ~Person() { delete[] m_szName; };

    const char* const GetName() const
    {
        return m_szName;
    };

private:
    char* m_szName;
};
```

In the above class the `GetName()` member function returns a pointer to the member variable `m_szName`. To prevent this member variable from getting accidentally modified the `GetName()` has been prototyped to return a constant pointer pointing to a constant value. Also the `const` keyword at the end of the function prototype states that the function does not modify any of the member variables.

## 8 References

- <http://www.gmonline.demon.co.uk/cscene/CS2/CS2-01.html>
- <http://autotoolset.sourceforge.net/tutorial.html>