CS 240 Programming in C

Macro, Preprocessor, Header Files and Library

February 22, 2024

Schedule

1. Macro
2. Header Files

Macro

What is a Macro

- A macro is a fragment of code that has been given a name. Whenever the name is used, it is replaced by the contents of the macro.
- A macro is defined by a directive command and is processed by a **preprocessor**.
- The C Preprocessor is not a part of the compiler but is a separate step in the compilation process.
- The C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation
- To see the preprocessing results we can use the -E option for gcc

```
[cs240] > gcc -E main.c
```

Example

```
#define PI 3.1415926
#include <stdio.h>
```

Preprocessor

Sr.No.	Directive & Description
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif #else and #if in one statement.
9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standar

Macro Substitution

- Simplest Form:

```
#define name {replacement text}
```

1. Literally the token name will be replaced by the replacement text
2. Substitutions are made only for tokens and do not take place within quoted strings or sub-strings.

```
   For example, if YES is a defined name,
there would be no substitution in printf("YES")
or in YESMAN.
```

Example

```
#define GREETINGS printf("HELLO\n");
#define PI 3.1415926
```

Macro Substitution

- Define macros with arguments. Form:

  `#define name(A, B) [replacement text with A and B]`

  1. The token name will be replaced by the replacement text containing its arguments.
  2. Compare to an equivalent function call, the macro saves the overhead for creating and destroying the stack frame for the function.
  3. However you have to be careful about what happens to compound expressions. It may produce a result that is not what you intended.
  4. (Cure: use () as many as possible to group together the arguments)
  5. Macro name can be undefined with undef such that the name is free to for other use

Example

```
#define max(A, B) ((A) > (B) ? \
                           (A) : (B))
#define square_1(x)  x * x
#define square(x) (x) * (x)
```

Macro Substitution

- # operator on arguments
  1. A parameter name of Macro is preceded by the # in the replacement text, the combination will be expanded into a quoted string with the parameter replaced by the actual argument.
- ## operator on arguments
  1. Actually concatenate two arguments

Macro Substitution

Example

```
int main(){
        // # opereator
        #define dprint(expr) printf(#expr " = %g\n", expr)
        dprint(5.0/2);
        printf("5.0/2" " = %g\n", 5.0/2);
        // equivalent to printf("x/y" " = %g\n", x/y);
        // And the two strings get concatenated or this way

        #define dprint1(expr) printf( "%s = %g\n",#expr, expr)
        dprint1(5.0/2);

        // ## operator
        #define concat(front, second) front ## second
        char *concat(name,1) = "Yeah";// define name1 = "Yeah"

        // char *name1 = "Yeah";
        printf("%s\n", name1);
 }
```

Multiple Line Macro

The macro continuation (\) operator is used to continue a macro that is too long for a single line.

Predefined Macros

Sr.No.	Macro & Description
1	__DATE__ The current date as a character literal in "MMM DD YYYY" format.
2	__TIME__ The current time as a character literal in "HH:MM:SS" format.
3	__FILE__ This contains the current filename as a string literal.
4	__LINE__ This contains the current line number as a decimal constant.
5	__STDC__ Defined as 1 when the compiler complies with the ANSI standard.

Conditional Pro-processing

- The header guards is a conditional preprocessing that uses the directives of

```
#ifndef MATH_H  // not defined and then
#define MATH_H

#endif
```

- There is also **#ifdef** is used for preprocessing the code differently.

```
#ifdef DEBUG
    // C code
#endif
```

Compile Time Macros for Development

- The compiler option -D for gcc can be used to define compile-time macros in code, which can be used for preprocessing our code differently
- And the "DEBUG" keyword is widely used as one of compile time macro names. Of course, you can use any other keyword you like as long as not causing conflict.
- And you can use more than one compile-time Macros
- Remember the option -E is used to produce pro-processed code.

Compile Time Macros for Development

```c
extern int printf (const char *__restrict __format, ...);
int main(int agrc, char *argv[]){

        #ifdef DEBUG
        printf("Compiling time macro DEBUG is being defined");
        printf("%d\n", DEBUG);
        #endif

        #ifdef TESTING
        printf("Compiling time macro TESTING is being defined");
        printf("%d\n", TESTING);
        #endif

        #if defined(DEBUG) && defined(TESTING)
        printf("DEBUG and TESTING are both defined\n");
        #endif

        return 0;
}
```

Compiling process

Stages

Main compiling stages

source code (main.c) $\rightarrow$ preprocessor (main.c with header files included) $\rightarrow$ compiler $\rightarrow$ assembler $\rightarrow$ object code (main.o) $\rightarrow$ linker $\rightarrow$ executable (a.out)

- The header file is needed for the preprocessing stages which offer the macros and declarations of functions or variables that are used within your code. Like the printf.

Declaration vs Definition

- A declaration introduces an identifier and describes its type, be it a type, object, or function. A declaration is what the compiler needs to accept references to that identifier.
- A definition actually instantiates/implements this identifier. It's what the linker needs in order to link references to those entities.
- A declaration is a description of a type (shape holder) and the definition is the actual function or data (the thing with that shape)
- In the C source code, there can be multiple declarations and only 1 definition of any function or variable. If there are multiple definitions of a function or variable the compiler will panic and stop running.

Header Files

Header files

- A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.
- There are two types of header files: the files that the programmer writes and the files that come with your compiler (means on the searching PATH for gcc).
- To include a header file of another library in your source code using the preprocessing directive **#include ""** (relative path) and **#include <>** (on the searching path)
- What the include header file does is literally place all the contents at the line of the include directive, such that your source code can use the declarations in that library
- But how does the preprocessor find the header files that are needed

Example

```c
// declaration of printf without the function body
extern int printf (const char *__restrict __format, ...);

// definition of the main function with a function body
int main(int agrc, char *argv[]){

        #ifdef DEBUG
        printf("Compiling time macro DEBUG is being defined");
        printf("%d\n", DEBUG);
        #endif

        #ifdef TESTING
        printf("Compiling time macro TESTING is being defined");
        printf("%d\n", TESTING);
        #endif

        #if defined(DEBUG) && defined(TESTING)
        printf("DEBUG and TESTING are both defined\n");
        #endif
```

Header File Search Path

- Reference (`https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html`)
- By default, the preprocessor looks for header files included by the quote form of the directive include "file" first relative to the directory of the current file
- and then in a preconfigured list of standard system directories. For example, if /usr/include/sys/stat.h contains include "types.h", GCC looks for types.h first in /usr/include/sys, then in its usual search path.
- For the angle-bracket form include <file>, the preprocessor's default behavior is to look only in the standard system directories. The exact search directory list depends on the target system, how GCC is configured, and where it is installed.
- You can find the default search directory list for your version of gcc this

```
echo | gcc -x c -E -Wp,-v -
```

Header File Search Path

- You can find the default search directory list for your version of gcc this

```
echo | gcc -x c -E -Wp,-v -
```

- And apparently, the header files for the standard library libc will be within one of these paths, like the stdio.h
- You can use the -Idir option for gcc to add a custom path for searching the header files
- Reference (`https://web.mit.edu/rhel-doc/3/rhel-gcc-en-3/directory-options.html`)
- Or use the env variable

```
use CPATH (c or c++), C\_INCLUDE\_PATH (c only),
  CPLUS\_INCLUDE\_PATH (c++ only).
  For example:
  CPATH=/home/allen/headers gcc -c foo.c -o foo.o
```

- Do some demos.

Object Library File

```
gcc -c main.c
```

- An object file is the real output from the compilation phase. It's mostly machine code, but has info that allows a linker to see what symbols are in it as well as symbols it requires in order to work.
- As object code offers the machine code of the definitions of functions and global variables of a source code.
- They can be compiled with other C source codes which utilize some functions and variable definitions of them.
- Object files are usually bundled together into a **library** file which is easy to distribute like the C standard library or libc.
- There are two types of libraries: static library (in Linux .a, in Window .lib) and dynamic library (in Linux .so, in Windows .dll).

Library Object File Search Path

- You can find the default search directory list for your version of gcc this

```
gcc -print-search-dirs | tr : "\n"
```

or

```
ld --verbose | grep SEARCH_DIR | tr -s ' ;' "\n"
```

- And apparently, the object files for the standard library libc will be within one of these paths, like the stdio.a
- You can use the -L<dir_path> option for gcc to add a custom path for searching the library files
- Or use the env variable

use LIBRARY_PATH

For example:

```
LIBRARY_PATH=/home/allen/lib gcc -c foo.c -o foo.o
```

- Do some demos.

Linking

Create a client.c source code that utilize the function in our libmylib.a

```
gcc client.c -o client -L. -lmylib
```

- The "-L." is used to tell that the static library is in the current folder
- The "-lmylib" is used to indicate to link libmylib.a library.

-lc -lm

-lc is for link libc and -lm is for link libm.

libc and libm are both in system lib path however libc is by default linked, but libm is not. That's why when we use math lib functions we have to use -lm option for linking the math library.

Naming Convention: Check this link
`https://stackoverflow.com/questions/1033898/why-do-you-have-to-link-the-math-library-in-c`

Overview

- Preprocess

  Method 1:
  use CPATH (c or c++), C_INCLUDE_PATH (c only),
  CPLUS_INCLUDE_PATH (c++ only).
  For example:
  CPATH=/home/allen/my_libs_headers gcc -c foo.c -o foo.o.
  Method 2:
  use -I<dir_path> to gcc when compiling.
  For example: gcc -I/home/allen/my_libs_headers -c foo.c -o foo.

- Linking

  Method 1:
  To add custom directories to the library linking
  Search directories list, use LIBRARY_PATH environment variable.
  For example: LIBRARY_PATH=./mylib:/home/allen/my_libs gcc foo.o
  Method 2:
  add the flag -L<dir_path> to gcc when linking.
  For example:
  gcc -L/home/allen/my_libs -L./mylib foo.o bar.o -o foo.

Using a C library

- To use a custom Library, we need two parts which are the headers and objects.
- And the headers and objects need to be on the gcc searching path respectively.
- We can manually do all these through command line commands like those shown above.
- But usually for a large project, we need a build system like make or CMake to manage all these. And of course, a build system can manage many other things.

Creating a static C library bundle

Creating a static Library bundle

- A library is basically just an archive of object files.
- You can use ar command to bundle them together

```
[~] ar rcs libmylib.a objfile1.o objfile2.o objfile3.o
# the lib[name].a is a name convention for creating
# static library
```

- When linking your program to the libraries, make sure you specify where the library can be found

```
[~]   gcc -o foo foo.o -L. -lmylib
```

- The -L. piece tells gcc to look in the current directory in addition to the other library directories for finding libmylib.a.
- Of course, when you compile your foo.c into foo.o you have to specify the included header files

Linking

-l option

- Because of the naming convention that -l will link a lib[name].a file like -l[name], otherwise will not recognize it.

```
allen@DESKTOP-UV2S8G7 MINGW64 ~/CLionProjects/cs240/src/lib/src (2023-fall)
$ ar rcs libcs240.a utils.o
allen@DESKTOP-UV2S8G7 MINGW64 ~/CLionProjects/cs240/src/lib/src (2023-fall)
$ dir
libcs240.a  utils.c  utils.o
allen@DESKTOP-UV2S8G7 MINGW64 ~/CLionProjects/cs240/src/clients/using_lib (2023-fall)
$ CPATH=../../lib/include   gcc  main.c -L../../lib/src/ -lcs240
allen@DESKTOP-UV2S8G7 MINGW64 ~/CLionProjects/cs240/src/clients/using_lib (2023-fall)
$ ./main.out
Hello 2 3.14169
1 add 2 3
```

- But if we really want to link the .o file itself, we can use -l:[utils].o

```
allen@DESKTOP-UV2S8G7 MINGW64 ~/CLionProjects/cs240/src/clients/using_lib (2023-fall)
$ CPATH=../../lib/include   gcc  main.c -L../../lib/src/ -l:utils.o -o main.out
allen@DESKTOP-UV2S8G7 MINGW64 ~/CLionProjects/cs240/src/clients/using_lib (2023-fall)
$ ./main.out
Hello 2 3.14169
1 add 2 3
```