Programming Design Guide

guide-software

Rev: 1

Cem Eden

2018–11–13

Table of Contents

1 C++

1.1 General Formatting

1.1.1 Preprocessor Directives

Preprocessor directives are commands executed at compile time. They can be identified by a '#' sign, followed by command name and any arguments.

Preprocessor directives should be formatted so that the commands are aligned to the left most side of the page with no leading tabs or spaces.

The command itself should be typed in all lowercase.

Define arguments should be typed in all uppercase and any spaces should be substituted with underscores.

Example:

```
#define FOO_BAR
#ifndef FOO_BAR
        //code
#endif
```

1.1.2 Tabs And Spaces

When formatting code, alignment is an important factor for readability.

When aligning code, the tabs should be used for the line of code. If a command is too long, or is better readable when separated into multiple lines, subsequent lines should be aligned with spaces.

Example:

```
void exampleFunction()
{
        otherFunct(somevar); // indented with tab

        diffFunct(var1, // single tab
                  var2, // single tab with 10 spaces
                  var3  // single tab with 10 spaces
                  );    // single tab with 10 spaces

        sFunc(var1, // single tab
              var2  // single tab with 6 spaces
              );    // single tab with 6 spaces
}
```

1.1.3 Brace Style

In cases where braces are used, they should be laid out in a way where it is easy to tell where a scope starts and ends.

A starting brace should always be placed on the next line, aligned with the statement that requires it.

For statements such as else statements, the ending brace as well as the starting brace should be on separate lines.

The only exception to the rule is for statements following an end brace, which do not have a starting brace. For example, a do-while loop.

Example:

```
void exampleFunction()
{
        if(condition)
        {
                // code if true
        }
        else
        {
                // code if false
        }

        while(condition)
        {
                // loop code
        }

        do
        {
                // loop code
        }while(condition);
}
```

1.1.4 Switch-Case Statements

Switch-Case statements are special in their way of using a separate keyword, namely case, which is placed within its scope. Due to this they have their own syntax.

The Switch statement should be treated as any other conditional along with its brace style. However the Case statements have some special rules:

The Case statements should be aligned with the indentation of the Switch statement, and not the current scope. Values that a case statement stands for, should be placed in parentheses.

The statements such as break or return however, should be aligned with the current scope.

Example:

```
switch(enumerable)
{
case(1):
        // some code
        break;

case(2):
        // other code
        break;

default:
        // default code
}
```

1.1.5 Return Statements

Return statements that should return a value, should have their return value in prentheses.

Example:

```
int main()
{
        return(0);
}
```

1.2 Header Files

1.2.1 Guards

To make sure that code can be included easily, each header file should include a #define guard.

This is to ensure that if other files include the header, that no double definition errors occur.

The #define guard should be placed immediately at the beginning of the file, or if any comments are added, on the first non-commented line.
The guard itself consists of a #ifndef statement followed immediately by a #define statement where both have a defined constant adhering to the formatting rules (Preprocessor Directives, subsection 1.1.1). On the last line of the file, the #ifndef statement is then completed with a #endif statement.

The name of the constant should be given to be descriptive of the contents of the header file but should not coincide with other #define definitions. In this case a longer variable name may be better than a shorter one.

Example:

```
// some comment
#ifndef HEADER_FILE_GUARD_NAME
#define HEADER_FILE_GUARD_NAME

// header file content

#endif
```

1.2.2 Class Definitions

Only one main class should be defined per header file. Only small complimentary classes or data structures should be included alongside the main class.

The main class should closely match the header file name but does not have to be exactly the same. The name should however be descriptive.

Access modifiers should be aligned with the scope of the class, and not the current scope. In general, this means that access modifiers should not have any leading tabs or spaces.

Any functions defined in a class should be as protected as possible. This means that if a function should only be accessible within a class, it should be set to private.

Where possible, classes should start with its public members, and should be seperated with as little duplicate sections as possible.

All variables within a class should ALWAYS be set to private. If these variables are to be modified from outside of the class, this has to be done using getter and setter functions.

Friend inheritance should be avoided as much as possible. If a friend relationship is sufficiently justified, then they must be sufficiently documented using comments at the declaration and definition. This documentation must include what other functions use this relationship.

Constructors and destructors should be placed as close to the start of the class declaration as possible. Constructors should be group together.

Overloaded functions should be placed within the same access block and should be grouped together. They should be sequenced such that functions with the least arguments are at the top of the group, and functions with the most arguments should be at the bottom of that group.

New lines should be used to signify grouping.

Function declarations should have the same name and type of arguments as their definitions. The only exceptions to this are default values and namespaces.

Example:

```
class someClass
{
public:
        someClass();
        ~someClasss();

        int someFunc(int arg1);

        void var1Set(int val);
        int var1Get();

private:
        int someInternalFunc(int arg1);

        int var1;
        int var2;
};
```

1.2.3 What belongs in the Header

The headers focus should be to tell what functions are contained within it. It should let a programmer know of any #define constants that are to be used with given public functions and should serve as an overview on what functions can be used.

The header should however not contain any complex implementations. This means, unless the header defines a very simple class, like for instance a linked list class, only minimal definitions should be included.

Example:

```
#ifndef HEADER_GUARD
#define HEADER_GUARD
class someClass
{
public:
        int someFunc(int arg1);
private:
        int someInternalFunc(int arg1);
};
#endif
```

If, however the class defined is very simple, definitions can be placed into the header. However, doing so should be considered carefully, as header implemented classes can be difficult to read.

Example:

```
// overcomplicated isOdd class
#ifndef HEADER_GUARD
#define HEADER_GUARD
class someClass
{
private
        bool someInternalFunc(int arg1){return(arg1%2);};
public:
        bool someFunc(int arg1){return(someInternalFunc(arg1));};
}
#endif
```

1.2.4 Structs and Classes

In C++ structs and classes are very similar, however they should be used for specific tasks.

Structs should be predominantly for data structures, and contain little to no functions.

Structs do not necessarily need any access modifiers as most members will be public.

Classes should be used for almost every other case.

Example:

```
struct someDatastructure
{
        int dataID;
        int someData;
        someDatastructure* next;
};

class someClass
{
public:
        int getdata(int ID);

private:
        someDatastructure data;
        char otherData;
};
```

1.3 Source Files

1.3.1 Include Directives

Include directives should be placed in the source file whenever possible. This is to ensure that headers are not needlessly included when trying to use a specific class.

The only time that Include directives can be places in the header, is if a function definition requires.

1.3.2 Using Namespace Directives

The using namespace directive is a great convenience tool, but at the same time it may cause issues.

To avoid ambiguity with identically named functions, it is vital to place the using namespace directives within source files ONLY. This is to ensure that when another class uses a header file, that they have no namespaces already defined.

Example:
Header File:

```
#include <string>

class someClass
{
public:
        std::string stringStuff(std::string stringArg);
}
```

Source File:

```
using namespace std;

string someClass::stringStuff(string stringArg)
{
        // implementation
}
```

1.3.3 Class Definitions

Classes should be declared in their corresponding header files. An exception to this rule is if a data structure is used, and is only relevant internally.

When defining a class function, everything up to the opening brace should be in a single line. If a new line should be added in between, then this should be done between arguments.

The order in which class functions should be defined should match the order in which they were declared.