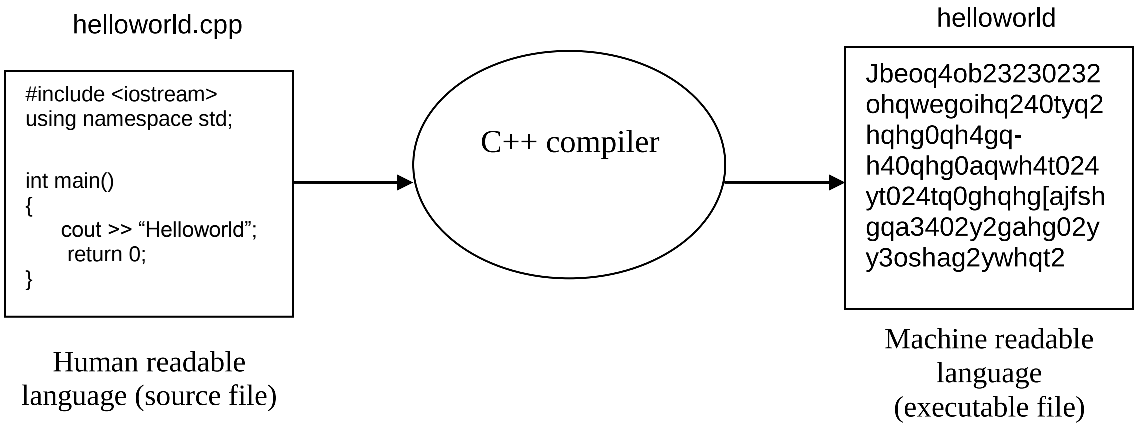
# EECS 211
# Introduction to Compiling and Linking

The transformation of a set of C++ source and header files into an executable file requires many steps. This document provides a brief overview of those steps to help you understand the various commands you yourself have to issue in order to run and test your programs.

## Compiling, Linking, and Building a Program

The process of transforming a set of text files containing C++ code into an executable file is called "building". Because the code for the project may be spread across many C++ code and header files, building a project involves many smaller steps. The one most commonly thought of is compiling (translating a single C++ file into machine instructions). But there are other steps as well – pre-processing and linking. This section briefly describes what happens in each of these steps.

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses and can understand. Typically, a programmer writes language statements in a language, such as C++, using an *editor*. The file that is created (*filename.cpp* in the C++ language) contains what are generally called the *source or high-level programming statements*. The programmer then runs the appropriate language compiler (such as C++), specifying the name of the file that contains the source statements. The job of the compiler, then, is to take that source file (cpp file), check it line by line for possible syntax errors, and transform it into a corresponding sequence of machine-level instructions. Note that each type of computer processor has it's own set of machine instructions. Thus, there is not a single C++ compiler but rather a compiler for each target hardware platform.



```
#include <iostream>
using namespace std;

int main()
{
    cout >> "Helloworld";
     return 0;
}
```

**Figure 1. Compiling a Single C++ File**

Compiling a C++ file doesn't produce an executable file! The program described in the C++ file would likely have included calls to functions in the operating system API (Application Programming Interface), such as I/O functions, or to built-in functions of the language, such as math functions. The file produced by the compiler is called an “object file”. In addition to the machine instructions corresponding to the (transformed) lines of C++ code it contains references to these various externally defined functions. It is not in executable form by itself but must have these externally defined functions added, or “linked”, into it. A more complete description of the process is given in Figure 2:

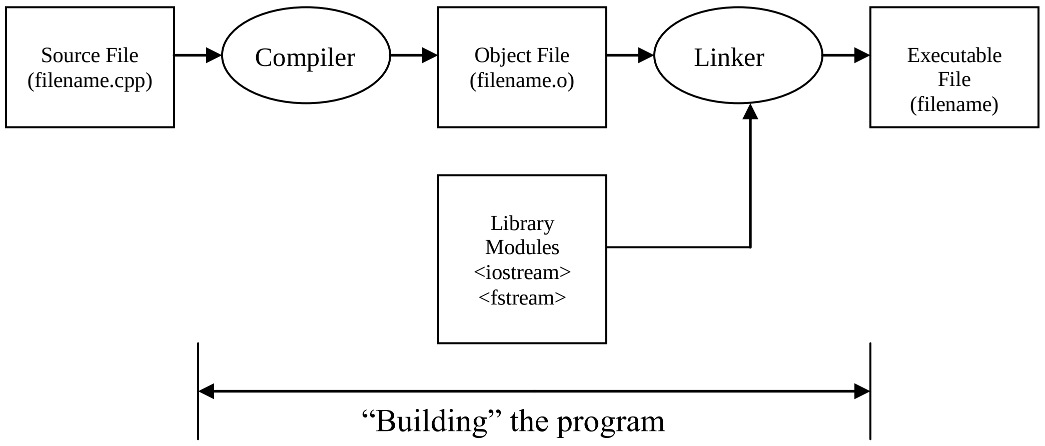

Source File (filename.cpp)
Compiler
Object File (filename.o)
Linker
Executable File (filename)
Library Modules <iostream> <fstream>
“Building” the program

**Figure 2. Compiling and Linking a Project in a Single C++ File**

In addition to the source (cpp) files, which contain executable statements, there are also header (*.h* ) files. Header files normally contain only declarations. Before “compilation” takes places, an additional step must be performed: “preprocessing”. The preprocessor program reads the source file as text and produces another text file as output. Source code lines which begin with the special character ‘#’ are actually written not in C++ itself, but in preprocessor language. Based on these statements, the preprocessor makes substitutions of one character string for another, substitutes the text of a header file for *#include* statements, and even expands some function calls into other text. This additional step is shown in Figure 3.

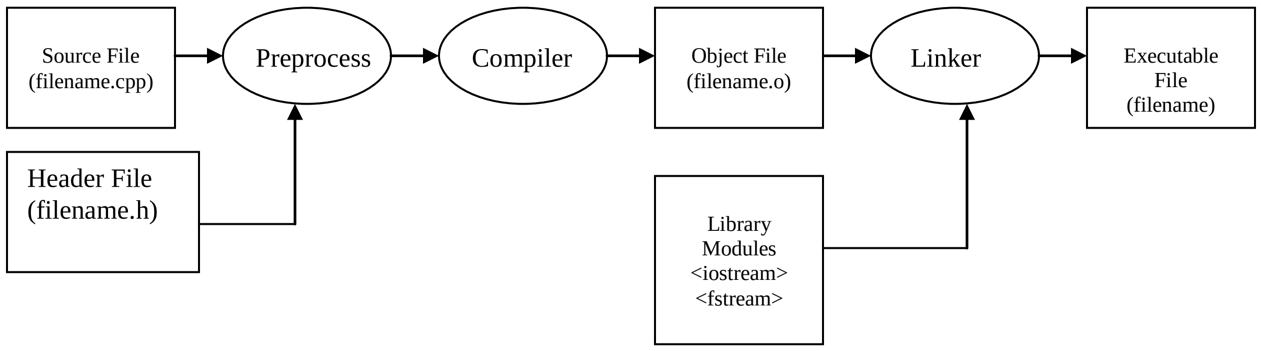

Source File (filename.cpp)
Preprocess
Compiler
Object File (filename.o)
Linker
Executable File (filename)
Header File (filename.h)
Library Modules <iostream> <fstream>

**Figure 3: Preprocessing step in the building operation**

Most projects have the code in a set of C++ and header files. This keeps the files of a more manageable size and allows the code to be “modularized’, that is, to be split up into smaller segments with each segment containing only closely related code such as the implementation of a class or a set of related utility functions. In order to build such a project, each C++ file must be compiled and the object files then linked together with the library functions. Each compilation step, of course, involves a preprocessing step in which the # instructions are performed to modify the text of the file. Figure 4 shows the build process for a multi-file project.

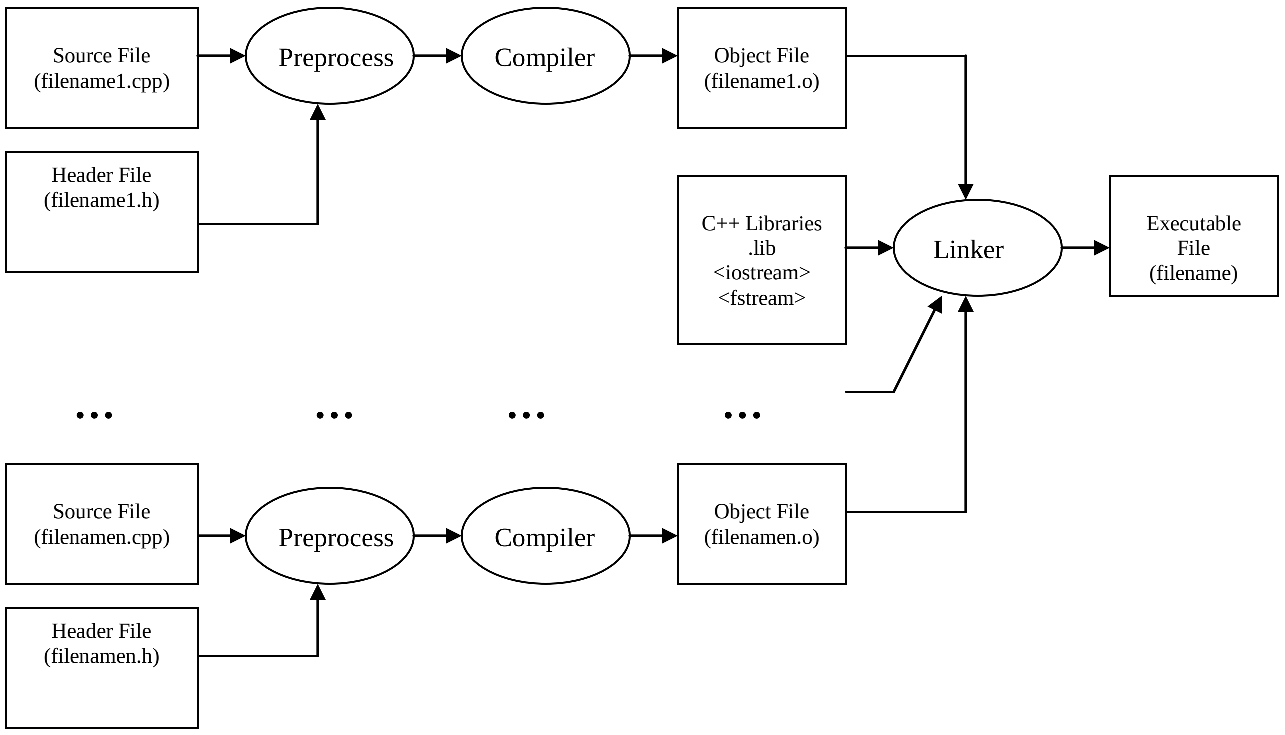

Source File (filename1.cpp)
Preprocess
Compiler
Object File (filename1.o)
Header File (filename1.h)
C++ Libraries .lib <iostream> <fstream>
Linker
Executable File (filename)
Source File (filenamen.cpp)
Preprocess
Compiler
Object File (filenamen.o)
Header File (filenamen.h)

**Figure 4: Building a Multi-file Project**

## Compiling/Linking vs. Testing/Debugging

Just because a project builds does not mean that the program is correct, that is, that the program does what it’s supposed to do. A successful build only means that the compiler did not find any syntactical errors in any of the C++ files and that the linker was able to resolve all external references. The program must be tested, and any errors that are found must be fixed.

Testing is the process of running the program on test input cases for which the correct behavior or output is known. There are well-developed theories and practices for designing good test sets. Some of the more common considerations are:

- Natural boundaries in the data. For example, if legal test scores are 0-100 inclussive, then the program should be tested with inputs of –1, 0, 1, 99, 100, and 101 as well as some other values away from these borders.
- Iterations performed correct number of times. For example, if a loop is supposed to execute N times, be sure that it doesn't execute N-1 or N+1 times. If a loop is terminated by a sentinal value, make sure the sentinal value does not get included with the valid data and that the last valid datum does get included.

Debugging is both the finding and fixing of errors. It involves testing, of course, but then also includes changing the code to fix the errors that are found. There are two main methods of debugging. The first method involves straightforward execution of the program followed by an analysis of the results. For each incorrect result the source code is examined to see what that error was made. It is interesting to note that test cases are often designed to help pinpoint where certain kinds of errors might occur. The second method is to use a special feature of moden compiler systems, called "the debugger", to step through the program, in some cases, line by line. Debuggers allow the programmer to see exactly how the program processes an input, to stop the program at virtually any point, and to examine and even change intermediate values of variables in the code.

## Visual C++ and other "integrated" C++ systems

Visual C++ is an "integrated" system, which means that the user can create and edit files, compile individual files, build the project and execute in normal mode or debug mode, all from within the same application. Like most WINDOWS applications, there is a command bar across the top of the screen. When a "Windows console" application is opened or created. A frame on the left side of the window is created in which the user can view the parts of the project in either "class" view or "file" view. In EECS 231, probably the most convenient mode is file view. In file view, the user can add source files, header files and other files to be used in the project (e.g., external DLLs, etc.). By double clicking a file, the user can open the file in the Visual C++ editor. The currently active file can be compiled. The project can be built and executed. During execution or debugging, a new window is opened in which the user types standard input (cin) and sees standard output (cout). If the user specifies debugging instead of normal execution, various other windows open up inside the Visual C++ frame which allow the user to specifiy execution (execute one line, step into a function, run to the next breakpoint, etc.) and to view and modify variables in the program.

## The GCC/C++ compiler

GCC stands for "*GNU Compiler Collection*". As its name states, GCC is a collection of compilers for several programming languages, and C++ is just one of them. That's why the GNU C++ compiler is referred to as the GCC/C++ compiler. GCC is the standard C/C++ compiler on UNIX-style systems.

As with most UNIX utilities, a compiler in the GCC collection is invoked by typing a command line to the shell. GCC assumes the files will be in the current working directory, so before invoking a GCC compiler the user must navigate to the directory containing the files.

Once you are in the correct directory, the simplest way to compile a single C++ file, say `filename.cpp, is` by typing the following command at the prompt:

```
g++ filename.cpp
```

GCC attempts to perform all three steps in the build process – preprocess, compile, and link. If the source code had no errors in it (i.e., no syntactic errors or unresolved external references), the command above will create an executable file in the working directory called `filename.out`. To actually run the program, simply type at the command prompt `./filename.out`. NOTE: Header files are NOT included on the command line; the preprocessor will find and process them automatically.

If your program does have syntax errors, no .out file will be created. Instead, the compiler will generate informative messages telling what errors it found and on which lines of the file they were found. For example, if it displays the following message:

```
%mumbo> g++ filename.cpp
filename.cpp: In function main()
filename.cpp: 4: 'i' undeclared (first use of this function)
%mumbo>
```

it means that in the function main() of your program, at line #4, your program uses the variable 'i' whose type was never specified before.

The GCC compiler accepts several additional options. Here are the most common ones.

**-ggdb:**
```
g++ -ggdb filename.cpp
```

The `-ggdb` option includes extra information in the .out file to allow that file to be "debugged" using the GDB debugger.

**-Wall**

```
g++ -Wall filename.cpp
```

The `-Wall` flag instructs the compiler to display ALL the warning messages you may get that describe possible errors in your source code. If warnings are the only messages you receive when you compile your source code, an executable will still be created. Only if there are ERROR messages will an executable not be created. GCC, like many C++ systems, will attempt to fix or overlook small differences with the standard C/C++ syntax. Sometimes this is ok because the fix is what you meant anyway. Other times the warning indicates something the programmer truly forgot or did wrong. Using `-Wall` will allow you to detect such warnings ahead and decide if they are really errors or not.

**-c**

Compile the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file. By default, the object file name for a source file is made by replacing the suffix *.cpp* with *.o*. This option is useful when working in a multi-file project but working on one of the cpp files. It allows you to check for typos and syntax errors in that file without trying to build the whole project.

**-o**

The `-o` flag is one of the more useful ones. This flag will allow you to rename the executable file from something other than `filename.out`. To use the `-o` option, place it after the source code file(s) and other options, followed by the name you wish to give the executable. For example, to compile a source code file named `filename.cpp` into an executable file called `myproject`, or `myproject.exe` type the following command:

```
g++ filename.cpp -o myproject
```

Doing so will create the executable `myproject` that you would type at the command prompt to run your program.

It is possible to use more than one flag at once when you compile. Except for the –o option, most options are typed after the g++ command and before the list of source files. For example, let's say you entered the following `g++` command:

```
g++ -ggdb -Wall filename.cpp -o filename
```

This command will compile the source code file `filename.cpp`, include the necessary information so that the executable can be used with the GDB debugger (`-ggdb`), will show all warning messages when compiled (`-Wall`), and will create the executable into a file called `myproject` (`-o myproject`).

As previously noted, most C++ programs are contained in several source and header files. GCC allows the user to build the entire project in one command by processing multiple source files on the line. (Remember, header files are NOT listed on the command line.) The format is:

```
g++ [options] source_file_1 source_file_2 source_file_3...
```

If the –o option is not used, the executable file will be given the same name as the last source file. For example, suppose a program is contained in 3 files: `project.cpp` contains the `main()` function, `functions.h` contains the prototypes for functions that your program uses, and `functions.cpp` contains the implementations for those functions. To compile these 3 files into one executable, type at the prompt:

```
g++ functions.cpp project.cpp
```

The above command will then compile the two source files and create one executable called `project.out`. Finally, just as with single-file programs, you can add options flags when you compile:

```
g++ -ggdb -Wall functions.cc project.cc -o myproject
```

This command will compile the two source files `functions.cc` and `project.cc` so that the executable can be used with the GDB debugger (`-ggdb`), all compilation warnings will be displayed (`-Wall`), and the executable file will be called `myproject` (`-o myproject`).