Introduction to Compilation Process

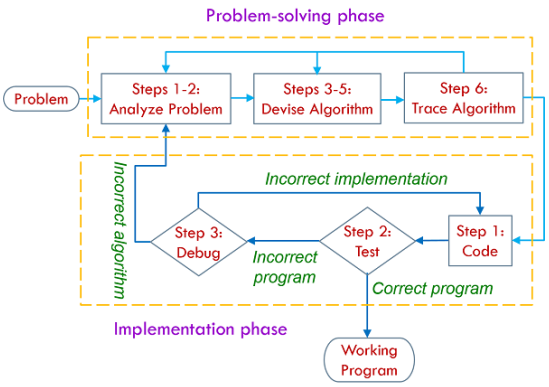
In the early days of computing, programmers used free software exclusively. Even computer companies often distributed free software. By the 1980s, almost all software was proprietary, which means that it had owners who forbade and prevented cooperation among users. Every computer user needs an operating system; if there is no free operating system, then you can't even get started using a computer without resorting to proprietary software. This was the motivation for the launch of The GNU Project by Free Software Foundation in 1983 - to develop a complete, free Unix-like GNU operating system. A Unix-like operating system includes a kernel, compilers, editors, text formatters, mail software, graphical interfaces, libraries, games and many other things. Thus, writing a whole operating system is a very large job. By 1990, the GNU Project had either found or written all the major components except one - the kernel. Then Linux, a Unix-like kernel, was developed by Linus Torvalds in 1991 and made free software in 1992. Combining Linux with the almost-complete GNU system resulted in a complete operating system: the GNU/Linux system and more commonly simply referred to as Linux or Linux distro [short for *distribution*]. Every desktop in DigiPen labs has a Linux distribution from ubuntu.

In the process of building the GNU system, a number of free and open-source programming tools were incubated including the GNU C compiler (GCC). Later a C++ compiler was developed and translators and compilers for languages such as Ada, Fortran, Objective-C, and Objective-C++, and Go were added. The addition of languages beyond C led to GCC having the meaning *The GNU Compiler Collection*.

When building ready-to-run applications from C source code, a compiler is not sufficient; run-time libraries, an assembler, and a linker, are also needed. The whole set of these tools is called a *compiler tool chain* or *compiler driver*. This document describes the process of writing, compiling, linking, and running C programs using the GCC C compiler driver.

Compiling a simple C program

Suppose you wish to create a computer program to compute the sum of the first n terms of the series $\sum_{i=1}^{n} i$ where n is a positive natural number. For example, if the program is given value 5 as input, it would return 15 since $\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 15$. The creation of this computer program can be divided into a six-step *problem-solving phase* followed by a three-step *implementation phase*.



Suppose, after the conclusion of the six-step problem-solving phase, an algorithm SUM is devised:

$$
\begin{array}{ll}
 & \text{Input: positive natural number } n \\
 & \text{Output: } 1 + 2 + \ldots + n \\
 & \\
\textbf{Algorithm SUM} & \textbf{1. } sum = 0 \\
 & \textbf{2. } i = 1 \\
 & \textbf{3. while } (i <= n) \\
 & \textbf{4. } \quad sum = sum + i \\
 & \textbf{5. } \quad i = i + 1 \\
 & \textbf{6. endwhile} \\
 & \textbf{7. } \text{print value of } sum
\end{array}
$$

The next phase called the implementation phase will use a specific programming language to convert algorithm SUM into a computer program. This phase comprises of three steps: Code, Test, and Debug. This document is concerned with filling in the details required by a C programmer to implement the Code step. Once the executable program is created, the programmer can continue with the Test and Debug steps of the implementation phase - these steps are not discussed here.

The Code step for algorithm SUM

The Code step can be further divided into two stages: *Edit* and *Compile*. The *Edit stage* uses a text editor to create and edit a *source file* containing the implementation of algorithm SUM. The subsequent *Compile stage* involves the use of a compiler toolchain to convert the contents of source file into machine code that is stored in a file known as an *executable file* or a *binary file*.

Editing source file `sum.c`

The first process of the Code step is to create a *source file* using a text editor. A source file contains code or human readable text representing an algorithm's implementation using the syntax of a specific programming language. This act of using a text editor to create a source file and typing English-like text representing C syntax and semantics is called *editing*.

Word processors such as *Microsoft Word* are not useful and functional for editing source files because they save files in proprietary native formats incompatible with programming tools. Since writing and editing code is a fundamental job of programmers, specialized text editors called source code editors have been invented to make programmers become more productive and efficient. The list of source code editors is lengthy ranging from the simple, no-frills text editors such as *vi* on Linux and *Notepad* on Windows to complex *integrated development environments* such as *Visual Studio* on Windows. We recommend Visual Studio Code since it is free, modern, multi-platform [can be installed on macOS, Windows 10, Linux], supports almost every programming language currently in use, and provides many functionalities useful to programmers. Visual Studio Code is installed in all DigiPen labs. Begin here to get an overview of the editor.

Begin by creating a folder in Windows called `test`. Type `wsl` on the command line to switch from Windows to Linux. Using Visual Studio Code, create a file called `sum.c` in folder `test`. This is easily done by typing the following in the bash shell [note that the `$` represents the Linux shell prompt and is not a part of the command]:

```
$ code sum.c
```

Type the following C code implementing algorithm SUM in source file `sum.c`:

```
#include <stdio.h>

int main(void) {
  int n;
  printf("Enter a positive natural number: ");
  scanf("%d", &n);

  int sum = 0;
  for (int i = 1; i <= n; ++i) {
    sum += i;
  }
  printf("Sum of first %d natural numbers is: %d\n", n, i);
  return 0;
}

```

Compiling source file `sum.c`

Compilation refers to the process of converting a program from the textual source code, in a programming language such as C or C++, into machine code. This machine code is then stored in a file known as an *executable file*, sometimes referred to as a *binary file*. In fact, compilation is an umbrella term that represents all of the individual stages of the compilation process. The complete set of tools used in the compilation process is referred to as a *compiler driver* or *compiler toolchain*.

To compile the file `sum.c` with GNU C compiler, use the following command:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror
sum.c -o sum.out
```

This command will compile source code in `sum.c` to machine code and store it in an executable file `sum.out`. The output file for the machine code is specified using `-o` option. If the `-o` option is omitted, output is written to a default file called `a.out`.

Executing binary file `sum.out`

To run executable program `sum.out`, type the executable's pathname like this:

```
$ ./sum.out
Enter a positive natural number: 100
Sum of first 100 natural numbers is: 5050
```

The loader program in Linux will load executable file `sum.out` from disk to memory and cause the CPU to begin executing the first instruction in function `main`.

Why does Linux require pathname `./sum.out` and not the simpler and straightforward `sum.out`? When the name of an executable such as `code` or `gcc` is typed in the shell, the operating system will search in directories specified in a variable called `PATH`. In fact, these directories can be displayed by typing the command `echo $PATH`. If the plain `sum.out` is typed, `command not found` is displayed by the shell since directory `test` [in which executable `sum.out` was created] is not specified in variable `PATH`. In every shell, character `.` on the command line means *current directory*. So, by typing `./sum.out`, you're telling Linux "*don't worry about directories in variable* `PATH`*, just run executable* `sum.out` *in the current directory* `.`".

Compiler options and dealing with errors

The compiler driver will catch straightforward syntax errors. To demonstrate this, edit source file `sum.c` with a missing `;` on line 10:

```
#include <stdio.h>

int main(void) {
  int n;
  printf("Enter a positive natural number: ");
  scanf("%d", &n);

  int sum = 0;
  for (int i = 1; i <= n; ++i) {
    sum += i
  }
  printf("Sum of first %d natural numbers is: %d\n", n, i);
  return 0;
}

```

Compiling `sum.c` with no options other than enforcing C11 [for brevity], the compiler prints the following error message:

```
$ gcc -std=c11 sum.c -o sum.out
sum.c: In function ‘main’:
sum.c:10:13: error: expected ‘;’ before ‘}’ token
   10 |     sum += i
      |             ^
      |             ;
   11 |   }
      |   ~

```

In addition to flagging incorrect syntax as errors, compilers provide many diagnostic warning messages about potential coding errors. To demonstrate this, a subtle error is introduced in line 12: in function `printf`, the correct integer format specifier `%d` is replaced with incorrect floating-point format specifier `%f`:

```
#include <stdio.h>

int main(void) {
  int n;
```

```
  printf("Enter a positive natural number: ");
  scanf("%d", &n);

  int sum = 0;
  for (int i = 1; i <= n; ++i) {
    sum += i;
  }
  printf("Sum of first %d natural numbers is: %f\n", n, sum);
  return 0;
}

```

Compiling the now buggy source file `sum.c` with no options other than enforcing C11, the following diagnostic warning is produced by the compiler:

```
$ gcc -std=c11 sum.c -o sum.out
sum.c: In function ‘main’:
sum.c:12:48: warning: format ‘%f’ expects argument of type ‘double’, but
argument 3 has type ‘int’ [-Wformat=]
   12 |   printf("Sum of first %d natural numbers is: %f\n", n, sum);
      |                                                ~^        ~~~
      |                                                 |        |
      |                                                 double   int
      |                                                %d

```

Notice that the compiler does create executable `sum.out`. To prevent programmers from passing off buggy software, `gcc` provides the `-Werror` option that prevents `gcc` from successful compilation when diagnostic warnings are generated:

```
$ gcc -std=c11 sum.c -o sum.out
sum.c: In function ‘main’:
sum.c:12:48: error: format ‘%f’ expects argument of type ‘double’, but
argument 3 has type ‘int’ [-Werror=format=]
   12 |   printf("Sum of first %d natural numbers is: %f\n", n, sum);
      |                                                ~^        ~~~
      |                                                 |        |
      |                                                 double   int
      |                                                %d
cc1: all warnings being treated as errors

```

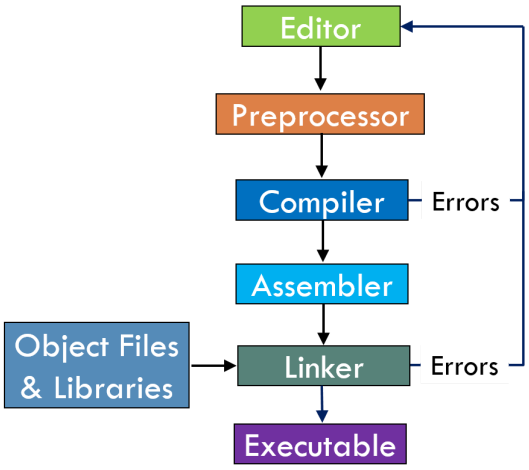
In other words, `-Werror` option converts diagnostic warnings to full blown errors and prevents programmers from proceeding any further until these warnings are heeded by repairing source code. Options `-Wall` and `-Wextra` turn on warning messages for many other common coding errors listed here. What do these compilation options such as `-std-c11`, `-pedantic-errors`, and `-Wstrict-prototypes` mean? These options support the creation of compliant code and a detailed explanation of these compilation options is provided in this section. Finding bugs is hard and programmers appreciate when compilers provide varied options that flag potential bugs by generating warnings. Therefore, it is important for students to use these options, as in:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror
sum.c -o sum.out
```

before submitting any code that will be used in assessments for this course. Non-compliance could result in submitted code receiving zero points and/or failing grade.

How the compiler works

This section describes in more detail how source files are transformed to an executable file. Compilation is a multi-stage process involving several tools, including the compiler itself such as `gcc`, the assembler `as`, and the linker `ld`. The complete set of tools used in the compilation process is known as the *compiler toolchain* or *compiler driver*. The sequence of commands executed by a single invocation of `gcc` consists of preprocessing, compilation proper, assembly, and linking stages:



As an example, the individual compilation stages will be examined using source file `hello.c`:

```
1  #include <stdio.h>
2
3  int main(void) {
4    int year = 2020;
5    printf("Hello World %d!!!\n", year);
6    return 0;
7  }
8
```

Although the code in source file `hello.c` is simple, it uses external header files [on line 1] and calls a C standard library function (on line 5), and therefore exercises the entire compiler toolchain.

The Preprocessor

The first stage of the compilation toolchain process is the use of the *preprocessor*. Think of the preprocessor as a text editor that modifies a C source file according to preprocessing directives. Before interpreting directives, the preprocessor performs a more basic global transformation on the source file: all single-line and multi-line comments are replaced with single spaces.

Preprocessing directives are lines in a source file that begin with character `#`. The `#` is followed by an identifier that is the *directive name*. The `include` directive name involves inclusion of header files. The `define` directive name involves macro expansion where a *macro* is a fragment of C code which has been given a name. Directive names `if`, `else`, and `elif` allow conditional compilation of the source file by allowing certain parts of the source file to be included or excluded from the compilation process. This document is only concerned with `include` directive name.

A *header file* is a file containing C declarations and macro definitions to be shared between multiple source files. A programmer requests the use of a header file in a source file with preprocessing directive `#include`. Line 1 of source file `hello.c` looks like this: `#include <stdio.h>`. The author is asking the preprocessor to search for C standard library header file `stdio.h` in the disk and then to replace line 1 with the contents of header file `stdio.h`. The delimiters `<` and `>` indicate to the preprocessor that it must search the standard list of system directories special to the compiler toolchain being used.

This stage of the compiler toolchain can be manually invoked like this:

```
$ gcc -std=c11 -E hello.c -o hello.i
```

In `gcc`, the output file is specified with the `-o` option. The resulting file `hello.i` is a transformation of source file `hello.c` with comments replaced by a single space and line 1 of source file `hello.c` replaced with contents of system header file `stdio.h` followed by the remaining lines of the source file. This is confirmed by examining the contents of `hello.i`:

```
// hundreds of lines omitted for brevity ...

extern int printf (const char *__restrict __format, ...);

// many hundereds of lines omitted for brevity ...

# 3 "hello.c"
int main(void) {
  int year = 2020;
  printf("Hello World %d!!!\n", year);
  return 0;
}

```

Notice that the purpose of including header file `stdio.h` is served because `hello.i` now contains a function prototype of `printf` on line 3 so that the compiler proper can understand the call to function `printf` on line 9. This is so that the cardinal rule in C and C++ that *all names must be declared before their first use* is satisfied.

The Compiler proper

The next stage of the compiling toolchain process is the actual compilation of preprocessed source code to assembly language, for a specific CPU. The command-line option `-S` instructs `gcc` to only convert preprocessed C source code in file `hello.i` to assembly language:

```
$ gcc -std=c11 -S hello.i
```

By default, the resulting assembly language is stored in file `hello.s`. A partial listing of the assembly language for an Intel $x86$ (Pentium) CPU looks like this:

```
1  // many lines deleted for brevity ...
2  main:
3  .LFB0:
4    .cfi_startproc
5    endbr64
6    pushq %rbp
7    // many lines deleted for brevity ...
8    subq  $16, %rsp
9    movl  $10, -4(%rbp)
10   movl  -4(%rbp), %eax
11   movl  %eax, %esi
12   leaq  .LC0(%rip), %rdi
13   movl  $0, %eax
14   call  printf@PLT
15   movl  $0, %eax
16   leave
17   .cfi_def_cfa 7, 8
18   ret
19   .cfi_endproc
20 .LFE0:
21 // many lines deleted for brevity ...
22
```

Notice that line 14 of the assembly language code above contains a call to C standard library function `printf`.

The Assembler

The *assembler* is a translator that transforms assembly language code into machine code and generates an *object file*. When there are calls to external functions in the assembly source file - as with line 14 in the assembly language code - the assembler leaves the addresses of external functions undefined, to be filled in later by the linker. The assembler is invoked with `-c` option of `gcc`:

```
1  $ gcc -c hello.s -o hello.o
```

or by directly calling the assembler:

```
1  $ as hello.s -o hello.o
```

The resulting object file `hello.o` contains the machine instructions for the source code in file `hello.c`, with an undefined reference to `printf`. Unlike other intermediate files generated by `gcc`, object file `hello.o` is a binary file and is therefore not human-readable.

The Linker

The final stage of compilation is to use the `ld` program to link object files to create an *executable file* or *binary file*. In practice, an executable requires many external functions from system and C run-time libraries. Consequently, the actual link
commands used internally by GCC are complicated:

```
/usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper -plugin-opt=-fresolution=/tmp/ccPZPAYK.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro -o hello.out /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/9/crtbeginS.o -L/usr/lib/gcc/x86_64-linux-gnu/9 -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/9/../../.. hello.o -lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -lgcc_s --pop-state /usr/lib/gcc/x86_64-linux-gnu/9/crtendS.o /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crtn.o
```

Fortunately there is never any need to explicitly use the `ld` command directly. `gcc` can transparently handle the entire linking process, as in:

```
$ gcc hello.o
```

This links object file `hello.o` with the C standard library. That is, it takes the `printf` function and other dependent functions from the C standard library, and the machine language version of `main` function defined in object file `hello.o` into an executable file `a.out`. The program can be executed by using the pathname of the executable `a.out`, as in:

```
$ ./a.out
Hello World 2020!!!

```

Alternatively, you can use the `-o` option to provide an other name than `a.out` to the executable:

```
$ gcc hello.o -o hello.out
```

Linking with external libraries

A *library* is a collection of precompiled object files which can be linked into programs. Libraries are typically stored in special archive files with extension `.a`, referred to as *static libraries*. They are created from object files with a separate tool, the GNU archiver `ar`, and used by the linker to resolve references to functions at compile-time. In the Linux distro on DigiPen lab desktops, the C standard library containing functions specified in C11 standard such as `printf` is located at `/usr/lib/x86_64-linux-gnu/libc.a`. By default, `libc.a` is linked by `gcc`. For historical reasons,

the math portion of C standard library is stored in a separate static library `/usr/lib/x86_64-linux-gnu/libm.a`. The functions in the math library are declared in header file `<math.h>`. However, again for historical reasons, `gcc` does not automatically link `libm.a`.

Consider the following source file `sin.c` that makes a call to external function `sin` in math library `libm.a`:

```
#include <stdio.h>
#include <math.h>

int main(void) {
  double angle = 0.785398; // 45 degrees in radians
  printf("sin(%f) = %f\n", angle, sin(angle));
  return 0;
}

```

Source file `sin.c` compiles without a hiccup:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c
sin.c -o sin.o
```

However, the linker throws an error when an attempt is made to generate an executable:

```
$ gcc sin.o -o sin.out
/usr/bin/ld: sin.o: in function `main':
sin.c:(.text+0x23): undefined reference to `sin'
collect2: error: ld returned 1 exit status

```

The problem is that the reference to function `sin` is neither defined in source file `sin.c` nor in the default C standard library `libc.a`. Instead, it is defined in external math library `libm.a` and the compiler does not link to file `libm.a` unless it is explicitly selected. This is done by using the `-lm` option to the linker stage of `gcc`:

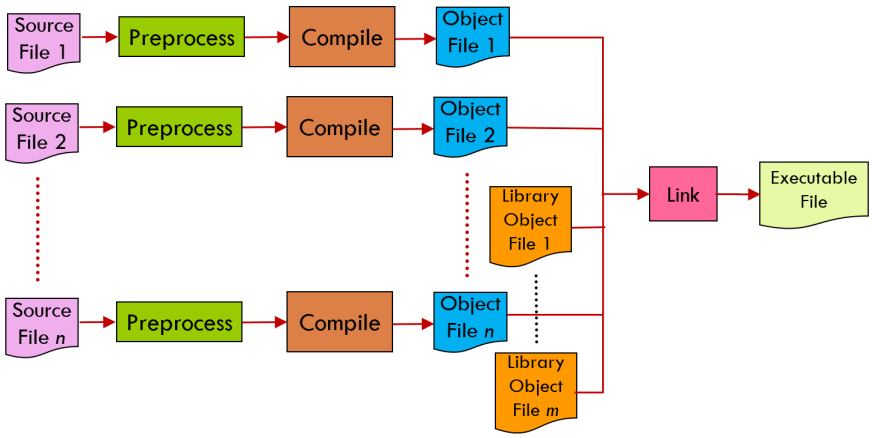
```
$ gcc sin.o -o sin.out -lm
```

Option `-lm` is a shorthand for link object files with a library file `/usr/lib/x86_64-linux-gnu/libm.a`.

Compiling multiple source files

Software development is a group activity with many programmers collaborating and working together to create a software application. Editing a single monolithic source file is not only cumbersome but also error-prone. Instead, common practice is to divide the program into sub-programs with each sub-program further divided into functions. Related functions are grouped in a source file and assigned to a specific programmer. This division of labor and purpose through multiple source files makes it easier and faster to design, implement, and test software because individual programmers can independently implement, compile, and test their source files. In fact, an individual programmer will never need access to the source code of other programmers to

create the executable. The following picture illustrates the process by which multiple source files and external libraries are combined to generate an executable.



In the following example, the implementation of algorithm SUM is divided into two source files `main.c` and `sum_fn.c` and a header file `sum.h`. The division of tasks to two programmers consists of a client programmer implementing `main.c` that uses the implementation of algorithm SUM and a second programmer independently implementing algorithm SUM in source file `sum_fn.c` and also providing an interface or header file `sum.h`. Neither programmer is aware of nor has access to the other programmer's source files [note that header files are by design supposed to be shared].

Here is the implementation of driver source file `main.c` containing the definition of function `main` [recall that every C program must contain *one and only one* function called `main` which is the CPU's entry point to the program] by the client programmer:

```
1  #include <stdio.h>
2  #include "sum.h"
3
4  int main(void) {
5    int n;
6    printf("Enter a positive natural number: ");
7    scanf("%d", &n);
8
9    printf("Sum of first %d natural numbers is: %d\n", n, sum(n));
10   return 0;
11 }
12
```

The implementation of algorithm SUM in the previous version of the program in source file `sum.c` has been replaced by a call to a new external function `sum`, which from the perspective of both the author of `main.c` and the compiler is defined *somewhere*.

Another new entry in source file `main.c` is the addition of preprocessor directive `#include "sum.h"` on line 2. This is an interface file provided by function `sum`'s author containing its function prototype. A *function prototype* specifies the function's name, its parameter list, and its return type. The compiler will use this function prototype to check for the correct use of function `sum` by the author of `main.c`. The compiler does this check by ensuring that the function call's arguments and return type match up correctly with the function definition's parameters and return type. In simpler terms, all of this means that since the compiler doesn't have access to the definition of function `sum` in file `sum_fn.c`, it will use the function prototype in header file `sum.h` to pass judgement on whether the call to function `sum` in `main.c` is correct.

Finally, notice the difference in syntax for `#include` directives in lines 1 [`#include <stdio.h>`) and 2 (`#include "sum.h"`]. The delimiters `<` and `>` indicate to the preprocessor that it must search the standard list of system directories special to the compiler toolchain being used. In the Linux distro on DigiPen lab desktops, C standard library header files are located at `/usr/include`. The delimiters `"` and `"` indicate to the preprocessor that it must search the current directory before looking in the standard list of system directories.

Suppose header file `sum.h` supplied by the implementer of file `sum_fn.c` looks like this:

```
1 int sum(int);
2
```

The header file contains a single line containing the function prototype for `sum`.

The implementer of file `main.c` now has everything required to create an object file:

```
1 $ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c
  main.c -o main.o
```

Independently, the second programmer has defined function `sum` in a separate file `sum-fn.c`:

```
1  #include "sum.h"
2
3  int sum(int N) {
4    int total = 0;
5    for (int i = 1; i <= N; ++i) {
6      total += i;
7    }
8    return total;
9  }
10
```

Notice that line 1 contains a preprocessor directive to include header file `sum.h`. This is a recommended (and sometimes necessary) practice to ensure that both the function prototype in the header file and definition in source file match up. The author of `sum-fn.c` can independently compile and generate a corresponding object file:

```
1 $ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c
  sum-fn.c -o sum_fn.o
```

Either of the two programmers or any other programmer can now link together function `main` in object file `main.o`, function `sum` in object file `sum-fn.o`, and C standard library function `printf` defined in static C standard library `libc.a` into a single executable, say `new-sum.out`:

```
1 $ gcc main.o sum-fn.o -o new-sum.out
```

To run executable program `new-sum.out`, type the pathname of the executable like this:

```
1 $ ./new-sum.out
2 Enter a positive natural number: 100
3 Sum of first 100 natural numbers is: 5050
```

Compilation flags for GCC C and Clang compilers

The aim of these compilation flags in GCC C and Clang compilers is to make the best possible attempt to ensure C code conforms as a subset of C++. Note that the Clang driver and language features are intentionally designed to be as compatible with the GCC C compiler as reasonably possible, easing migration from GCC C compiler to Clang. In most cases, code "just works". Therefore, only links to the GCC options are provided here. Information and details related to the Clang compiler can be found in the Clang compiler user's manual.

- `-std=c11`: Information about the C11 standard is available in the C11 N1570 standard draft.
- `-pedantic-errors`: Gives an error when base standard C11 requires a diagnostic message to be produced. C89 allows the declaration of a variable, function argument, or structure member to omit the type specifier, implicitly defaulting its type to `int`. Although, legal in C89, this is considered illegal in C99, C11, and C++:

```
#include <stdio.h>
int main(void) {
  static x = 10;
  printf("x: %d\n", x);
  return 0;
}

```

  However, compiling this code with a C11 compiler, as in: `gcc -std=c11 tester.c` elicits only a warning message. However, compiling the same code with the `-pedantic-errors` option produces an error.

  According to Section $5.1.1.3$ of the C11 N1570 standard draft,

> 1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.[9])
>
> 9. The intent is that an implementation should identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. It may also successfully translate an invalid program.

  Based on the above text, `-pedantic-errors` cannot be solely used to check programs for strict C conformance. The flag finds some non-standard practices, but not all - only those for which C *requires* a diagnostic, and some others for which diagnostics have been added.

- `-Wall` : This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid [or modify to prevent the warning]. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning. Some of them are enabled by `-Wextra` but many of them must be enabled individually. The entire list of warning flags enabled by `-Wall` for GCC C compiler can be found here.

- `-Wextra`: This enables some extra warning flags that are not enabled by `-Wall`. The entire list can be found here.
- `-Werror`: Converts diagnostic messages generated by the base compiler and warnings generated by flags `-Wall` and `-Wextra` to errors. This is a necessary feature for generating cleanly compiled code that doesn't generate any warning messages.
- `-Wstrict-prototypes`: It is legal in all C standards to specify a function declaration as:

```
return-type f();
```

C89, C99, and C11 compilers read the above declaration as `f` is a function that takes unknown number of parameters with unknown types and return a value of type `return-type`. This means the following code in a source file `test.c` will compile and link with undefined behavior when executed.

```
#include <stdio.h>

extern int foo();

int main(void) {
  int x = 2, y = 4, z = 6;
  printf("%d + %d = %d\n", x, y, foo(x, y));
  printf("%d + %d + %d = %d\n", x, y, z, foo(x, y, z));
  return 0;
}

int foo(int i, int j) {
  return i + j;
}

```

The code will compile with GCC C and Clang compilers (and also with Microsoft Compiler) without any diagnostic messages:

```
$ gcc -std=c11 -pedantic-errors -Wall -Wextra -Werror test.c
```

However, the same code will not compile with a C++ compiler:

```
$ g++ -std=c++11 test.c
```

since the declaration `return-type f();` in C++ compilers indicates that `f` is a function that takes no parameters and returns a value of type `return-type`.

To ensure compatibility with C++ standards, the `-Wstrict-prototypes` must be used with GCC and Clang to ensure that `test.c` doesn't successfully compile.

Make and Makefiles

The previous section has emphasized the necessity for using a variety of GCC options to ensure code is cleanly compiled without any warnings. Typing the entire set of required options each time can be annoying. When writing complex programs consisting of multiple (think tens or hundreds or thousands) source files, making small changes to a few files will require the entire set

of source files to be recompiled. These recompilations may occur hundreds of times every day causing substantial delays as programmers wait for the executable to be created. More importantly, programmers will have to remember dependencies between different files. For example, if source file `b.c` includes header file `a.h` and if `a.h` is updated, then `b.c` must be recompiled even though it was not altered.

It can be difficult to remember the entire list of source files and the dependencies required to create an executable from them. To solve this problem, a program called *make* is used. The version of *make* provided by GCC is coincidentally called `make`. *make* is a facility for automated maintenance and build executables from source files. *make* uses a *makefile* that specifies the dependencies between files and the commands that will bring all files up to date and build an executable from these up to date files. In short, *makefile* contains the following information:

- the name of source and header files comprising the program
- the interdependencies between these files
- the commands that are required to create the executable

A simple *makefile* consists of *rules* with each *rule* consisting of three parts: a *target*, a list of *prerequisite*s, and a *command*. A typical rule has the form:

```
target : prereq-1 prereq-2 ...
  command1
  command2
  ...
```

`target` is the name of the file to be created or an action to be performed by *make*. `prereq-1`, `prereq-2`, and so on represent the files that will be used as input to create `target`. If any of the prerequisites have changed more recently than `target`, then *make* will create `target` by executing commands `command1`, `command2`, and so on. *make* will terminate and shutdown if any command is unsuccessful. *Note that every command must be preceded by a tab and not spaces!!!*

Here's an example:

```
example.out : main.o file1.o file2.o
  gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror
main.c file1.c file2.c -o example.out
```

Line 1 says that target `example.out` must be remade (or made if it doesn't exist) if any of the prerequisite files [`main.o`, `file1.o`, `file2.o`] have been changed more recently than the target. Before checking the times prerequisite files were changed, *make* will look for rules that start with each prerequisite file. If such a rule is found, *make* will make the target if any of its prerequisites are newer than the target. After checking that all prerequisite files are up to date and remaking any that are not, *make* brings `example.out` up to date.

Line 2 tells *make* how it should remake target `example.out`. This involves calling `gcc` with the usual and required GCC options to compile and link source files `main.c`, `file1.c`, and `file2.c`.

A makefile can also contain *macro definitions* where a macro is simply a name for something. A macro definition has the form:

```
NAME = value
```

The value of macro `NAME` is accessed by either `$(NAME)` or `${NAME}`. *make* will replace every occurrence of either `$(NAME)` or `${NAME}` in *makefile* with `value`.

Here's a complete annotated example:

```
1  # makefile for example.out
2  # the # symbol means the rest of the line is a comment
3
4  # this is definition of macro GCC_OPTIONS
5  GCC_OPTIONS = -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -
   Werror
6  # this is definition of macro OBJS
7  OBJS = main.o file1.o file2.o
8
9  # this rule says that target example.out will be built if prerequisite files
10 # main.o file1.o file2.o file3.o have changed more recently than example.out
11 # the text $(OBJS) will be substituted with list of options in line 7
12 # the next line says to build example.out using command gcc
13 # the text $(GCC_OPTIONS) will be substituted with list of options in line 5
14 example.out : $(OBJS)
15   gcc $(GCC_OPTIONS) $(OBJS) -o example.out
16
17 # the next line says main.o depends on main.c
18 # the line after it says to create main.o with the command gcc
19 main.o : main.c
20   gcc $(GCC_OPTIONS) -c main.c -o main.o
21
22 # file1.o depends on both file1.c and file1.h
23 # and is created with command gcc $(GCC_OPTIONS) -c file1.c -o file1.o
24 file1.o : file1.c file1.h
25   gcc $(GCC_OPTIONS) -c file1.c -o file1.o
26
27 # file2.o depends on both file2.c and file1.h
28 file2.o : file2.c file1.h
29   gcc $(GCC_OPTIONS) -c file2.c -o file2.o
30
31 # clean is a target with no prerequisites;
32 # typing the command in the shell
33 # make clean
34 # will only execute the command which is to delete the object files
35 clean :
36   rm $(OBJS)
```

Target `clean` on line 35 is different from the other targets; it has no prerequisites. If the following command is issued in the shell:

```
1  $ make clean
```

then `make` will execute only the command on line 36 in rule `clean` and then exit.

Let's conclude this section by writing a simple *makefile* called `Makefile` for the `new_sum` program that consists of two source files `main.c` and `sum_fn.c` and a header file `sum.h`. The default *makefile* is named `makefile` or `Makefile`; other names can be used but `make` must be provided the non-default *makefile* name.

```
GCC_OPTIONS = -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -
Werror
OBJS = main.o sum_fn.o
EXEC = new_sum.out

$(EXEC) : $(OBJS)
  gcc $(GCC_OPTIONS) $(OBJS) -o example.out

main.o : main.c sum.h
  gcc $(GCC_OPTIONS) -c main.c -o main.o

sum_fn.o : sum_fn.c sum.h
  gcc $(GCC_OPTIONS) -c sum_fn.c -o sum_fn.o

clean:
  rm $(OBJS) $(EXEC)
```

The most common error with a *makefile* is programmers forgetting to put a horizontal tab at the beginning of a command line, and instead place space characters there. Here's what happens if line 12 is prefixed with space characters rather than a tab:

```
Makefile:12: *** missing separator.  Stop.

```