Bjarne Stroustrup's *The C++ Programming Language* has a chapter titled "A Tour of C++: The Basics"—Standard C++. That chapter, in 2.2, mentions in half a page the compilation and linking process in C++. Compilation and linking are two very basic processes that happen all the time during C++ software development, but oddly enough, they aren't well understood by many C++ developers.

Why is C++ source code split into header and source files? How is each part seen by the compiler? How does that affect compilation and linking? There are many more questions like these that you may have thought about but have come to accept as convention.

Whether you are designing a C++ application, implementing new features for it, trying to address bugs (especially certain strange bugs), or trying to make C and C++ code work together, knowing how compilation and linking works will save you a lot of time and make those tasks much more pleasant. In this article, you will learn exactly that.

The article will explain how a C++ compiler works with some of the basic language constructs, answer some common questions that are related to their processes, and help you work around some related mistakes that developers often make in C++ development.

Note: This article has some example source code that can be downloaded from https://bitbucket.org/danielmunoz/cpp-article

The examples were compiled in a CentOS Linux machine:

```
$ uname -sr
Linux 3.10.0-327.36.3.el7.x86_64
```

Using g++ version:

```
$ g++ --version
g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-11)
```

The source files provided should be portable to other operating systems, although the Makefiles accompanying them for the automated build process should be portable only to Unix-like systems.

The Build Pipeline: Preprocess, Compile, and Link

Each C++ source file needs to be compiled into an object file. The object files resulting from the compilation of multiple source files are then linked into an executable, a shared library, or a static library (the last of these being just an archive of object files). C++ source files generally have the .cpp, .cxx or .cc extension suffixes.

A C++ source file can include other files, known as header files, with the `#include` directive. Header files have extensions like .h, .hpp, or .hxx, or have no extension at all like in the C++ standard library and other libraries' header files (like Qt). The extension doesn't matter for the C++ preprocessor,

which will literally replace the line containing the `#include` directive with the entire content of the included file.

The first step that the compiler will do on a source file is run the preprocessor on it. Only source files are passed to the compiler (to preprocess and compile it). Header files aren't passed to the compiler. Instead, they are included from source files.

Each header file can be opened multiple times during the preprocessing phase of all source files, depending on how many source files include them, or how many other header files that are included from source files also include them (there can be many levels of indirection). Source files, on the other hand, are opened only once by the compiler (and preprocessor), when they are passed to it.

For each C++ source file, the preprocessor will build a translation unit by inserting content in it when it finds an #include directive at the same time that it'll be stripping code out of the source file and of the headers when it finds conditional compilation blocks whose directive evaluates to `false`. It'll also do some other tasks like macro replacements.

Once the preprocessor finishes creating that (sometimes huge) translation unit, the compiler starts the compilation phase and produces the object file.

To obtain that translation unit (the preprocessed source code), the `-E` option can be passed to the g++ compiler, along with the `-o` option to specify the desired name of the preprocessed source file.

In the `cpp-article/hello-world` directory, there is a "hello-world.cpp" example file:

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "Hello world" << std::endl;
    return 0;
}
```

Create the preprocessed file by:

```
$ g++ -E hello-world.cpp -o hello-world.ii
```

And see the number of lines:

```
$ wc -l hello-world.ii
17558 hello-world.ii
```

It has 17,588 lines in my machine. You can also just run `make` on that directory and it'll do those steps for you.

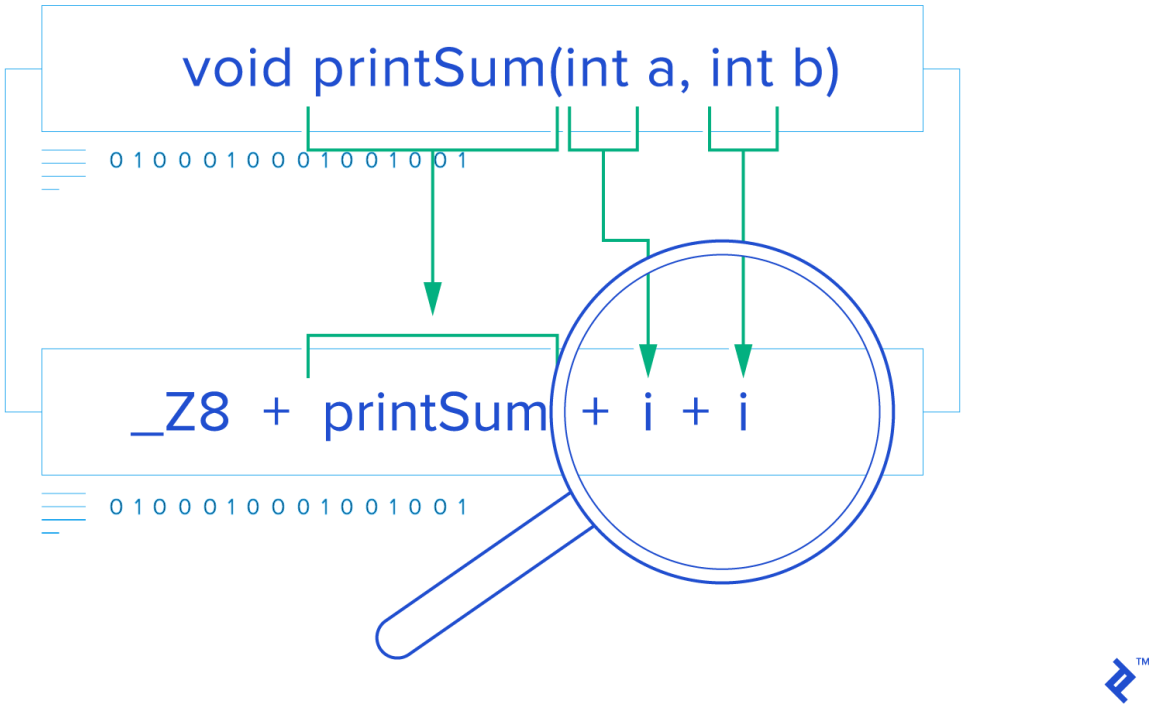
We can see that the compiler must compile a much larger file than the simple source file that we see. This is because of the included headers. And in our

example, we have included just one header. The translation unit becomes bigger and bigger as we keep including headers.

This preprocess and compile process is similar for C language. It follows the C rules for compiling, and the way it includes header files and produces object code is nearly the same.

How Source Files Import and Export Symbols

Let's see now the files in `cpp-article/symbols/c-vs-cpp-names` directory.



There is a simple C (not C++) source file named sum.c that exports two functions, one for adding two integers and one for adding two floats:

```
int sumI(int a, int b) {
    return a + b;
}

float sumF(float a, float b) {
    return a + b;
}
```

Compile it (or run `make` and all the steps to create the two example apps to be executed) to create the sum.o object file:

```
$ gcc -c sum.c
```

Now look at the symbols exported and imported by this object file:

```
$ nm sum.o
0000000000000014 T sumF
0000000000000000 T sumI
```

No symbols are imported and two symbols are exported: `sumF` and `sumI`. Those symbols are exported as part of the .text segment (T), so they are function names, executable code.

If other (both C or C++) source files want to call those functions, they need to declare them before calling.

The standard way to do it is to create a header file that declares them and includes them in whatever source file we want to call them. The header can have any name and extension. I chose `sum.h`:

```
#ifdef __cplusplus
extern "C" {
#endif

int sumI(int a, int b);
float sumF(float a, float b);

#ifdef __cplusplus
} // end extern "C"
#endif
```

What are those `ifdef/endif` conditional compilation blocks? If I include this header from a C source file, I want it to become:

```
int sumI(int a, int b);
float sumF(float a, float b);
```

But if I include them from a C++ source file, I want it to become:

```
extern "C" {

int sumI(int a, int b);
float sumF(float a, float b);

} // end extern "C"
```

C language doesn't know anything about the `extern "C"` directive, but C++ does, and it needs this directive applied to C function declarations. This is because C++ mangles function (and method) names because it supports function/method overloading, while C doesn't.

This can be seen in the C++ source file named print.cpp:

```
#include <iostream> // std::cout, std::endl
#include "sum.h" // sumI, sumF

void printSum(int a, int b) {
    std::cout << a << " + " << b << " = " << sumI(a, b) << std::endl;
}

void printSum(float a, float b) {
```

```
    std::cout << a << " + " << b << " = " << sumF(a, b) << std::endl;
}

extern "C" void printSumInt(int a, int b) {
    printSum(a, b);
}

extern "C" void printSumFloat(float a, float b) {
    printSum(a, b);
}
```

There are two functions with the same name (`printSum`) that only differ in their parameters' type: `int` or `float`. Function overloading is a C++ feature which isn't present in C. To implement this feature and differentiate those functions, C++ mangles the function name, as we can see in their exported symbol name (I'll only pick what's relevant from nm's output):

```
$ g++ -c print.cpp
$ nm print.o
0000000000000132 T printSumFloat
0000000000000113 T printSumInt
                 U sumF
                 U sumI
0000000000000074 T _Z8printSumff
0000000000000000 T _Z8printSumii
                 U _ZSt4cout
```

Those functions are exported (in my system) as `_Z8printSumff` for the float version and `_Z8printSumii` for the int version. Every function name in C++ is mangled unless declared as `extern "C"`. There are two functions that were declared with C linkage in `print.cpp`: `printSumInt` and `printSumFloat`.

Therefore, they cannot be overloaded, or their exported names would be the same since they aren't mangled. I had to differentiate them from each other by postfixing an Int or a Float to the end of their names.

Since they are not mangled they can be called from C code, as we'll soon see.

To see the mangled names like we would see them in C++ source code, we can use the `-C` (demangle) option in the `nm` command. Again, I'll only copy the same relevant part of the output:

```
$ nm -C print.o
0000000000000132 T printSumFloat
0000000000000113 T printSumInt
                 U sumF
                 U sumI
0000000000000074 T printSum(float, float)
0000000000000000 T printSum(int, int)
                 U std::cout
```

With this option, instead of `_Z8printSumff` we see `printSum(float, float)`, and instead of `_ZSt4cout` we see std::cout, which are more human-friendly names.

We also see that our C++ code is calling C code: `print.cpp` is calling `sumI` and `sumF`, which are C functions declared as having C linkage in `sum.h`. This can be seen in the nm output of print.o above, that informs of some undefined (U) symbols: `sumF`, `sumI` and `std::cout`. Those undefined symbols are supposed to be provided in one of the object files (or libraries) that will be linked together with this object file output in the link phase.

So far we have just compiled source code into object code, we haven't yet linked. If we don't link the object file that contains the definitions for those imported symbols together with this object file, the linker will stop with a "missing symbol" error.

Note also that since `print.cpp` is a C++ source file, compiled with a C++ compiler (g++), all the code in it is compiled as C++ code. Functions with C linkage like `printSumInt` and `printSumFloat` are also C++ functions that can use C++ features. Only the names of the symbols are compatible with C, but the code is C++, which can be seen by the fact that both functions are calling an overloaded function (`printSum`), which couldn't happen if `printSumInt` or `printSumFloat` were compiled in C.

Let's see now `print.hpp`, a header file that can be included both from C or C++ source files, which will allow `printSumInt` and `printSumFloat` to be called both from C and from C++, and `printSum` to be called from C++:

```
#ifdef __cplusplus
void printSum(int a, int b);
void printSum(float a, float b);
extern "C" {
#endif

void printSumInt(int a, int b);
void printSumFloat(float a, float b);

#ifdef __cplusplus
} // end extern "C"
#endif
```

If we are including it from a C source file, we just want to see:

```
void printSumInt(int a, int b);
void printSumFloat(float a, float b);
```

`printSum` can't be seen from C code since its name is mangled, so we don't have a (standard and portable) way to declare it for C code. Yes, I can declare them as:

```
void _Z8printSumii(int a, int b);
void _Z8printSumff(float a, float b);
```

And the linker won't complain since that's the exact name that my currently installed compiler invented for it, but I don't know if it'll work for your linker (if your compiler generates a different mangled name), or even for the next

version of my linker. I don't even know if the call will work as expected because of the existence of different calling conventions (how parameters are passed and return values are returned) that are compiler specific and may be different for C and C++ calls (especially for C++ functions that are member functions and receive the this pointer as a parameter).

Your compiler can potentially use one calling convention for regular C++ functions and a different one if they are declared as having extern "C" linkage. So, cheating the compiler by saying that one function uses C calling convention while it actually uses C++ for it can deliver unexpected results if the conventions used for each happen to be different in your compiling toolchain.

There are standard ways to mix C and C++ code and a standard way to call C++ overloaded functions from C is to wrap them in functions with C linkage as we did by wrapping `printSum` with `printSumInt` and `printSumFloat`.

If we include `print.hpp` from a C++ source file, the `__cplusplus` preprocessor macro will be defined and the file will be seen as:

```
void printSum(int a, int b);
void printSum(float a, float b);
extern "C" {

void printSumInt(int a, int b);
void printSumFloat(float a, float b);

} // end extern "C"
```

This will allow C++ code to call the overloaded function printSum or its wrappers `printSumInt` and `printSumFloat`.

Now let's create a C source file containing the main function, which is the entry point for a program. This C main function will call `printSumInt` and `printSumFloat`, that is, will call both C++ functions with C linkage. Remember, those are C++ functions (their function bodies execute C++ code) that only don't have C++ mangled names. The file is named `c-main.c`:

```
#include "print.hpp"

int main(int argc, char* argv[]) {
    printSumInt(1, 2);
    printSumFloat(1.5f, 2.5f);
    return 0;
}
```

Compile it to generate the object file:

```
$ gcc -c c-main.c
```

And see the imported/exported symbols:

```
$ nm c-main.o
0000000000000000 T main
                 U printSumFloat
                 U printSumInt
```

It exports main and imports `printSumFloat` and `printSumInt`, as expected.

To link it all together into an executable file, we need to use the C++ linker (g++), since at least one file that we'll link, `print.o`, was compiled in C++:

```
$ g++ -o c-app sum.o print.o c-main.o
```

The execution produces the expected result:

```
$ ./c-app
1 + 2 = 3
1.5 + 2.5 = 4
```

Now let's try with a C++ main file, named `cpp-main.cpp`:

```
#include "print.hpp"

int main(int argc, char* argv[]) {
    printSum(1, 2);
    printSum(1.5f, 2.5f);
    printSumInt(3, 4);
    printSumFloat(3.5f, 4.5f);
    return 0;
}
```

Compile and see the imported/exported symbols of the `cpp-main.o` object file:

```
$ g++ -c cpp-main.cpp
$ nm -C cpp-main.o
0000000000000000 T main
                 U printSumFloat
                 U printSumInt
                 U printSum(float, float)
                 U printSum(int, int)
```

It exports main and imports C linkage `printSumFloat` and `printSumInt`, and both mangled versions of `printSum`.

You may be wondering why the main symbol isn't exported as a mangled symbol like `main(int, char**)` from this C++ source since it's a C++ source file and it isn't defined as `extern "C"`. Well, `main` is a special implementation defined function and my implementation seems to have chosen to use C linkage for it no matter whether it's defined in a C or C++ source file.

Linking and running the program gives the expected result:

```
$ g++ -o cpp-app sum.o print.o cpp-main.o
$ ./cpp-app
1 + 2 = 3
```

```
1.5 + 2.5 = 4
3 + 4 = 7
3.5 + 4.5 = 8
```

How Header Guards Work

So far, I've been careful not to include my headers twice, directly or indirectly, from the same source file. But since one header can include other headers, the same header can indirectly be included multiple times. And since header content is just inserted in the place from where it was included, it's easy to end with duplicated declarations.

See the example files in `cpp-article/header-guards`.

```
// unguarded.hpp
class A {
public:
    A(int a) : m_a(a) {}
    void setA(int a) { m_a = a; }
    int getA() const { return m_a; }
private:
    int m_a;
};

// guarded.hpp:
#ifndef __GUARDED_HPP
#define __GUARDED_HPP

class A {
public:
    A(int a) : m_a(a) {}
    void setA(int a) { m_a = a; }
    int getA() const { return m_a; }
private:
    int m_a;
};

#endif // __GUARDED_HPP
```

The difference is that, in guarded.hpp, we surround the entire header in a conditional that will only be included if `__GUARDED_HPP` preprocessor macro isn't defined. The first time that the preprocessor includes this file, it won't be defined. But, since the macro is defined inside that guarded code, the next time it's included (from the same source file, directly or indirectly), the preprocessor will see the lines between the #ifndef and the #endif and will discard all the code between them.

Note that this process happens for every source file that we compile. It means that this header file can be included once and only once for each source file. The fact that it was included from one source file won't prevent it to be included from a different source file when that source file is compiled. It'll just prevent it to be included more than once from the same source file.

The example file `main-guarded.cpp` includes `guarded.hpp` twice:

```
#include "guarded.hpp"
#include "guarded.hpp"

int main(int argc, char* argv[]) {
    A a(5);
    a.setA(0);
    return a.getA();
}
```

But the preprocessed output only shows one definition of class `A`:

```
$ g++ -E main-guarded.cpp
# 1 "main-guarded.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main-guarded.cpp"
# 1 "guarded.hpp" 1



class A {
public:
    A(int a) : m_a(a) {}
    void setA(int a) { m_a = a; }
    int getA() const { return m_a; }
private:
    int m_a;
};
# 2 "main-guarded.cpp" 2


int main(int argc, char* argv[]) {
    A a(5);
    a.setA(0);
    return a.getA();
}
```

Therefore, it can be compiled without problems:

```
$ g++ -o guarded main-guarded.cpp
```

But the `main-unguarded.cpp` file includes `unguarded.hpp` twice:

```
#include "unguarded.hpp"
#include "unguarded.hpp"

int main(int argc, char* argv[]) {
    A a(5);
    a.setA(0);
```

```
    return a.getA();
}
```

And the preprocessed output shows two definitions of class A:

```
$ g++ -E main-unguarded.cpp
# 1 "main-unguarded.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main-unguarded.cpp"
# 1 "unguarded.hpp" 1
class A {
public:
    A(int a) : m_a(a) {}
    void setA(int a) { m_a = a; }
    int getA() const { return m_a; }
private:
    int m_a;
};
# 2 "main-unguarded.cpp" 2
# 1 "unguarded.hpp" 1
class A {
public:
    A(int a) : m_a(a) {}
    void setA(int a) { m_a = a; }
    int getA() const { return m_a; }
private:
    int m_a;
};
# 3 "main-unguarded.cpp" 2

int main(int argc, char* argv[]) {
    A a(5);
    a.setA(0);
    return a.getA();
}
```

This will cause problems when compiling:

```
$ g++ -o unguarded main-unguarded.cpp
```

In file included from `main-unguarded.cpp:2:0`:

```
unguarded.hpp:1:7: error: redefinition of 'class A'
 class A {
       ^
In file included from main-unguarded.cpp:1:0:
unguarded.hpp:1:7: error: previous definition of 'class A'
 class A {
       ^
```

For the sake of brevity, I won't use guarded headers in this article if it isn't necessary since most are short examples. But always guard your header files.

Not your source files, which won't be included from anywhere. Just header files.

Pass by Value and Constness of Parameters

Look at `by-value.cpp` file in `cpp-article/symbols/pass-by`:

```
#include <vector>
#include <numeric>
#include <iostream>

// std::vector, std::accumulate, std::cout, std::endl
using namespace std;

int sum(int a, const int b) {
    cout << "sum(int, const int)" << endl;
    const int c = a + b;
    ++a; // Possible, not const
    // ++b; // Not possible, this would result in a compilation error
    return c;
}

float sum(const float a, float b) {
    cout << "sum(const float, float)" << endl;
    return a + b;
}

int sum(vector<int> v) {
    cout << "sum(vector<int>)" << endl;
    return accumulate(v.begin(), v.end(), 0);
}

float sum(const vector<float> v) {
    cout << "sum(const vector<float>)" << endl;
    return accumulate(v.begin(), v.end(), 0.0f);
}
```

Since I use the `using namespace std` directive, I don't have to qualify the names of symbols (functions or classes) inside the std namespace in the rest of the translation unit, which in my case is the rest of the source file. If this were a header file, I shouldn't have inserted this directive because a header file is supposed to be included from multiple source files; this directive would bring to the global scope of each source file the entire std namespace from the point they include my header.

Even headers included after mine in those files will have those symbols in scope. This can produce name clashes since they were not expecting this to happen. Therefore, don't use this directive in headers. Only use it in source files if you want, and only after you included all headers.

Note how some parameters are const. This means that they can't be changed in the body of the function if we try to. It'd give a compilation error. Also, note that all the parameters in this source file are passed by value, not by reference (&) or by pointer (*). This means that the caller will make a copy of

them and pass to the function. So, it doesn’t matter for the caller whether they are const or not, because if we modify them in the function body we’ll only be modifying the copy, not the original value the caller passed to the function.

Since the constness of a parameter that is passed by value (copy) doesn’t matter for the caller, it is not mangled in the function signature, as it can be seen after compiling and inspecting the object code (only the relevant output):

```
$ g++ -c by-value.cpp
$ nm -C by-value.o
000000000000001e T sum(float, float)
0000000000000000 T sum(int, int)
0000000000000087 T sum(std::vector<float, std::allocator<float> >)
0000000000000048 T sum(std::vector<int, std::allocator<int> >)
```

The signatures don’t express whether the copied parameters are const or not in the bodies of the function. It doesn’t matter. It mattered for the function definition only, to show at a glance to the reader of the function body whether those values will ever change. In the example, only half of the parameters are declared as const, so we can see the contrast, but if we want to be const-correct they should all have been declared so since none of them are modified in the function body (and they shouldn’t).

Since it doesn’t matter for the function declaration which is what the caller sees, we can create the `by-value.hpp` header like this:

```
#include <vector>

int sum(int a, int b);
float sum(float a, float b);
int sum(std::vector<int> v);
int sum(std::vector<float> v);
```

Adding the const qualifiers here is allowed (you can even qualify as const variables that aren’t const in the definition and it’ll work), but this is not necessary and it’ll only make the declarations unnecessarily verbose.

Pass by Reference

Let's see `by-reference.cpp`:

```
#include <vector>
#include <iostream>
#include <numeric>

using namespace std;

int sum(const int& a, int& b) {
    cout << "sum(const int&, int&)" << endl;
    const int c = a + b;
    ++b; // Will modify caller variable
    // ++a; // Not allowed, but would also modify caller variable
    return c;
}

float sum(float& a, const float& b) {
    cout << "sum(float&, const float&)" << endl;
    return a + b;
}

int sum(const std::vector<int>& v) {
    cout << "sum(const std::vector<int>&)" << endl;
    return accumulate(v.begin(), v.end(), 0);
}

float sum(const std::vector<float>& v) {
    cout << "sum(const std::vector<float>&)" << endl;
    return accumulate(v.begin(), v.end(), 0.0f);
}
```

Constness when passing by reference matters for the caller, because it'll tell the caller whether its argument will be modified or not by the callee. Therefore, the symbols are exported with their constness:

```
$ g++ -c by-reference.cpp
$ nm -C by-reference.o
0000000000000051 T sum(float&, float const&)
0000000000000000 T sum(int const&, int&)
00000000000000fe T sum(std::vector<float, std::allocator<float> > const&)
00000000000000a3 T sum(std::vector<int, std::allocator<int> > const&)
```

That should also be reflected in the header that callers will use:

```
#include <vector>

int sum(const int&, int&);
float sum(float&, const float&);
```

```
int sum(const std::vector<int>&);
float sum(const std::vector<float>&);
```

Note that I didn't write the name of the variables in the declarations (in the header) as I'd been doing so far. This is also legal, for this example and for the previous ones. Variable names aren't required in the declaration, since the caller doesn't need to know how do you want to name your variable. But parameter names are generally desirable in declarations so the user can know at a glance what each parameter mean and therefore what to send in the call.

Surprisingly, variable names aren't either needed in the definition of a function. They are only needed if you actually use the parameter in the function. But if you never use it you can leave the parameter with the type but without the name. Why would a function declare a parameter that it'd never use? Sometimes functions (or methods) are just part of an interface, like a callback interface, which defines certain parameters that are passed to the observer. The observer must create a callback with all the parameters that the interface specifies since they'll be all sent by the caller. But the observer may not be interested in all of them, so instead of receiving a compiler warning about an "unused parameter," the function definition can just leave it without a name.

Pass by Pointer

```
// by-pointer.cpp:
#include <iostream>
#include <vector>
#include <numeric>

using namespace std;

int sum(int const * a, int const * const b) {
    cout << "sum(int const *, int const * const)" << endl;
    const int c = *a+ *b;
    // *a = 4; // Can't change. The value pointed to is const.
    // *b = 4; // Can't change. The value pointed to is const.
    a = b; // I can make a point to another const int
    // b = a; // Can't change where b points because the pointer itself
    return c;
}

float sum(float * const a, float * b) {
    cout << "sum(int const * const, float const *)" << endl;
    return *a + *b;
}

int sum(const std::vector<int>* v) {
    cout << "sum(std::vector<int> const *)" << endl;
    // v->clear(); // I can't modify the const object pointed by v
    const int c = accumulate(v->begin(), v->end(), 0);
    v = NULL; // I can make v point to somewhere else
    return c;
}
```

```
float sum(const std::vector<float> * const v) {
    cout << "sum(std::vector<float> const * const)" << endl;
    // v->clear(); // I can't modify the const object pointed by v
    // v = NULL; // I can't modify where the pointer points to
    return accumulate(v->begin(), v->end(), 0.0f);
}
```

To declare a pointer to a const element (int in the example) you can declare the type as either of:

```
int const *
const int *
```

If you also want the pointer itself to be const, that is, that the pointer cannot be changed to point to something else, you add a const after the star:

```
int const * const
const int * const
```

If you want the pointer itself to be const, but not the element pointed by it:

```
int * const
```

Compare the function signatures with the demangled inspection of the object file:

```
$ g++ -c by-pointer.cpp
$ nm -C by-pointer.o
000000000000004a T sum(float*, float*)
0000000000000000 T sum(int const*, int const*)
0000000000000105 T sum(std::vector<float, std::allocator<float> > const*)
000000000000009c T sum(std::vector<int, std::allocator<int> > const*)
```

As you see, the `nm` tool uses the first notation (const after the type). Also, note that the only constness that is exported, and matters for the caller, is whether the function will modify the element pointed by the pointer or not. The constness of the pointer itself is irrelevant for the caller since the pointer itself is always passed as a copy. The function can only make its own copy of the pointer to point to somewhere else, which is irrelevant for the caller.

So, a header file can be created as:

```
#include <vector>

int sum(int const* a, int const* b);
float sum(float* a, float* b);
int sum(std::vector<int>* const);
float sum(std::vector<float>* const);
```

Passing by pointer is like passing by reference. One difference is that when you pass by reference the caller is expected and assumed to have passed a valid element’s reference, not pointing to NULL or other invalid address, while a

pointer could point to NULL for example. Pointers can be used instead of references when passing NULL has a special meaning.

Since C++11 values can also be passed with move semantics. This topic will not be treated in this article but can be studied in other articles like Argument Passing in C++.

Another related topic that won't be covered here is how to call all those functions. If all those headers are included from a source file but are not called, the compilation and linkage will succeed. But if you want to call all functions, there will be some errors because some calls will be ambiguous. The compiler will be able to choose more than one version of sum for certain arguments, especially when choosing whether to pass by copy or by reference (or const reference). That analysis is out of the scope of this article.

Compiling with Different Flags

Let's see, now, a real-life situation related to this subject where hard-to-find bugs can show up.

Go to directory `cpp-article/diff-flags` and look at `Counters.hpp`:

```
class Counters {
public:
    Counters() :
#ifndef NDEBUG // Enabled in debug builds
    m_debugAllCounters(0),
#endif
    m_counter1(0),
    m_counter2(0) {
    }

#ifndef NDEBUG // Enabled in debug build
#endif
    void inc1() {
#ifndef NDEBUG // Enabled in debug build
        ++m_debugAllCounters;
#endif
        ++m_counter1;
    }

    void inc2() {
#ifndef NDEBUG // Enabled in debug build
        ++m_debugAllCounters;
#endif
        ++m_counter2;
    }

#ifndef NDEBUG // Enabled in debug build
    int getDebugAllCounters() { return m_debugAllCounters; }
#endif
    int get1() const { return m_counter1; }
    int get2() const { return m_counter2; }
```

```
private:
#ifndef NDEBUG // Enabled in debug builds
    int m_debugAllCounters;
#endif
    int m_counter1;
    int m_counter2;
};
```

This class has two counters, which start as zero and can be incremented or read. For debug builds, which is how I'll call builds where the `NDEBUG` macro isn't defined, I also add a third counter, which will be incremented every time that any of the other two counters are incremented. That will be a kind of debug helper for this class. Many third-party library classes or even built-in C++ headers (depending on the compiler) use tricks like this to allow different levels of debugging. This allows debug builds to detect iterators going out of range and other interesting things that the library maker could think about. I'll call release builds "builds where the `NDEBUG` macro is defined."

For release builds, the precompiled header looks like (I use `grep` to remove blank lines):

```
$ g++ -E -DNDEBUG Counters.hpp | grep -v -e '^$'
# 1 "Counters.hpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "Counters.hpp"
class Counters {
public:
    Counters() :
    m_counter1(0),
    m_counter2(0) {
    }
    void inc1() {
        ++m_counter1;
    }
    void inc2() {
        ++m_counter2;
    }
    int get1() const { return m_counter1; }
    int get2() const { return m_counter2; }
private:
    int m_counter1;
    int m_counter2;
};
```

While for debug builds, it'll look like:

```
$ g++ -E Counters.hpp | grep -v -e '^$'
# 1 "Counters.hpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
```

```
# 1 "Counters.hpp"
class Counters {
public:
    Counters() :
    m_debugAllCounters(0),
    m_counter1(0),
    m_counter2(0) {
    }
    void inc1() {
        ++m_debugAllCounters;
        ++m_counter1;
    }
    void inc2() {
        ++m_debugAllCounters;
        ++m_counter2;
    }
    int getDebugAllCounters() { return m_debugAllCounters; }
    int get1() const { return m_counter1; }
    int get2() const { return m_counter2; }
private:
    int m_debugAllCounters;
    int m_counter1;
    int m_counter2;
};
```

There is one more counter in debug builds, as I explained earlier.

I also created some helper files.

```
// increment1.hpp:
// Forward declaration so I don't have to include the entire header here
class Counters;

int increment1(Counters&);

// increment1.cpp:
#include "Counters.hpp"

void increment1(Counters& c) {
    c.inc1();
}
```

```
// increment2.hpp:
// Forward declaration so I don't have to include the entire header here
class Counters;

int increment2(Counters&);

// increment2.cpp:
#include "Counters.hpp"

void increment2(Counters& c) {
    c.inc2();
}
```

```
// main.cpp:
#include <iostream>
#include "Counters.hpp"
#include "increment1.hpp"
#include "increment2.hpp"

using namespace std;

int main(int argc, char* argv[]) {
    Counters c;
    increment1(c); // 3 times
    increment1(c);
    increment1(c);
    increment2(c); // 4 times
    increment2(c);
    increment2(c);
    increment2(c);
    cout << "c.get1(): " << c.get1() << endl; // Should be 3
    cout << "c.get2(): " << c.get2() << endl; // Should be 4
#ifndef NDEBUG // For debug builds
    cout << "c.getDebugAllCounters(): " << c.getDebugAllCounters() << er
#endif
    return 0;
}
```

And a `Makefile` that can customize the compiler flags for `increment2.cpp` only:

```
all: main.o increment1.o increment2.o
    g++ -o diff-flags main.o increment1.o increment2.o

main.o: main.cpp increment1.hpp increment2.hpp Counters.hpp
    g++ -c -O2 main.cpp

increment1.o: increment1.cpp Counters.hpp
    g++ -c $(CFLAGS) -O2 increment1.cpp

increment2.o: increment2.cpp Counters.hpp
    g++ -c -O2 increment2.cpp

clean:
    rm -f *.o diff-flags
```

So, let's compile it all in debug mode, without defining `NDEBUG`:

```
$ CFLAGS='' make
g++ -c -O2 main.cpp
g++ -c  -O2 increment1.cpp
g++ -c -O2 increment2.cpp
g++ -o diff-flags main.o increment1.o increment2.o
```

Now run:

```
$ ./diff-flags
c.get1(): 3
```

```
c.get2(): 4
c.getDebugAllCounters(): 7
```

The output is just as expected. Now let's compile just one of the files with `NDEBUG` defined, which would be release mode, and see what happens:

```
$ make clean
rm -f *.o diff-flags
$ CFLAGS='-DNDEBUG' make
g++ -c -O2 main.cpp
g++ -c -DNDEBUG -O2 increment1.cpp
g++ -c -O2 increment2.cpp
g++ -o diff-flags main.o increment1.o increment2.o
$ ./diff-flags
c.get1(): 0
c.get2(): 4
c.getDebugAllCounters(): 7
```

The output isn't as expected. `increment1` function saw a release version of the Counters class, in which there are only two int member fields. So, it incremented the first field, thinking that it was `m_counter1`, and didn't increment anything else since it knows nothing about the `m_debugAllCounters`field. I say that `increment1` incremented the counter because the inc1 method in `Counter` is inline, so it was inlined in `increment1` function body, not called from it. The compiler probably decided to inline it because the `-O2` optimization level flag was used.

So, `m_counter1` was never incremented and `m_debugAllCounters` was incremented instead of it by mistake in `increment1`. That's why we see 0 for `m_counter1` but we still see 7 for `m_debugAllCounters`.

Working in a project where we had tons of source files, grouped in many static libraries, it happened that some of those libraries were compiled without debugging options for `std::vector`, and others were compiled with those options.

Probably at some point, all libraries were using the same flags, but as time passed, new libraries were added without taking those flags into consideration (they weren't default flags, they had been added by hand). We used an IDE to compile, so to see the flags for each library, you had to dig into tabs and windows, having different (and multiple) flags for different compilation modes (release, debug, profile…), so it was even harder to note that the flags weren't consistent.

This caused that in the rare occasions when an object file, compiled with one set of flags, passed a `std::vector` to an object file compiled with a different set of flags, which did certain operations on that vector, the application crashed. Imagine that it wasn't easy to debug since the crash was reported to happen in the release version, and it didn't happen in the debug version (at least not in the same situations that were reported).

The debugger also did crazy things because it was debugging very optimized code. The crashes were happening in correct and trivial code.

The Compiler Does a Lot More Than You May Think

In this article, you have learned about some of the basic language constructs of C++ and how the compiler works with them, starting from the processing stage to the linking stage. Knowing how it works can help you look at the whole process differently and give you more insight into these processes that we take for granted in C++ development.

From a three-step compilation process to mangling of function names and producing different function signatures in different situations, the compiler does a lot of work to offer the power of C++ as a compiled programming language.

I hope you will find the knowledge from this article useful in your C++ projects.