CS 240 Programming in C

Get Started!

September 9, 2019

Schedule

1. C program Structure
2. Dissection of hello.c

Schedule

- Today's lecture is about getting you started with C programming by introducing you a basic C program structure.

Reminder

- Well, we will need three software for this class, which are gcc, gnu make and gdb.
- GCC stands for GNU Compiler Collection is our compiler. It can compile many other languages besides C.
- GNU Make is build automation tool. We will use slightly.
- GDB stands for GNU Project debugger.
- BTW, I am using Sublime text for class demos.

- Today we will learn to write our first C program, compile it and run it.

- First, let's assume that we know nothing about coding but just how to write plain text file.
- As we have said, C program source code is nothing but a plain texts with certain structures.

C Program Structure

C program source code is usually made up of 5 parts:

- Comments
- Reprocessing directives
- Functions
- Statements
- Variables

Comments

Comments are for the ease of users to understand the source code.

They will not be part of the machine code that to be produced.

They are discarded at compiling.

Comments are closed within /* and */. Example,

```
/* Project: HelloWorld
 * Name    : Haoyu Wang
 * Date    : 09/08/2019
 * File    : hello.c
 * Notes   : prints "hello world!"
 */
```

Comments

- More examples:

```
/* basic comment on its own line */
/*****************************************************
* multiline comments sometimes use formatting like
* this to make the commment stand out.
*****************************************************/
printf("example\n"); /* comments can follow statements */
```

- Standards after C99 recognize double slash for comments, but it is not supported in K&R or ANSI C

```
// double slash comment goes from slashes to end of line
printf("example\n"); // and can follow statements
```

- GCC that we are using supports double slash commenting by default.

Preprocessor Directive

- Recall that actual main processes of a modern compiler from our first lecture:

  source code $\rightarrow$ preprocessor $\rightarrow$ complier $\rightarrow$ assembler $\rightarrow$ object code $\rightarrow$ linker $\rightarrow$ executables

- there is a preprocessing stage before source code gets compiled.
- This preprocessing stage is a text auto modification stage based on the directives that been written in the source code.

Preprocessor Directive

- We will discuss more of directives later of this course, for now we are only caring about this one directive statement:

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

- `#include` is a preprocessor directive, and `<stdio.h>` is a header file of the standard input/out library of C.
- A directive must starts with the "#" character
- A header file of a library contains the information of that library
- This statement means the code in the library of `stdio` is to be "included" into this program for utilizing.

Preprocessor Directive

- As there are standard library and custom library, there are also their headers files respectively.
- For including them, "$<>$" and """" are used respectively:

```
\#include <stdio.h>  standard library header file
\#include "mylib.h"    custom library header file
```

Functions

- Functions are building blocks of a C programs at its running time.
- In a sense, C programs are big nested functions.

The Entry Point of a Program: `main`

- First, every C program at run time starts with the main function and ends with the return of the main function.
- Every definition of function is made of 4 parts: function name, return data type, argument data types and function body.
- Example,

```
int main(void)
{
  [function body];
  return 0;
}
```

Statements

- Function body is made of many statements which are to be executed when the program runs.
- Statements contains many kinds, like variable assignment expressions, function calls, loops as well as the mix of these kinds etc.
- Every statement in C must end with a semicolon ";". It tells compiler the execution of this statement ends here.
- On the other hand, directives are not end with semicolon. Since they are not handled by compiler.

Variables

- Variables in C are actually names to a bunch of addresses which holds certain block of memory in terms of what data types they are.
- If a C program is not complied specifically for debugging, the names of variables will be discarded, only data addresses are useful to computer.
- There are all sorts of operations that can be applied to the variable which we will get to later.
- Variable name in a source code can be composed of letters, digits, and the underscore character.
- It must begin with either a letter or an underscore.
- Upper and lowercase letters are distinct because C is case-sensitive.

Data types

- Every variable in C has a data type, which determines how much memory will be assign to it, and what kind of value that it stands for.
- It's time to write our first C program.

hello.c

```
/* Project: HelloWorld
 * Name    : Haoyu Wang
 * Date    : 09/08/2019
 * File    : hello.c
 * Notes   : prints "hello world!"
 */
#include <stdio.h>

int main(void)
{
  printf("hello, world!\n");
  return 0;
}
```

Dissection of hello.c

1. Comments

   ```
   /* Project: HelloWorld
    * Name    : Haoyu Wang
    * Date    : 09/08/2019
    * File    : hello.c
    * Notes   : prints "hello world!"
    */
   ```

2. A C preprocessor directive

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

3. A C function which includes:
   1. C function header

      ```
      int main(void)
      ```

   2. C function body, containing statements enclosed in braces

      ```
        {
          printf("hello, world\n");
          // statement which calls the function printf
          return 0; // return 0 means program runs properly
         }
      ```

GCC

- To compile our C source code, we need to invoke gcc in a terminal.
- We type gcc or just cc, which is an alias
- GCC has various command-line options that we can use for many purposes.
- `https://www.thegeekstuff.com/2012/10/gcc-compiler-options/`
- `https://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html#Overall-Options`
- `https://renenyffenegger.ch/notes/development/languages/C-C-plus-plus/GCC/options/index`

GCC

- -W, -o, -c, -g, -std, -W, -ansi etc.
- -lm will link the standard math library

Compilation

- Object code is source code that has been translated to machine code, but has not yet been linked into a complete program
- Object code files use the .o extension
- Object code vs executable code.

Running a Program

- After you have successfully produced an executable file called `hello`, you can run it by typing:
  ```
  $ ./hello
  ```
- Recall that . denotes the current working directory
- Because the executable file you just created is not included in your PATH environment variable (which tells the shell where to find executable files), you are explicitly specifying where to find `hello`
- On windows, you can run it by typing:
  ```
  $ hello
  ```

echo $?

- echo $? means status of your previous command
- `https://superuser.com/questions/556140/on-a-linux-shell-what-is-echo-supposed-to-do`

Do it!

- Now, it is your turn to try everything that we just covered.

Types of errors

- The types of errors you will encounter are generally divided into three categories:

1. Compile errors
   An error makes the compiler unable to create an executable file
2. Runtime errors
   The program compiles, but performs some illegal operation during execution
3. Logical errors
   The program compiles and runs without generating errors, but it does not provide the desired result

Compiler Error Messages

- May direct you to a specific error in your program
- May be vague about what the error is and why it is an error
- Some compilers are better than others in this regard
- Example: try to compile `hello.c` with the closing brace missing

GNU Make

- GNU Make is build automation tool.
- You must write a file called the makefile to use make.
- A makefile is just a text file describes how you want to compile your program under certain rules.
- Once a suitable makefile exists, type shell command make will perform what has been written in makefile.

A simple makefile

- A simple makefile consists of “rules” with the following shape:

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

  `there is an exact tab before any recipe`

- Once makefile is executed, if there is no target specified, make will execute the first target and up until all its dependencies. (We will get to it later.)

A simple makefile

- A target is usually the name of a file that is generated by a program
- A target can also be the name of an action(a shell command) to carry out.
- A prerequisite is a file that is used as input to create the target. A target often depends on several files or none.
- A recipe is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line.

A simple makefile

- Example,

```
hello.out: helloworld.c
    gcc -o hello.out helloworld.c

clean:
    rm  hello.out
```

A simple makefile

- Example,

```
exe = hello.out
option = -Wall -std=c99

$(exe): helloworld.c
    gcc -o $(exe) $(option) helloworld.c

clean:
    rm  $(exe)
```

A specific makefile

-
  ```
  exe = hello.out
  option = -Wall -std=c99


  clean: run
      rm  $(exe)

  run: $(exe)
      ./$(exe)

  $(exe): helloworld.c
      gcc -o $(exe) $(option) helloworld.c
  ```