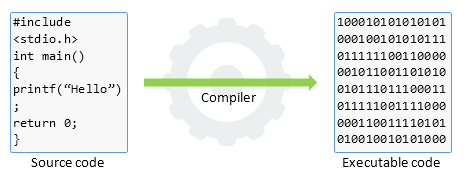
# Compilation Process

## What Is Compilation?

The process of translating source code written in high-level language to low-level machine code is called *compilation*. The compilation is done by a special software known as [compiler](https://codeforwin.org/2017/05/compiler-and-its-need.html). The compiler checks source code for any syntactical or structural error, and generates object code with extension .obj (in Windows) or .o (in Linux) if source code is error-free.

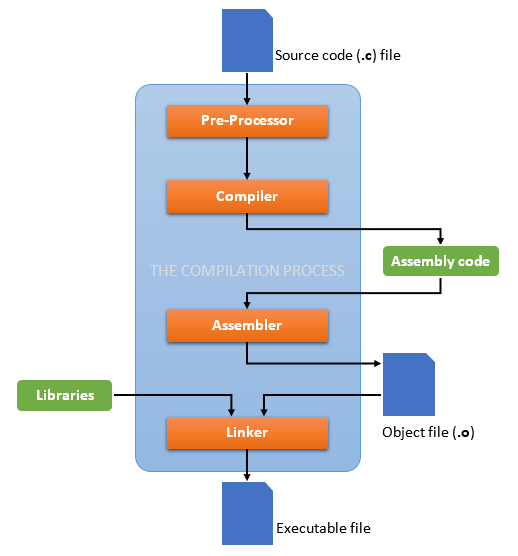


## Compilation Process

The entire C/C++ compilation process is broken to four stages.

* Pre-processing
* Compilation
* Assembling
* Linking

The below image describes the entire C/C++ compilation process:



To take a deep dive inside the C/C++ compilation process, let’s compile a C program. Create a text file named helloworld.c without following content:

#include <stdio.h>

int main()

{

printf("Hello, World!");

return 0;

}

To compile the above program, open command prompt and use below command:

gcc -save-temps helloworld.c -o helloworld

The -save-temps option will preserve and save all temporary files created during the C/C++ compilation. It will generate four files in the same directory namely:

// On Linux:

helloworld.i (generated by pre-processor)

helloworld.s (generated by compiler)

helloworld.o (generated by assembler)

helloworld (generated by linker)

// On Windows:

helloworld.i (generated by pre-processor)

helloworld.s (generated by compiler)

helloworld.obj (generated by assembler)

helloworld.exe (generated by linker)

Now let's look into these files and learn about different stages of compilation.

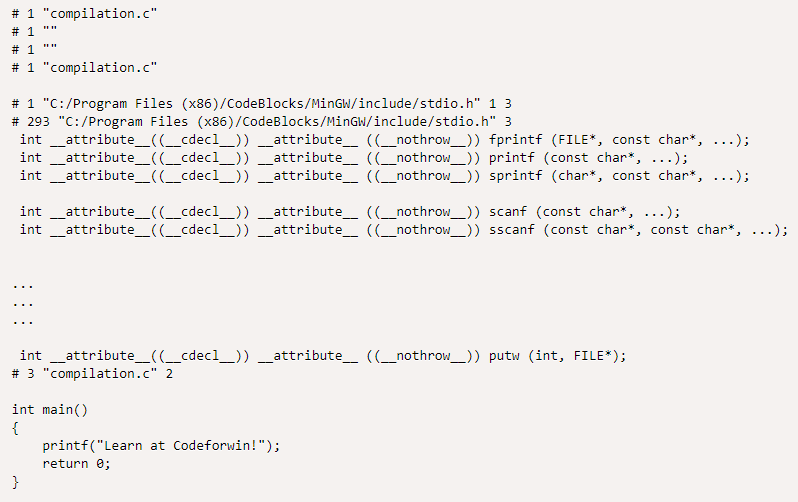
### Pre-Processing

Pre-processor is a small software that performs below tasks:

* Remove comments from the source code.
* Expansion of included header files.
* Macro expansion.

After pre-processing, a temporary with .i extension is generated. Since, it inserts contents of header files to the source code file, this generated file has a larger size than the original source code file.

Here is an extract of compilation.i file:



You can notice that the statement #include<stdio.h> is replaced by its contents. Comment before the #include line is also trimmed.

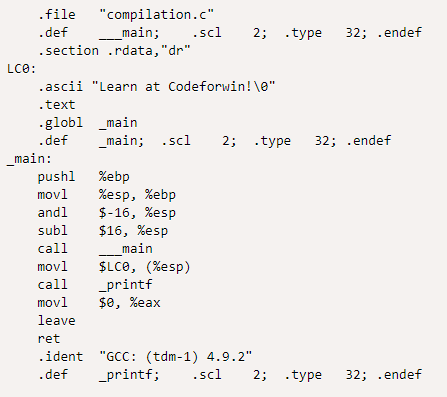
### Compilation

In next phase, the compiler performs following tasks:

* Check the program for syntax errors.
* Translate the file into assembly language (intermediate code).
* Optionally optimize the translated code for better performance.

After compiling, an ***intermediate code file*** (in assembly language) with .s extension is generated.

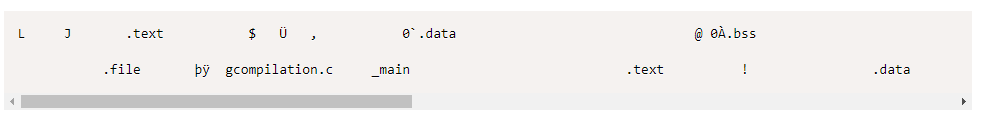
Let us look into compilation.s file:



### Assembling

Assembler accepts the intermediate code file and translates to machine code. After successful assembling, it generates .o file (on Linux) or .obj file (on Windows) known as ***object file***.

In our case, a compilation.o file is generated. It is encoded in machine language and cannot be viewed using text editors. However, if you still open it in a text editor, it looks like:



### Linking

Finally, the linker comes in action and performs the final task of compilation process. It accepts the object file. It links all the function calls with their original definition; that means the function printf() gets linked to its original definition.

The linker can generate one of following files based on your configuration:

* ***Executable file*** (no extension on Linux, or .exe on Windows).
* ***Static library*** (.a on Linux – also called *archive library*, or .lib on Windows).
* ***Dynamic library*** (.so on Linux – also called *shared object library*, or .dll on Windows).

**Static Linking and Dynamic Linking**

|  |  |
| --- | --- |
| **Static linking** | **Dynamic linking** |
| Done by the linker in the final step of the compilation. | Done at run time by the OS. |
| Statically linked files consume more disk and memory as all the modules are already linked. | Only one copy of the reference module is stored which is used by many programs, thereby saving memory and disk space. |
| All the library modules are copied to the final executable image. When the program is loaded, the OS keeps only a single file in the memory which contains both the source code and the referencing libraries. | Only the names of external or shared libraries are kept in the memory. Dynamic linking lets many programs use single copy of executable module. |
| If external source program is changed then they have to be re-compiled and re-linked. | Only a single module needs to be updated and re-compiled. This is one of the greatest advantages that dynamic linking offers. |
| Statically linked programs are faster than their dynamic counterpart. | Dynamically linked programs are slower than their static counterpart. |
| Since the statically linked file contains every package and module, no compatibility issues occur. | Since the library files are separately stored there may be compatibility issues (say one library file is compiled by new version of compiler). |
| Statically linked programs always take constant load time. | The time is variable in dynamically linked programs. |

# GCC/G++

Simple GCC command generator: <https://www.rapidtables.com/code/linux/gcc.html>

List of command options/flags: <https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html>

# GDB

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjHGWPN_S9NS_Ky1-tC8ZrUI>

About **GUI for GDB**, there are many options. But I would suggest Eclipse (see *Tutorials/IDEs/Eclipse/Eclipse Tutorial.docx*) which runs based on Makefile and gcc/g++ (we can think that "Eclipse is like a front-end to GDB"), and offers both local and remote debugging.

# Makefile

**Note**

This guide is for Linux. For Window, follow the same principle.

## What Is Makefile?

A Makefile is basically **a script which defines and controls the whole building process automatically**. It works based on a strict set of rules defined and implemented by the [***GNU make***](http://www.gnu.org/software/make/manual/make.html) program.

For example, let’s assume we have the following source files in the same directory:

// In main.c

#include <stdio.h>

#include "functions.h"

int main() {

   print\_hello();

   printf("The factorial of 5 is %d.\n"), factorial(5);

   return 0;

}

// In hello.c:

#include <stdio.h>

#include "functions.h"

void print\_hello() {

   prinf("Hello, World!\n");

}

// In factorial.c:

#include "functions.h"

int factorial(int n) {

   if (n != 1) {

      return (n \* factorial(n-1));

   } else {

      return 1;

   }

}

// In functions.h:

void print\_hello();

int factorial(int n);

The trivial way to compile these files into executable is:

$ gcc main.c hello.c factorial.c –I. -o hello

Or:

$ cc main.c hello.c factorial.c –I. -o hello

Or you can reverse the order of output and input files as follows:

$ cc -o hello main.c hello.c factorial.c –I.

For a large project where we might have hundreds of source files, it becomes extremely difficult and time-wasting to maintain the building process. Moreover, you might notice that you usually only work on a small section of the program (e.g., several functions in some files) and much of the remaining program is unchanged; re-compiling all source files is definitively unnecessary.

That's when a Makefile comes to play! The following sections describe how to write Makefile steps by steps.

## Rules

### File Name

A Makefile should be named as Makefile or Makefile.

### Target - Dependency

A target entry form looks like this:

target: dependency-list

    command

    command

    command

Where:

* The target is the output file. When you run 'make', the target serves as an argument ($ make <target>).
* The dependency-list is the list of input files. Multiple dependencies are separated by spaces.
* The commands are a series of steps used to make the target. Each command has to start with a tab character, NOT spaces.

### Flow

**The first target is referred as the default target which is called first when we run make**. The make program then looks at the default target's list of dependencies; if any of them are older, it will run the targets for those dependencies before running itself.

**Example 1**:

some\_file: other\_file

echo "This will run second, because it depends on 'other\_file'."

other\_file:

echo "This will run first."

Output:

$ make

This will run first.

This will run second, because it depends on 'other\_file'.

**Example 2**:

blah: blah.o

cc blah.o -o blah

blah.o: blah.c

cc -c blah.c -o blah.o

blah.c:

echo "int main() { return 0; }" > blah.c

clean:

rm -f blah.o blah.c blah

Output:

$ make

echo "int main() { return 0; }" > blah.c

cc -c blah.c -o blah.o

cc blah.o -o blah

$ make clean

rm -f blah.o blah.c blah

### Re-Compiling

The *GNU make* program is intelligent and works based on the changes you do in your source files. From the very first example, we know that all main.c, hello.c and factorial.c are dependent on functions.h, and main.c is dependent on both hello.c and factorial.c. If you make any changes in functions.h, then the *make* program will recompile all the source files to generate new object files. However, if you make a change in main.c, as this is not dependent of any other file, then only main.c file will be recompiled.

**While compiling a file, the *make* program checks its object file and compares the time stamps. If the source file has a newer time stamp than the object file, it will generate a new object file** assuming that the source file has been changed.

## Variables

<https://www.gnu.org/software/make/manual/html_node/Using-Variables.html>

## Macros

The *make* program allows to use macros, which are similar to variables and can only be strings.

### User-Defined Macros

files = file1 file2

some\_file: $(files)

echo "Look at this variable: " $(files)

touch some\_file

file1:

touch file1

file2:

touch file2

### Pre-Defined Macros

**Conventional Macros**

There are various pre-defined macros used in implicit rules. You can see them by typing "make -p" to print out the defaults. They fall into two classes:

1. Macros that are names of programs:

* CC: Program to compiling C programs. Default is 'cc'.
* CXX: Program to compiling C++ programs. Default is 'g++'.
* CPP: Program to running the C preprocessor. Default is '$(CC) -E'.
* LINT: Program to use to run lint on source code. Default is 'lint'.
* RM: Command to remove a file. Default is 'rm -f'.

2. Macros that contain arguments of the programs:

* ASFLAGS: Flags to give to the assembler when explicitly invoked on a '.s' or '.S' file.
* CFLAGS: Flags to give to the C compiler.
* CXXFLAGS: Flags to give to the C++ compiler.
* CPPFLAGS: Flags to give to the C preprocessor and programs, which use it (such as C and Fortran compilers).
* LDFLAGS: Flags to give to compilers when they are supposed to invoke the linker, 'ld'.
* LINTFLAGS: Flags to give to lint.

**Notes**:

* You can cancel all variables used by implicit rules with '-R' or '--no-builtin-variables' option.
* You can define macros at the command line as: $ make CPP = /home/courses/spring02

**Special Macros**

There are some special macros predefined, such as:

* $@ is the name of the output file being generated (left hand side of :).
* $? is the list of names of changed dependencies (
* $^ is the dependencies list (right hand side of :).
* $< is the first item in the dependencies list (first item at the right-hand side of :).
* $\* is the prefix shared by target and dependencies.

Example 1:

# $@ represents hello and $? picks up all the changed source files.

SRC = main.c factorial.c hello.c

CFLAG = -Wall -g

CC = gcc

hello: ${SRC}

${CC} ${CFLAGS} $? $(LDFLAGS) -o $@

Example 2:

# Make of .o files out of .c files

# Way 1:

%.o: %.c:

$(CC) $(CFLAGS) -c $^ -o $@

# Way2:

# Note: This way is old (not recommended). Should use the first way instead.

.c.o:

$(CC) $(CFLAGS) -c $^ -o $@

Alternatively:

.c.o:

$(CC) $(CFLAGS) -c $\*.c

## Examples

The following Makefile example is generic. You can apply it (and customize it) to many projects:

# Run 'make' or 'make all' to build executable file

# Run 'make clean' to remove all object files and executable files

# Run 'make depend' to use makedepend to automatically generate dependencies  (which are added to end of Makefile)

# Define the compiler to use ('gcc' if C, or 'g++' if C++)

CC = gcc

# Define compiler flags

#   For example, '-Wall' enables all warnings, '-g' adds debug info

CFLAGS = -Wall -g

# Define directories containing header files other than /usr/include

#   The rule is appending '-I' before the directory name

HEADERS = -I/home/triho/include  -I../include

# Define directories containing libraries other than /usr/lib

#   The rule is appending '-L' before the directory name

LFLAGS = -L/home/triho/lib  -L../lib

# Define names of libraries to link into executable:

#   The rule is appending '-l' before the lib name

#   For example, to link in libraries mylib.so or mylib.a, use -lmylib (no extension)

LIBS = -lmylib1 -lmylib2

# Define the C source files

SRCS =  emitter.c \

Better way: Search all .c files in dir: https://stackoverflow.com/a/3774731

        error.c \

        init.c \

        main.c \

        parser.c

# Define the C object files

#   This uses Suffix Replacement within a macro: $(name:oldstr=newstr)

#   For each word in 'name', replace 'oldstr' with 'newstr'

# Below we're replacing the suffix .c of all words in SRCS with the .o suffix

OBJS = $(SRCS:.c=.o)

# Define the executable file

EXE = mycc

# Running 'make' will invoke the first target entry in the file

# You can name this target entry anything, but "default" or "all" are the convention

all: $(EXE)

echo "All source files have been compiled"

Without it, still works

# Compiling: Create object files from source files

# Below we're using pre-defined macros:

#    '$^' is the list of names of the dependencies (.c files)

#    '$@' is the name of the target (.o file)

.c.o:

Another way:

%.o: %.c:

$(CC) $(CFLAGS) $(HEADERS) -c $^ -o $@

# Linking: Create the executable file from object files

$(EXE): $(OBJS)

$(CC) $(CFLAGS) $(HEADERS) $(OBJS) $(LFLAGS) $(LIBS) -o $(EXE)

# Running 'make clean' removes the executable file, all .o files and \*~ backup files

clean:

$(RM) \*.o \*~ $(EXE)

# Running 'make depend' generates dependencies of C source files automatically

# Install makedepend on Ubuntu: sudo apt-get install xutils-dev

depend: $(SRCS)

makedepend $(HEADERS) $^

# DO NOT DELETE THIS LINE -- make depend needs it

## Tips

### .PHONY

Adding .PHONY to a target prevents *make* from confusing the phony target with a file name.

In the below example, if there is a source code file named clean.c, then make clean will still run correctly.

.PHONY: clean

clean:

$(RM) \*.o \*~ $(EXECUTABLE)

### Makedepend (Auto Dependency Creation)

Install:

* Ubuntu: sudo apt-get install xutils-dev

<https://linux.die.net/man/1/makedepend>

[https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/make.html#Makefile\_depends](https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/make.html#makefile_depends)

### Others

* More features: [https://www.tutorialspoint.com/Makefile/Makefile\_features.htm](https://www.tutorialspoint.com/makefile/makefile_features.htm)
* Compile all C files at once: [here](https://stackoverflow.com/questions/170467/makefiles-compile-all-c-files-at-once)
* Using if … else in Makefile: [here](https://www.avrfreaks.net/forum/how-use-if-else-endif-makefile)
* Include other makefiles: by using the include directive.
* Debug mode vs release mode: [here](https://randu.org/tutorials/c/make.php)
* [CVS](https://www.gnu.org/software/automake/manual/html_node/CVS.html) is another tool that may be useful for very large projects. CVS stands for Concurrent Versions System and it allows you to record the history of your source files. CVS stores the base source and then stores the differences for each version. CVS also allows for protecting code pieces of a multi-developer effort from accidental overwriting.

# CMake

## What Is CMake?

The 'make' utility (or rather a Makefile) is a buildsystem. It drives the compiler and other build tools to build code.

CMake is a **generator of buildsystems**. It can produce:

* Makefile (Unix, MinGW, NMake, etc.)
* Buildsystem files (Ninja, etc.)
* IDE project files (Visual Studio, XCode, Eclipse CDT, CodeBlock, KDevelop, etc.)

The best thing about CMake is that the **same CMakeLists.txt file is used across platforms**. So, if you have a platform-independent project, CMake is a way to make it buildsystem-independent. For example, if you have Windows developers who used to Visual Studio and Unix developers who swear by GNU Make, CMake is a very convenient way to go.

CMake is widely used for the C and C++ languages, but it may be used to build source code of other languages too.

## How To Install

<https://cmake.org/install/>

## How To Use

Tutorial: <https://cmake.org/cmake/help/latest/index.html>

Examples: <https://github.com/ttroy50/cmake-examples>

# Libraries

## Shared/Dynamic Libraries

### Advantages

The biggest advantage of shared/dynamic library is that they make it easy to share functions and resources across multiple executable files. Multiple applications can also access the contents of a single copy of a shared/dynamic library in memory at the same time.

* **Uses fewer resources:** Don't get loaded into the RAM together with the main program, so don't occupy space unless required. When a lib is needed, it is loaded and run.
* **Promotes modular architecture:** Help develop large programsthat require multiple language versions or a program that requires modular architecture.
* **Aid easy deployment and installation:** When a function within a shared/dynamic library needs an update or fix, the deployment and installation of the lib does not require the program to be relinked with the lib. Additionally, if multiple programs use the same lib, then all of them get benefited from the update or the fix.

### Create and Use

**Note**

This guide uses g++ (minGW) to create shared/dynamic libraries. If you use another compiler, check this [guide](https://www.oreilly.com/library/view/c-cookbook/0596007612/ch01s05.html).

#### Windows

**Note**: After the building process, the folder tree will look like that:

$ create-dll

│ build.bat

│ clean.bat

│

├───bin

│ addition\_lib.dll

│ test\_lib.exe

│

├───build

│ addition\_lib.o

│ test\_lib.o

│

└───src

addition\_lib.cpp

addition\_lib.h

test\_lib.cpp

**1. Prepare the source code**

In src/addition\_lib.h:

#pragma once

#ifdef \_\_cplusplus

    extern "C" {

#endif

double add(double a, double b);

#ifdef \_\_cplusplus

    }

#endif

In src/addition\_lib.cpp:

#include <stdio.h>

#include "addition\_lib.h"

double add(double a, double b)

{

    return a + b;

}

In src/test\_lib.cpp:

#include <iostream>

#include "addition\_lib.h"

void test\_add()

{

    double sum = add(1.2, 2.4);

    std::cout << "Sum is " << sum << std::endl;

}

int main()

{

    test\_add();

    return 0;

}

**2. Prepare the build script**

In build.bat:

set libfile=addition\_lib

set testlibfile=test\_lib

set src=src

set build=build

set bin=bin

Here we decide to export all functions and variables to the DLL

:: Build dll

g++ -c %src%\%libfile%.cpp

move /Y \*.o %build%

g++ -shared -o %bin%\%libfile%.dll %build%\%libfile%.o -Wl,--export-all-symbols

:: Build exe

g++ -c %src%\%testlibfile%.cpp

move /Y \*.o %build%

g++ -o %bin%\%testlibfile%.exe %build%\%testlibfile%.o -L%bin% -l%libfile%

pause

If you want to rebuild the program, you can also need a clean.bat to clean output files quickly:

set build=build

set bin=bin

del -f %build%\\*.a %build%\\*.o   %bin%\\*.dll %bin%\\*.exe

pause

**3. Build the DLL**

To build the DLL, run:

# In create-dll dir:

build.bat

**4. Run the DLL**

To run the DLL, run:

# In create-dll/bin dir:

test\_lib.exe

Output:

Sum is 3.6

##### Common Errors

* Running this "g++ -o out-file.exe -L. llib-file.dll out-file.o" will cause error "*cannot file lib-file.dll*". The correct way is: "g++ -o out-file.exe -L. llib-file out-file.o". So **remove the .dll extension from the lib-file name**.
* Running this "g++ -o out-file.exe -L. llib-dir\lib-file out-file.o" will cause error "*cannot file lib-dir\lib-file*". The correct way is: "g++ -o out-file.exe -Llib-dir llib-file out-file.o". So the **input lib only accepts file name, not file path**. That's why -L is created.

##### Tip: Choose Functions to Export

In the above example, we export all functions and variables of addition\_lib to the DLL. What if we only want to export some functions? We have to change things a bit as below:

In src/addition\_lib.h:

#pragma once

#ifdef \_\_cplusplus

    extern "C" {

#endif

#ifdef BUILD\_LIB

Here we only export add() to the DLL

    #define DLL\_LIB \_\_declspec(dllexport)

#else

    #define DLL\_LIB \_\_declspec(dllimport)

#endif

DLL\_LIB double add(double a, double b);

#ifdef \_\_cplusplus

    }

#endif

In build.bat, modify the following lines:

From:

:: Build dll

g++ -c %src%\%libfile%.cpp

move /Y \*.o %build%

g++ -shared -o %bin%\%libfile%.dll %build%\%libfile%.o -Wl,--export-all-symbols

To:

:: Build dll

g++ -c -DBUILD\_LIB %src%\%libfile%.cpp

move /Y \*.o %build%

g++ -shared -o %bin%\%libfile%.dll %build%\%libfile%.o

#### Linux

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFn6zQQkJ2e8vxCVxhl2yuD>

<https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html>

**Example:**

$ g++ -std=c++11 -Wall -g -fPIC src/addition\_lib.cpp -shared -o libmylib.so

$ g++ -std=c++11 -Wall -g src/test\_lib.cpp -Wl,-rpath=. -L. -lmylib

**Notes:**

* The dynamic library must be named as lib\* (e.g., "libmylib") when creating it. And when calling it, must replace the "lib" with "l" (e.g., "libmylib" will become "-lmylib")

## Static Libraries

### Windows

### Linux

<https://medium.com/@meghamohan/all-about-static-libraries-in-c-cea57990c495>

# Cross Compiler

<http://eslinuxprogramming.blogspot.com/2015/04/cross-compiler.html>

<https://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/>