

## C Language

Request Topic

Compilation All Versions

## Examples

## File Types

7 Compiling C programs requires you to work with four kinds of files:

1. **Source files:** These files contain function definitions, and have names which end in `.c` by convention.
2. **Header files:** These files contain function prototypes and various pre-processor statements (see below). They are used to allow source code files to access externally-defined functions. Header files end in `.h` by convention.
3. **Object files:** These files are produced as the output of the compiler. They consist of function definitions in binary form, but they are not executable by themselves. Object files end in `.o` by convention, although on some operating systems (e.g. Windows, MS-DOS), they often end in `.obj`.
4. **Binary executables:** These are produced as the output of a program called a "linker". The linker links together a number of object files to produce a binary file which can be directly executed. Binary executables have no special suffix on Unix operating systems, although they generally end in `.exe` on Windows.
5. **Libraries:** A library is a compiled binary but is not in itself an executable (i.e., there is no `main()` in a library). A library contains functions that may be used by more than one program. A library should ship with header files which contain prototypes for all functions in the library; these header files should be referenced (i.e. `#include`) in any source file that uses the library. The linker then needs to be referred to the library so the program can successfully compile. There are two types of libraries: static and dynamic.
  - **Static library:** A static library (`.a` files for Unix systems and `.lib` files for Windows) is statically built into the program.
  - **Dynamic library:** A dynamic library (`.so` files for Unix systems and `.dll` files for Windows) is dynamically linked at runtime by the program. These are also sometimes referred to as shared libraries because one library can be shared by many programs. Dynamic libraries have the advantage taking up less disk space if more than one application is using the library. Static libraries have the advantage of the program knowing exactly which version of a library the program is using.

edited Jul 27 at 12:26



## The Compiler

Improvements requested by [vasili111](#), [gdc](#):

6 After the C pre-processor has included all the header files and expanded all statements, the compiler can compile the program. It does this by turning the C source code into an object code file, which is a file ending in `.o` which contains the binary version of the source code. Object code is not directly executable, though. In order to make an executable, you also have to add code for all of the library functions that were `#include` d into the file (this is not the same as including the declarations, which is what `#include` does). This is the job of the linker (see the next section).

In general, the compiler is invoked as follows:

```
% gcc -c foo.c
```

where % is the UNIX prompt. This tells the compiler to run the pre-processor on the file `foo.c` and then compile it into the object code file `foo.o`. The `-c` option means to compile the source code file into an object file but not to invoke the linker. If your entire program is in one source code file, you can instead do this:

```
% gcc foo.c -o foo
```

This tells the compiler to run the pre-processor on `foo.c`, compile it and then link it to create an executable called `foo`. The `-o` option states that the next word on the line is the name of the binary executable file (program). If you don't specify the `-o`, (if you just type `gcc foo.c`), the executable will be named `a.out` for historical reasons.


In general the compiler takes four steps when converting a `.c` file into an executable: pre-processing, compilation, assembly, and linkage. The pre-processing step optimizes your existing `.c` file. The compilation

step converts the program into assembly, you can stop the compiler at this step by adding the '-S' option. Assembly converts the assembly into machine code. Finally, the linkage step links the object code to external libraries to create an executable.

Note also that the name of the compiler we are using is GCC, which stands for both "GNU C compiler" and "GNU compiler collection", depending on context. (G++ is part of GCC the compiler collection, for example.) Other C compilers exist; many of them have the name `cc`, for "C compiler". On Linux systems `cc` is often an alias for GCC.

The POSIX standards currently mandate `c99` as the name of the C compiler — it supports the C99 standard by default. Earlier versions of POSIX mandated `c89` as the compiler.

edited Jul 29 at 16:40



The Linker

Improvements requested by [vasili111](#):

- 6
- The job of the linker is to link together a bunch of object files ( `.o` files) into a binary executable. The process of *linking* mainly involves *resolving symbolic addresses to numerical addresses*. The result of the link process is normally an executable program.
- During the link process, the linker will pick up all the object modules specified on the command line, add some system-specific *startup code* in front and try to resolve all *external* references in the object module with *external definitions* in other object files (object files can be specified directly on the command line or may implicitly be added through libraries). It will then assign *load addresses* for the object files, that is, it specifies where the code and data will end up in the address space of the finished program. Once it's got the load addresses, it can replace all the symbolic addresses in the object code with "real", numerical addresses in the target's address space. The program is ready to be executed now.
- This includes both the object files that the compiler created from your source code files as well as object files that have been pre-compiled for you and collected into library files. These files have names which end in `.a` or `.so`, and you normally don't need to know about them, as the linker knows where most of them are located and will link them in automatically as needed.
- Like the pre-processor, the linker is a separate program called `ld`. Also like the pre-processor, the linker is invoked automatically for you when you use the compiler. The normal way of using the linker is as follows:

```
% gcc foo.o bar.o baz.o -o myprog
```


This line tells the compiler to link together three object files ( `foo.o`, `bar.o`, and `baz.o` ) into a binary executable file named `myprog`. Now you have a file called `myprog` that you can run and which will hopefully do something cool and/or useful.

This is all you need to know to begin compiling your own C programs. Generally, we also recommend that you use the `-Wall` command-line option:

```
% gcc -Wall -c foo.cc
```

The `-Wall` option causes the compiler to warn you about legal but dubious code constructs, and will help you catch a lot of bugs very early.

edited yesterday



4

The Pre-processor

 [Add Example](#)

Syntax

Parameters

Remarks

Filename extension	Description
.c	Source file. Contains code.
.h	Header file. Contains code.
.o	Object file. Compiled source file(s) and/or header file(s).
.obj	Alternative extension for object files.

.a	Library file. Package of object files.
.dll	Dynamic-Link Library on Windows.
.so	Shared object (library) on many Unix-like systems.
.exe , .com	Windows executable file. Formed by linking object files and library files. In Unix-like systems, there is no special file name extension for executable file.

POSIX c99 compiler flags	Description
-o filename	Output file name eg. ( bin/program.exe , program )
-I directory	search for headers in direrctory .
-D name	define macro name
-L directory	search for libraries in directory .
-l name	link library libname .

Compilers on POSIX platforms (Linux, mainframes, Mac) usually accept these options, even if they are not called c99 .

- See also [c99 - compile standard C programs](#)

GCC (GNU Compiler Collection) Flags	Description
-Wall	Enables all warning messages that are commonly accepted to be useful.
-Wextra	Enables more warning messages, can be too noisy.
-pedantic	Force warnings where code violates the chosen standard.
-Wconversion	Enable warnings on implicit conversion, use with care.
-c	Compiles source files without linking.
-v	Prints compilation info.

- gcc accepts the POSIX flags plus a lot of others.
- Many other compilers on POSIX platforms ( clang , vendor specific compilers) also use the flags that are listed above.
- See also [Invoking GCC](#) for many more options.

TCC (Tiny C Compiler) Flags	Description
-Wimplicit-function-declaration	Warn about implicit function declaration.
-Wunsupported	Warn about unsupported GCC features that are ignored by TCC.
-Wwrite-strings	Make string constants be of type const char * instead of char *.
-Werror	Abort compilation if warnings are issued.
-Wall	Activate all warnings, except -Werror , -Wunusupported and -Wwrite strings .

edited yesterday

 +6

Still have question about Compilation?

Ask Question