csc2404 Practical

Introduction to C

Week 1

Rational for C

The reason for introducing the C language into this course is that it illustrates many of the operating system concepts discussed in this course, in particular memory addressing and pointers, compiling, linking, libraries &c. Also as the Unix/Linux OS is written in C, as is the extensive Unix/Linux System Library it can be used to illustrate how threads are managed, interprocess communication, file handling, memory management at the low level required for this course.

What this means is that a deep understanding of the language syntax is not required. Try and view it as an alternate way of illustrating some of the concepts in this course. Try and make the links between the theory in the textbook and the limitations of the hardware and the design and requirements of C.

C references

Any introductory text or reference on C will be suitable for this course. The classic tutorial and reference text for C is "The C Programming Language" by Kernighan and Ritchie. Many people find this book too terse as an introductory text but it is an excellent reference text. If you plan to program extensively in C then this text is a must.

For this course we will be using the following free references:

"An Introduction to the C Programming Language and Software Design" by *Tim Bailey*,

"The GNU C Programming Tutorial" by *Mark Burgess & Ron Hale-Evens*

and

"A Tutorial on Pointers and Arrays in C" by *Ted Jensen*

they are not necessarily the best resources but they do have the advantages of being free and they are available on the course web site.

There are many resources available on the Web, some better than others, one of the better ones can be downloaded from the URL

http://www.tutorialspoint.com/

(the C-tutorial can be downloaded as a PDF).

Compiling "Hello World"

Reading A : Chapter 1, "Introduction" of "An Introduction to the C Programming Language and Software Design" by *Tim Bailey*

Chapter 2, "Using a Compiler" of "The GNU C Programming Tutorial" by *Mark Burgess & Ron Hale-Evens*

Code: hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5   printf("Hello, World!\n");
6   return 1;
7 }
```

The `hello` program begins life as a high level C program because it can be read and understood by human beings in that form. However the CPU does not understand it—so in order to run `hello.c` on the system. the individual C statements must be translated by other programs into a sequence of low-level machine-language instructions. These instructions are then brought together and combined in a form called an *executable object program* or *executable object file* and stored as a binary file on disk. Under Linux the format of the file is Executable and Linking Format or ELF, under Windows it is Portable Executable format or PE (based on the early Unix Common Object File Format or COFF)

The translation from source file to object file is performed by a compiler driver:

```
prompt> gcc -o hello hello.c
```

The GCC compiler driver reads the source file `hello.c` and translates it into an executable object file called `hello`. The translation is performed in the sequence of four phases shown in Figure 1. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the compilation system.

- *Preprocessing phase.* The preprocessor (cpp) modifies the original C program according to directives that begin with

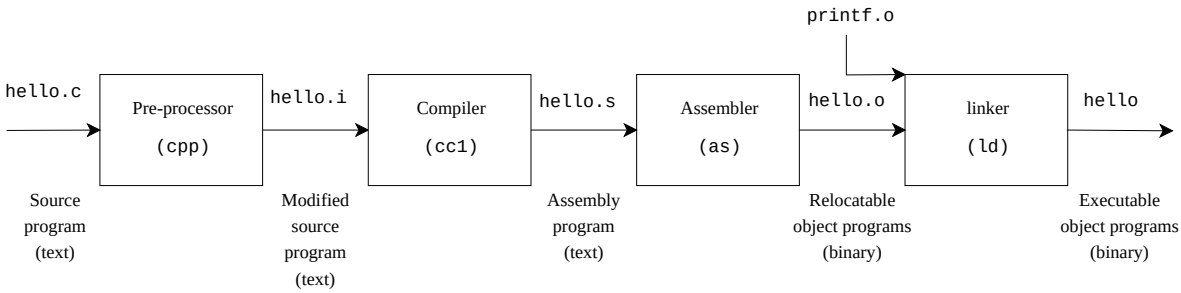

Figure 1: The Compilation System

the # character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file */usr/include/stdio.h* and insert it directly into the program text. The result is another C program, normally with the `.i` suffix.

The preprocessor is a *text to text* parser. It parses a text file and produces a text file. Though the C-preprocesor was initially designed for C programs it can be used to parse any text file. I have found it extremely useful in building and maintaining complex Cascading Style Sheet (CSS) files.

- *Compilation phase.* The compiler (cc1) translates the text file `hello.i` into the text file `hello.s` which contains an *assembly-language* program. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standardised text form (this means that the assembly language produced is architecturally dependent). Assembly language is useful because it provides a common output language for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language (assuming they are compiled on the same architecture).

- *Assembly phase.* The assembler (as) then translates `hello.s` into machine language instructions, and packages them in a form known as a *relocatable object* program, and stores the result in the object file `hello.o`. The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters. Machine instructions are single byte integers, double byte integers (or even triple byte integers) that have meaning to the CPU. If you were to view *hello.o* with a text editor, it would appear to be gibberish.

- *Linking phase.* The `hello` program calls the function `printf`, which is part of what is called the standard C library (the

code for it is clearly not in the file `hello.c` it must come from somewhere else.)—which is provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with the `hello.o` program. The linker (ld) handles this merging. The result is the `hello` file. which is an *executable object* file (or simply executable) that is ready to be loaded into memory and executed by the operating system.

Exercise 1: Compile and run the `hello.c` code above. The code has been added to this PDF file as an attachment it can be extracted by clicking on the text "hello.c" in the margin.

Reading B :

Browse Chapter 10 "The C preprocessor" of "An Introduction to the C Programming Language and Software Design" by *Tim Bailey*

Browse Chapter 12, "Preprocessor directives" of "The GNU C Programming Tutorial" by *Mark Burgess & Ron Hale-Evens*

Code: cppexample.c

Exercise 2: Consider the following code `cppexample.c`:

```
1  #define LOOP 10
2  #define CUBE(z) ((z)*(z)*(z))
3  #define QUAD(z) (CUBE(z)*(z))
4
5  int main()
6  {
7    int i;
8    int result;
9    for(i=0; i<LOOP; i++) {
10     result += CUBE(i);
11     result += QUAD(i+3);
12     result += CUBE(QUAD(LOOP-i));
13   }
14 }
```

After the C-preprocessor has parsed this code what will line 10 look like? What is the effect on line 11 of the preprocessor?

To check your answer run the following command:

```
prompt> gcc -E cppexample.c
```

Exercise 3: One of the major uses of the preprocessor is it allows "conditional compilation"—the code to be compiled can be modified depending on parameters passed to the preprocessor, Consider the following code `cppexample2.c`:

Code: cppexample2.c

```
1  #include <stdio.h>
2
3
4
5  int main()
```

```
6  {
7    int i;
8    int accumulate;
9
10 #ifdef FACT
11   accumulate = 1;
12 #else
13   accumulate = 0;
14 #endif
15
16   for(i=1; i<11; i++) {
17
18 #ifdef FACT
19     accumulate *= i;
20 #else
21     accumulate += i;
22 #endif
23
24   }
25
26 #ifdef FACT
27   printf("Factorial=%d\n", accumulate);
28 #else
29   printf("Sum=%d\n", accumulate);
30 #endif
31
32 }
```

What will be the C-preprocessor's output if the macro `FACT` is defined? If it is not defined?

To check your answer run the following command:

```
prompt> gcc -E cppexample2.c
```

and then

```
prompt> gcc -E -DFACT cppexample2.c
```

the `-Dname` option to the compiler is passed to the preprocessor and it defines the `name` macro.

Typed Variables

A CPU can can only recognise a limited set of number types—integers of 1, 2, 4 and 8 bytes, and real numbers of 4 or 8 bytes (depending on the architecture it may also recognise 16 byte real numbers). As a compiler has to convert a high-level language into low-level machine instructions—the high-level language normally also defines base variable types that match the underlying architecture.

C is a *typed* language—this means that a variable has to be assignment a type before it is used. The type tells the compiler the amount of memory required to hold the variable's value and how the variable is to be used. The compiler can then add the appropriate machine instructions that allocate the required memory for the variable and can check the code it is translating to make certain that the variable is being used correctly.

Note :

It is important to understand that since the CPU can only recognise integers and floats—variable names do not survive the compiling process. Variable names are used so people can read the code—not the CPU. The compiler replaces all variable names with instructions to fecth a value from a memory address or place a value in a memory address (a memory address is just an integer). The space in memory is allocated for the variable when the variable is declared and typed. The type of the variable tells the compiler how much space to allocate.

Reading C :

Chapter 2, "Types, Operators, and Expressions" of "An Introduction to the C Programming Language and Software Design" by *Tim Bailey*

Chapter 5, "Variables and declarations" of "The GNU C Programming Tutorial" by *Mark Burgess & Ron Hale-Evens*

Code: sizeof.c

Exercise 4: Consider the following code `sizeof.c`:

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("size in bytes of a char is %d\n", sizeof(char));
6    printf("size in bytes of a short is %d\n", sizeof(short));
7    printf("size in bytes of a int is %d\n", sizeof(int));
8    printf("size in bytes of a long is %d\n", sizeof(long));
9    printf("size in bytes of a float is %d\n", sizeof(float));
10   printf("size in bytes of a double is %d\n", sizeof(double));
11   return 0;
12 }
```

This code uses the C-language `sizeof` operator—which returns the size of an object in bytes.

Compile and run this program with the following commands:

```
prompt>  gcc -o sizeof sizeof.c
prompt> ./sizeof
```

C also defines the type `long double` for extended floating-point precision. Does your architecture support `long double`? Add and extra line to the code and output the size of `long double`. If your architecture does not support it then it should have the same size as `double`.

Code: cast.c

Exercise 5: Consider the following code `cast.c`:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    float f;
6    int i;
7    char c;
8
9
10   f = 72.764;
11   i = (int)f;
12   c = (char)f;
13
14   printf("Float=%f, Integer=%d, Character=%c\n", f, i, c);
15
16   i = 89;
17   f = (float)i;
18   c = (char)i;
19
20   printf("Float=%.12f, Integer=%d, Character=%c\n", f, i, c);
21
22   c = 'Z';
23   f = (float)c;
24   i = (int)c;
25
26   printf("Float=%f, Integer=%d, Character=%c\n", f, i, c);
27
28   i=34;
29   c=4;
30
31   f = i/c;
32
33   printf("Float (%d/%d) = %f\n", i, c, f);
34
35   f = (float)i/(float)c;
36
37   printf("Float (float)%d/(float)%d = %f\n", i, c, f);
38
39   f = i/(float)c;
40
41   printf("Float %d/(float)%d = %f\n", i, c, f);
42
43   f = (float)i/c;
44
45   printf("Float (float)%d/%d = %f\n", i, c, f);
46
47   return 0;
48 }
```

Compile and run this program with the following commands:

```
prompt>  gcc -o cast cast.c
prompt> ./cast
```

Explain the output from each printf-statement.

This code shows the effect of "explicit" casts—that is the programmer explicitly converting one type to another. There are also examples of "implicit" casts—the compiler changing the type of a variable. Which lines exhibit "implicit" casts?

Exercise 6: Why does the compiler have to perform implicit casts?

Code: storage.c

Exercise 7: Consider the following code `storage.c`:

```
1 #include <stdio.h>
2
3 int main()
4 {
5   double d = 3.14159265358979323844;
6   float f = d;
7
8   printf("Double=%.20f, Float=%.20f\n",d,f);
9
10  return 0;
11 }
```

Compile and run this program with the following commands:

```
prompt>  gcc -o storage storage.c
prompt> ./storage
```

Explain the output. Why is the output different from the original constant?

Where is the implicit cast?

Function Prototypes

Reading D : Chapter 4 "Functions" and Chapter 5 "Scope and Extent" of "An Introduction to the C Programming Language and Software Design" by *Tim Bailey*

Chapter 4, "Functions" and Chapter 6 "Scope" of "The GNU C Programming Tutorial" by *Mark Burgess & Ron Hale-Evens*

The compiler while parsing code will check that variables are used correctly (or cast them to the correct type in mathematical statements), but more importantly all functions are checked that they are being used correctly. Remember: the compiler compiles one file at a time. A reference to a function not in the current file and the compiler will know nothing about it. It will not know if you are using it correctly, Whether the correct number and type of parameters are being passed to the function. Whether the return value is being passed to a variable of the same type. Whether the compiler has to include an impicit cast. To be able to check function calls the compiler must know about the function before it is used in any

code. There are two ways to apprise the compiler about functions before they are used in code—

- Functions must be defined in the file before they are used.

  This can be done with local functions but what about library functions or if you want your functions in a separate file?

- Functions can also be "prototyped" before they are used. Prototyping means the entire function is not defined at the top of a file but only how it is to be called. This is the main purpose of header files—to prototype standard C library functions and system functions—to show the compiler how functions are to be used.

  For example, to use a `tan()` function that takes a double and returns a double the statement at the head of the file that planned to use the function would be:

  ```
  double tan(double x);
  ```

  There is no function definition — just how it will be called and what it will return. This is enough information for the compiler to check the code to see that the `tan()` function is being used correctly.

  Note: The variable name `x` is optional it could easily have been `double tan(double);`

Exercise 8: What function must appear in every C-program? What is special about it?

Exercise 9: The `hello.c` program has the C-preprocessor line—
`#include <stdio.h>`
the standard header files (both system and C) can be found (under Unix) in the directory `/usr/include/`.

Headers contain preprocessor constants and function prototypes—find the function prototype required by the compiler to be able to compile the `hello.c` program without errors.

Code: distance.c

Exercise 10: Consider the following code `distance.c`:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double distance(double x, double y, double z)
5  {
6    return sqrt(x*x+y*y+z*z);
7  }
8
9
10 int main()
11 {
12   double d = distance(4.0, -3.0, 5.34);
```

```
13
14   printf("Distance=%g\n",d);
15
16   return 0;
17 }
```

Compile the code with the following command:

```
prompt> gcc -o distance distance.c -lm
```

The option `-lm` tells the loader to look for functions in the math library. What happens if the `-lm` option is not included?

Rewrite the program so that the function `distance()` is declared after the `main()` function where it is used. What happens when you try and compile it?

Add a prototype line for the `distance()` function before the the `main()` function—does it compile now?

Code: parameters.c

Exercise 11: Consider the following code `parameters.c`:

```
1 #include <stdio.h>
2
3 int scope(int a, double d)
4 {
5   float f;
6
7   f = d + 3.141596;
8   a++;
9
10  printf("Scope f=%f, a=%d\n", f, a);
11
12  return (int)f;
13 }
14
15 int main()
16 {
17  float f = 2.7123409;
18  int a = 33;
19
20  scope(777, f);
21
22  printf("Main f=%f, a=%d\n", f, a);
23
24  return 0;
25 }
26
```

The parameters of the function `scope()` "a" and "f" are called dummy parameters—what does this mean?

Explain why the variables "a" and "f" are different in the two functions `scope()` and `main()`

Standard C Library

The C language definition provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation, mathematical functions &c. Instead, these facilities are defined in the C standard library.

Any C-compiler on any system must supply the standard C-library. The functions of the library on different operating systems will be implemented using different system routines though.

See the C Library Reference Guide available on the course web site.

Note :

All of the C-library functions specified in the reference have their own Unix man pages that fully explain their usage and the header-file that must be incorporated before they can be used.

Information about the C-library functions can also be found in "Appendix A: C Language Facilities in the Library" in the The GNU C Library Reference Manual. This manual describes every function, macro and constant available to the programmer contained in the GNU C library—the backbone of all free Unix systems.

Code: input.c

Exercise 12: Consider the following code `input.c`:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char input[8];
6    int k = 3;
7
8    printf("Please enter a string: ");
9    gets(input);
10   printf("The string you entered is: %s\n", input);
11   printf("The integer k=%d\n", k);
12
13   return 0;
14 }
```

Compile and run this program with the following commands:

```
prompt>  gcc -o input input.c
prompt> ./input
```

What happens when the string you enter is greater than 12 characters? What is causing the odd behaviour?

The coding error is called a "buffer overflow". How can it be prevented?

Rewrite the code to use the safer input function `fgets` (the input stream used should be "stdin"). Why is this function "safer" than `gets`?

Exercise 13: Browse the GNU C Library Reference Manual. This manual describes the basic services available to any free Unix system—services that programs can access. This manual is available on the course web site.

Note : Though these procedures are written in C—many languages (including scripting languages) can access them.

The procedures provided by the GNU C Library are not "system routines" but they will need to call the system routines provided by the kernel to perform their tasks.

Compare the following sections:

- Input/Output on Streams (high level Standard C library functions),
- Low-Level Input/Output, and
- Input and Output Primitives

As the functions get lower-level they come closer to the underlying "system routine" that must be called to actually perform the I/O operation.

Exercises

Ex. 14: Browse Chapter 3 "Branching and Iteration" of "An Introduction to the C Programming Language and Software Design" by *Tim Bailey* and familiarise yourself with the C syntax.