COMP2017 / COMP9017 Tutorial 2

Addressable Memory and Standard Library Functions

The Preprocessor and Function Prototypes

The preprocessor is part of the compilation pipeline that allows functions from separate translation units to be utilised in your source files and linked together. We are also able to define compile time constants and compile code based on conditions known at compile time. This functionality is exposed through the C preprocessor language and we will be focusing on using `#define` and `#include` to statements in our following C programs. Prior to compilation it will evaluate the preprocessor statements and replace the value the preprocessor symbol is defined as.

`#define`

Although there exists a const modifier within the C language, you are able to define constants and macros to be evaluated at compilation using preprocessor directives. The `#define` directive allows us to express constants and macros to be used as part of our source code.

```c
#define PI 3.14
```

We can therefore use PI through out our program, during the preprocessor phase, wherever `'PI'` is encountered by the preprocessor, it will replace it with the value `3.14`.

`#include`

The `#include` statement allows you to bring in header files which will expose function prototypes your program for you to use. This preprocessor directive allows you to include other files (commonly header files) into your source code (similar to a copy and pasting).

A very common use with `#include` is to include `stdio` and `stdlib` header files. These header files provide access to the function prototypes and preprocessor constants such as `printf`, `fopen`, `scanf` and `NULL`.

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

For local header files, we use the the double quotes around the file.

```c
#include "local.h"
```

Compilation with flags

You should make sure to compile your programs with the C11 standard and enable all compiler warnings with `-Wall`. You will be taught more useful compiler options in the future, in the mean time you can look up the `clang` man page to see other possible options.

```
$ clang -std=c11 -Wall <file.c> -o <file>
```

`printf`

`printf` is used to print values to the standard output in C. This is a very useful function and you need to learn its parameters. The function prototype is:

```c
int printf(const char* format, ...);
```

The first argument is the format string, and then takes zero or more arguments which are used to fill in the placeholders within the format string.

In the simplest form, the format string simply echoes its value to standard output. The string is printed as given by the programmer. There are escape code/for characters that are bothersome to type or are used as part the C syntax, these are listed in the table below.

ESCAPE	REPRESENTS	ESCAPE	REPRESENTS
\a	Bell (alert)	\'	Single quote
\b	Backspace	\"	Double quote
\n	Newline	\\	Backslash
\r	Carriage return	\ooo	Character in octal notation
\t	Tab space	\xhh	Character in hex notation

Note: different operating systems use different characters to designate a new line. On Linux, the line feed character is used as the new line character. On Windows, the combination of a carriage return and line feed designate a new line. In C, if you specify \n, then the compiler will substitute it with the appropriate characters for your operating system.

```c
#include <stdio.h>

int main(void) {
    printf("int=%d float=%f string=%s\n", 420, 3.141f, "Hello from C");
    return 0;
}
```

The format string uses placeholders for printing C data types.
The placeholder always begins with % and has the following format:

```
%[flags][width][.precision][length]specifier
```

Here is a list of the most commonly used placeholders:

FORMAT	OUTPUT	EXAMPLE
%c	character	a
%d or %i	signed integer	392
%f	decimal floating point	392.65
%s	NULL-terminated sequence of characters	sample
%u	unsigned integer	7235
%x	unsigned hexadecimal integer	7fa
%X	unsigned hexadecimal integer	7FA
%p	pointer address	0x7fff6efe9ca8
%%	literal percentage sign	%%

The example program shown below outputs some records using more complicated placeholders:

```c
#include <stdio.h>

int main(void) {
  printf("%-3s %-15s %-5s\n", "ID", "Name", "Score");
  const char* format = "%03d %-15s %.2f\n";

  printf(format, 1, "Bryan Cantrill", 99.449);
  printf(format, 2, "Scott Forstall", 99.999);

  return 0;
}
```

The %-15s tells printf to print all strings into a 15 character wide field and fill any leftover room with spaces. The - gives it left alignment. %010d tells printf to pad the signed integer into a 10 character wide field and fill any leftover room with zeros. %.2f says to round the floating point value into 2 decimal places.

The output of the program is:

```
ID  Name            Score
001 Bryan Cantrill  99.45
002 Scott Forstall  100.00
```

You can read through the documentation for printf using: `$ man 3 printf`

scanf

To read and interpret string data from standard input, `stdio.h` provides the `scanf` function.

```c
#include <stdio.h>
int main(void) {
  int number;
  if (scanf("%d", &number) != 1) {
    puts("No number given");
    return 1;
  }
  printf("Your number is: %d\n", number);
  return 0;
}
```

`scanf` uses the same placeholder format as the `printf` function. `scanf` requires you to pass a pointer to the variable as its parameters. This allows `scanf` to update the variable.

The `&` operator converts a variable to a pointer containing its address. Here's an example of using `scanf` to read formatted input:

```c
#include <stdio.h>
int main(void) {
  int x;
  double y;
  char buffer[20];
  // Check all three inputs are given
  if (scanf("%d %lf %19s", &x, &y, buffer) != 3) {
    fprintf(stderr, "Invalid input\n");
    return 1;
  }
  printf("%d %f %s\n", x, y, buffer);
  return 0;
}
```

The `scanf` function returns the number of input items successfully matched and assigned. This can be fewer than the number of arguments passed into it. It will return zero when it fails to read any input for any reason.

Note: `buffer` is an array and has type `char[20]`, it automatically decays into a `char *` that contains the address of the first element of the array. Its usage here is the same as writing `&buffer[0]`.

When reading strings with `scanf`, you should provide a width specifier. This is because the `buffer` array we have declared is a fixed size. If a string longer than 20 characters was input (including the null terminator), `scanf` will write outside the bounds of array. This can potentially crash the program, overwrite the values of other variables or introduce a buffer overflow security vulnerability.

Pre-tutorial Work

Question 1: Shout

Write a C program that uppercases all letters inputted from the standard input. Characters that are not in the range [a-z] should pass through unchanged. You can use the `toupper` and `getchar` functions.

Sample input:

```
abc 123
lorem ipsum
dolor sit amet
```

Sample output:

```
ABC 123
LOREM IPSUM
DOLOR SIT AMET
```

You can use `toupper` function as part of the `ctype.h` header, you can check out the toupper function using `man 3 toupper`. Alternatively you can use the ascii table as a way of transforming the characters. As an extension, write a program that will transform all input to lower case.

Question 2: Travel and conversion

You are planning a road trip with your American friends. Given a kilometres per hour rate and the travel duration, you are curious how far you and your friends can travel.

Remembering that you need to communicate this distance to your American friends, you will also need to convert KM/h to M/h (miles per hour) as well as the distance travelled into miles. Round this down to

```
$ ./travel
What is your current km/h: 50
How many hours are you travelling for: 2.5
You will cover: 125.00 km (77.57 mi)
While travelling at 50 km/h (31.07 mph)
```

C strings and math library

C does not have a string type within its standard library. To represent a string in C, you will need to use a character array to store a sequence of characters, ending with a null character terminator. However, part of the C standard library is the `string.h` header that allows you to manipulate strings.

The standard library provides math functions that can be found within the `math.h` header file. You can find the functions by using the command `man math.h`. Usages:

```
#include <math.h>
```

Typically when using math functions you will need to include `link` to the math library.

```
gcc my_math_program.c -lm -o program
```

You will have your typical mathematical functions. However you will notice that some create a small distinction of the type they return by prepending an l or f to their name, denoting long or float respectively.

```
...
exp2(x)
fdim(x,y)
fma(x,y,z)
fmax(x,y)
fmin(x,y)
hypot(x,y)
lgamma
llrint
lrint
llround
lround
log1p(x)
...
```

Question 3: C types and pointers

Answer the following questions without running any code.
You may check with the compiler after you have attempted these.

- Suppose the variable `int x` has address `0x1000`, and `sizeof(int)` is 4.
  What does `printf("%p %p\n", &x, &x + 1)` output?
- Suppose the variable `char y` has address `0x1000`.
  What does `printf("%p %p\n", &y, &y + 1)` output?
- Suppose the first element of `char z[100]` has address `0x1000`.
  What does `printf("%p %p %td\n", &z[1], z + 5, &z[12] - &z[9])` output?
  What does `printf("%p %p\n", &z[3] - 1, &z[20] - 5)` output?
- Suppose the first element of `double u[100]` has address `0x1000`, and `sizeof(double)` is 8
  What does `printf("%p %p %td\n", &u[1], u + 5, &u[12] - &u[9])` output?
  What does `printf("%p %p\n", &u[3] - 1, &u[20] - 5)` output?
- What does `printf("%c %d\n", 'a' + 1, 'z' - 'a')` output?

Question 4: Cricket

Write a C program that will store a batsman's name and the number of runs that they have scored and report the total runs and each score. If a batsman has not scored any runs, a `Duck` should be printed next to their name.

```
$ ./batsman
Enter Name and Score for batter 1: Cameron Bancroft 40
Enter Name and Score for batter 2: Mitchell Marsh 67
Enter Name and Score for batter 3: Joe Burns 123
Enter Name and Score for batter 4: David Warner 59
Enter Name and Score for batter 5: Travis Head 21
Enter Name and Score for batter 6: Will Pucovski 0
Enter Name and Score for batter 7: Ben McDermott 0
Enter Name and Score for batter 8: Shaun Marsh 10
Enter Name and Score for batter 9: Cameron White 78
Enter Name and Score for batter 10: Usman Khawja 54

1. C. Bancroft: 40
2. M. Marsh: 67
3. J. Burns: 123
4. D. Warner: 59
5. T. Head: 21
6. W. Pucovski: Duck
7. B. McDermott: Duck
8. S. Marsh: 10
9. C. White: 78
10. U. Khawja: 54
```

You may assume the format is: `<first name> <last name> <score>` and there at maximum 10 batsmen. The length of each first name and last name can be up to 100 characters.

As an **extension** you can attempt to solve this problem without assuming the maximum length of someone's first and last name.

Question 5: Discuss Work

During the tutorial, discuss with your tutor your answers to question 3 and your implementation to question 4 (cricket scoring program).

Tutorial Work

Question 6: Volume of a sphere

Write a C program that will calculate the volume of a sphere. The program must allow the user to input a radius.

In case you have forgotten how to calculate the the volume of a sphere, you can use the following formula:

$V = \frac{4}{3}\pi r^3$

```
$ ./vsphere
Specify the radius of the sphere: 2
Volume is: 33.509335
```

You may use following definition of PI

```
const float PI = 3.1415;
```

After you have successfully written and tested your program, replace PI with `M_PI` from `math.h`. Check with your tutor about your compilation step if you run into trouble or refer to the *C strings and math library* of this tutorial.

Question 7: `strlen`

Implement your own `strlen` function. Note that the function definition for `strlen` uses `const char*` , this tells the compiler that this pointer can only be used to read. It is good practice to use only `const` pointers if you don't need to write to the memory referenced by the pointer.

```
#include <stdio.h>

int my_strlen(const char* s) {
  //TODO
}

int main(void) {
  printf("%d\n", my_strlen(""));                  // should output 0
  printf("%d\n", my_strlen("123"));              // should output 3
  printf("%d\n", my_strlen("abc\n"));            // should output 4
  printf("%d\n", my_strlen("lorem\0ipsum\n")); // should output 5
  printf("%d\n", my_strlen("lorem ipsum\n"));  // should output 12
  return 0;
}
```

The `strlen` function is declared in the `string.h` header. You can double check your implementation by comparing it to the standard library function.

Question 8: Finding a substring

Construct a function that will find the locate the starting index of a string within another. This function should return a negative value if the substring does not exist within the line.

```c
int substring(const char* line, const char* substr) {
    //TODO
}

int main() {
        printf("%d\n", substring("racecar", "car")); //4
        printf("%d\n", substring("telephone", "one")); //6
        printf("%d\n", substring("monkey", "cat")); //-1

        return 0;
}
```

Question 9: Run-length Encoding

A common compression method is to encode the run length of the same symbol. The method also shows up in puzzle games such as picross.

```c
int encode_run(const char* sequence, unsigned len,
    char* buf, unsigned int buf_len) {

    //TODO
}

int main() {
    char encoded_run[128];
    const char* line_run = "1122333334423";

    encode_run(line_run, 14, encoded_run, 128);
    printf("%s\n", encoded_run); // 225211
}
```

Question 10: Run-length Tuples

Similar to the previous question, we want to encode the runs of a sequence. However, the twist with this function is that we want to encode what symbol the run relates to.

```
int encode_run(const char* sequence, unsigned len,
    char* buf, unsigned int buf_len) {

    //TODO, Modify your existing run
}

int main() {
    char encoded_run[128];
    const char* line_run = "1122333334423";

    encode_run(line_run, 14, encoded_run, 128);
    // (1,2) (2,2) (3,5) (4,2) (2,1) (3,1)
}
```

Write a decoder function that will output the run from the tuples buffer.

```
int decode_run(const char* tuples, unsigned len) {

    //TODO, Modify your existing run
}

int main() {
    char encoded_run[128];
    const char* line_run = "1122333334423";

    encode_run(line_run, 14, encoded_run, 128);
    //(1,2) (2,2) (3,5) (4,2) (2,1) (3,1)

    decode_run(encoded_run, 128); //1122333334423

    return 0;
}
```

Question 11: Password generator

Using the random function `rand` and `srand` functions from `stdlib.h`, set the seed for the random function and generate a password for a user. Your program should utilise the `ascii` table and will need to include valid keyboard characters, symbols and integers.

The length of the password will be specified as a command line argument.

```
./pass_gen 8
kl^f832n
```

You will need to use `rand` and `srand`, these functions are declared in the `stdlib.h` file. You can read the man pages of the `stdlib` header file by using the command `man stdlib.h`.

Question 12: `out`

Your program should behave in a similar way to the Unix `cat` program. If no arguments are passed to your program, it should read from `stdin` and output the input. If arguments are passed to your program, it should treat each argument as a file and output the contents of each file in order.

You should use the following functions: `fgets`, `fread`, `fopen` and `fprintf`.

```
$ echo hello | ./out
hello

$ echo hello | ./out - file1
hello
<contents of file1>

$ ./cat file1 file2 file3
<contents of file1>
<contents of file2>
<contents of file3>
```