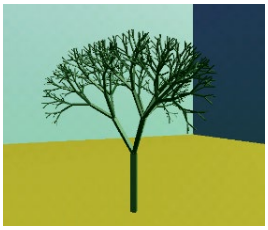


GATS Companion
Multifile Projects in C++

Author: Garth Santor
Editors: Trinh Hān
Version/Copyright: 0.1.0 (2025-01-30)

Overview

C++ projects can become quite large (it was designed to handle large, complex solutions). Large projects can take a considerable amount time to compile and link.

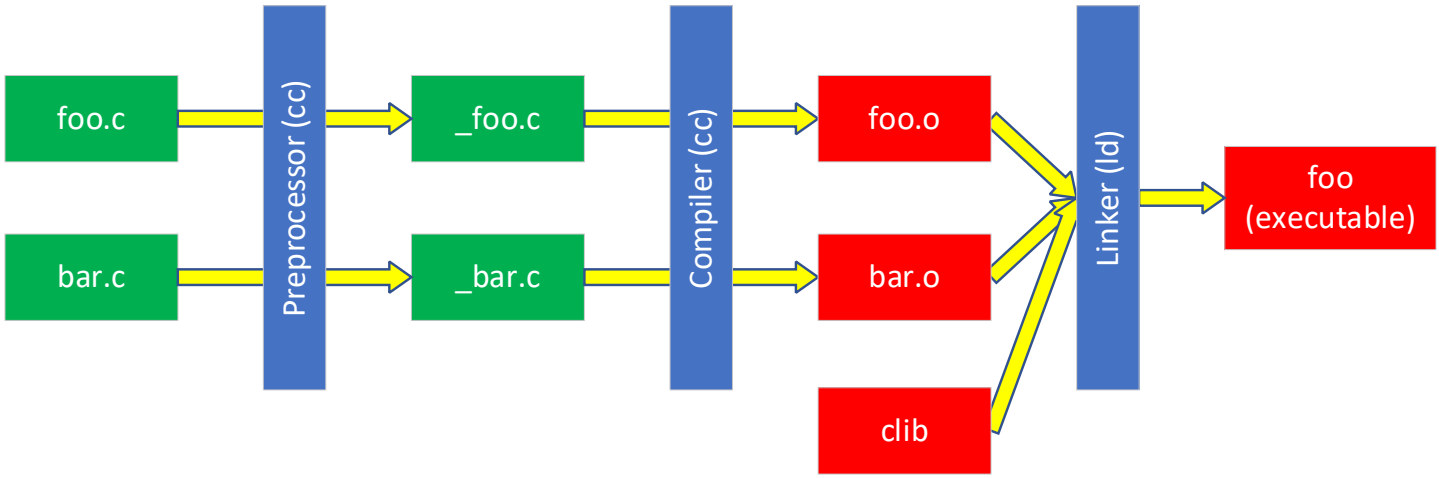
Back in the 1990s I routinely saved up a bunch of edits then performed the compile during lunch or coffee breaks (45-minute build time on that project). The worst I was ever involved in took a network of 20 Sun Workstations 30 hours to complete.

By breaking up the project into multiple source files and employing a build manager we can limit the number of files that need to be recompiled after an edit. This practice along with technology like incremental compilers can drastically reduce build times.

In this document I will focus on how to separate source code into multiple files. What types of files to use, how to name them, and what goes in them. I will not be discussing build management tools (such as *make*, or *Visual Studio*).

Build Process

Here is an abstraction of the typical C & C++ build process.



1. The C compiler – cc – is invoked for each source file (they have the extension .c)
    a. The compiler's preprocessor expands the C-source code storing the result in a temporary file (either on disk or in RAM).
    b. Those temporary files are then compiled which into machine code formatted by that system's *abstract binary interface* (ABI[1])
    c. The result is stored in a file having the same filename as the original source file but with a new extension. UNIX/Linux systems typically use .o whereas Windows typically uses .obj.
2. The *object files* are then fed into the linker along with the necessary libraries. The linker *replaces* the relative addresses of functions and variables with the actual addresses of those elements. Advanced linkers may perform additional rounds of optimization on the completed code (*whole program optimization*). The linker's output is the executable file.

[1] An abstract binary interface is the format rules that binary program modules must follow to guarantee linkage to libraries or system facilities.

Understanding Header Files

Header files are nothing special: they are just text files that are pasted into a source file.

Header files are never compiled directly. Their contents are compiled when a standard source file (.c / .cpp) includes them. If a header file is included into two separate source files, their contents are included in both files!

Example

First, we have the C++ header file containing several constexpr variables.

iec_constants.hpp
```
#pragma once
constexpr int KiB{ 1024 };
constexpr int MiB{ 1024 * KiB };
constexpr int GiB{ 1024 * MiB };
```

After the preprocessor ingests the code on the left, the code on right is produced and sent on to the compiler.

mega_bypes.cpp
```
#include "iec_constants.hpp"
int mega_bytes(int bytes) {
    return bytes / MiB + bytes % MiB>0;
}
```

→

_mega_bypes.cpp
```
constexpr int KiB{ 1024 };
constexpr int MiB{ 1024 * KiB };
constexpr int GiB{ 1024 * MiB };

int mega_bytes(int bytes) {
    return bytes / MiB + bytes % MiB>0;
}
```

What can go wrong? And how to fix it!

Variable definition in a header.

In this example the variable 'common' is to be shared between the two C++ source files. A common first impulse is place the variable to be shared in a header file that can be included into both source files.

common.hpp
```
int common{ 42 };
```

gve_main.cpp
```
#include "common.hpp"
#include <iostream>

int main() {
    std::cout << "main: " <<
        common << "\n";
}
```

secondary.cpp
```
#include "common.hpp"
#include <iostream>

void secondary() {
    std::cout << "second: " <<
        common << "\n";
}
```

When building, the compiler issues the errors:

```
gve_secondary.obj : error LNK2005: "int common" (?common@@3HA) already defined in gve_main.obj

fatal error LNK1169: one or more multiply defined symbols found
```

The reason for this error is that the definition of 'common' was included into both files and therefore memory was allocated for the variables in both .obj files along with the variable identifiers being placed in each file's export list.

When the linker tries to join the two files together, a name clash occurs since you may have not two variables in the same scope with the same name.

Solution

Declare the variable in one of the source files, then tag the identifier in the header file as external.

The 'extern' tag tells the compiler, "There will be an **int** variable called *common*, but it will be defined elsewhere."

common.hpp
```
extern int common;
```

gve_main.cpp
```
#include "common.hpp"
#include <iostream>

int main() {
    std::cout << "main: " <<
        common << "\n";
}
```

secondary.cpp
```
#include "common.hpp"
#include <iostream>

int common{ 42 };

void secondary() {
    std::cout << "second: " <<
        common << "\n";
}
```

It doesn't matter which source file contains the definition of 'common' so long as there is only one.

Repeated Include

In this example a header file containing the variable 'MiB' is included by two other C++ header files – foo.hpp and bar.hpp. A programmer includes each of the two headers to access their respective functions. Unknown to that programmer MiB.hpp has been included twice.

MiB.hpp
```
constexpr int MiB{ 1024 * 1024 };
```

foo.hpp
```
#include "MiB.hpp"

constexpr int foo() {
      return 2 * MiB;
}
```

bar.hpp
```
#include "MiB.hpp"

constexpr int bar() {
      return 3 * MiB;
}
```

RIE_main.cpp
```
#include "foo.hpp"
#include "bar.hpp"
#include <iostream>

int main() {
    std::cout << foo() << "\n";
    std::cout << bar() << "\n";
}
```

When building, the compiler issues the errors:

```
MiB.hpp(8,15): error C2374: 'MiB': redefinition; multiple initialization
```

This can be confusing to new programmers since the variable MiB is only declared once in a single file.

Solution

Since we can't prevent a programmer from including the header multiple time in the same scope (most won't even know this is happening) we must instead prevent the multiply included file from delivering its declarations and definitions more than once.

We'll first look at the traditional and ISO compliant solution, the header guard.

The header guard uses the preprocessor's conditional instruction inherited from C to skip over the enclosed lines if a custom preprocessor identifier has already been defined (it would be defined the first time the header is included.

Let's take a look…

```
MiB.hpp
#if !defined(MIB_GUARD)
#define MIB_GUARD
constexpr int MiB{ 1024 * 1024 };
#endif
```

The first time this header is included the identifier 'MIB_GUARD' does not exist and therefore the conditional (#if) tests true and the enclosed code is processed. 'MiB' is included and defined.

The second time this header is included the identifier 'MIB_GUARD' does exist and therefore the conditional tests false and the enclosed code is **not** processed. The definition of 'MiB' is not repeated.

A new problem…

There is a small potential problem with this solution: if you are not careful and you the same header guard identifier for two different header files. Then including one header, could block the other header.

Solution

As a response to this problem, Microsoft, back in 1995 introduced a proprietary compiler pragma that prevents a header file from multiple inclusions. It has since been adopted by virtually every significant C and C++ compiler making it a de facto industry standard.

The equivalent safe header would be…

```
MiB.hpp
#pragma once
constexpr int MiB{ 1024 * 1024 };
```

You place the pragma at the top of the file (I always place it as the first line of a header – even before the comments). When this pragma is parsed by the preprocessor, the rest of the file is immediately abandoned.

A perfect solution that I'm hoping will make it into the standard one day.

What goes where?

NOTE: This is work in progress and more will be added soon.

Element	Location	Reason
Normal global variable `int x{42}; // outside of any function.`	.cpp	Locate in one source file to avoid a multiple definition error.
Extern global variable declaration `extern int x;`	.hpp	Typically placed in a header file. Can be placed in any .cpp file that requires access to that variable (including the file that defines it).
Normal function definition `int foo() { return 42; }`	.cpp	Locate in one source file to avoid a multiple definition error.
Normal function declaration `int foo();`	.hpp	The function prototype correctly instructs the compiler how to generate the call to that function.
constexpr variable	.hpp	Typically placed in a header file.

`constexpr int MiB{1024 * 1024 };`		Can be placed in any .cpp file that requires access to that variable. The definition must be visible at the scope of the call to allow for the optimization to work.

constexpr, consteval functions
 	
`constexpr int foo(int n) {`
`  return n*n;`
`}`
	.hpp	Typically placed in a header file.
		Can be placed in any .cpp file that requires access to that function.
		The definition must be visible at the scope of the call to allow for the optimization to work.

inline functions `inline int bar() { return 24; }`	.hpp	Typically placed into header files (for the same reasons as constexpr and consteval functions). **NOTE**: the inline keyword is considered a suggestion by modern C++ and may be ignored by the compiler. Best practice to convert any inline function to a constexpr function. If the function is not constant evaluation possible, consider using std::is_constant_evaluatable() and make it a hybrid function.

Document History

Version	Date	Activity
0.0.0	2024-01-23	Document created.
0.1.0	2025-01-30	Added function prototype.