Preprocessing

/run/media/lijunjie/资料/Learning Video/writing-running-fixing-code/02_compiling-and-running/01_introduction/04_preprocessing_instructions.html

The first step, in the upper left is the *preprocessor* , which takes your C source file and combines it with any *header files* that it includes, as well as expanding any *macros* that you might have used. To help understand this process, we will look at our first complete C program, which is algorithmically simple (all it does is print *Hello World*), but useful for explaining the compilation process.

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

int main (void) {
  printf ("Hello World\n");
  return EXIT_SUCCESS;
}
```

While the main function's contents should be mostly familiar—a call to *printf* , and a **return** statement—there are several new concepts in this program. The first two lines are *include* **directives** . These lines of code are not actually statements which are executed when the program is run, but rather directives to the preprocessor portion of the compiler. In particular, these directives tell the preprocessor to literally include the contents of the named file at that point in the program source, before passing it on to the later steps of the compilation process.

Header Files

These *#include* directives name the file to be included in angle brackets (<>) because that file is one of the standard C header files. If you wrote your own header file, you would include it by placing its name in quotation marks (*e.g.* , *#include "myHeader.h* "). (This is not a formal rule, but a very common convention.) Preprocessor directives begin with a pound sign (#) and have their own syntax.

In this particular program, there are two include directives. The first of these directs the preprocessor to include the file stdio.h and the second directs it to include stdlib.h. These header files—and header files in general—primarily contain three things: *function prototypes* , macro definitions, and type declarations.

A function prototype looks much like a function definition, except that it has a semicolon in place of the body. The prototype tells the compiler that the function exists somewhere in the program, as well as the return and argument types of the function. Providing the prototype allows the compiler to check that the correct number and type of arguments are passed to the function, and that the return value is used correctly without having the

entire function definition available. In the case of printf, stdio.h provides the prototype. The actual implementation of printf is inside the C standard library, which we will discuss further when we learn about the linker later.

Macros

Header files may also contain macro definitions. The simplest use of a macro definition is to define a constant, such as

```
#define EXIT_SUCCESS 0
```

This directive (from stdlib.h) tells the preprocessor to define the symbol EXIT_SUCCESS to be 0. Whenever the preprocessor encounters the symbol EXIT_SUCCESS, it sees that it is defined as a macro and expands the macro to its definition. In this case, the definition is just 0, so the preprocessor replaces EXIT_SUCCESS with 0 in the source it passes on to the later stages of compilation. Note that the preprocessor splits the input into identifiers, and checks each identifier to see if it is a defined macro, so EXIT_SUCCESS_42 will NOT expand to 0_42, but rather will not be considered a macro and preprocessor will leave it alone (unless it is defined elsewhere).

Using macro definitions for constants provides a variety of advantages to the programmer over writing the numerical constant directly. For one, if the programmer ever needs to change the constant, only the macro definition must be changed, rather than all of the places where the constant is used. Another advantage is that naming the constant makes the code more readable. The naming of the constant in **return** EXIT_SUCCESS gives you a clue that the return value here indicates that the program succeeded. In fact, this is exactly what this statement does. The return value from main indicates the success or failure of your program to whatever program ran it.

A third advantage of using macro defined constants is *portability* . While 0 may indicate success on your platform—the combination of the type of hardware and the operating system you have—it may mean failure on some other platform. If you hardcode the constant 0 into your code, then it may be correct on your platform, but may need to be rewritten to work correctly on another platform. By contrast, if you use the constants defined in the standard header files, then when you recompile your program on the new platform, it will just work correctly—the header files on that platform will have those constants defined to the correct values.

You will learn more about macros later, after you write and run a simple "Hello World" program.