THE UNIVERSITY OF SYDNEY

COMP2129 Week 3 Tutorial

IO, Pointers, and Scripts

Exercise 1: `printf(3)`

For printing values in C, the function **`printf`** is used. Note that this is very important in C and you need to learn its parameters. The function prototype is given below,

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

where 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 except the escape characters in C, such as `\n`, `\r` and `\ooo` (where `o` are octal digits). These escape characters are for special codes such as line-feed, carriage-return etc. that cannot be typed, but are necessary for formatting your output.

Escape	Represents	Escape	Represents
`\a`	Bell (alert)	`\'`	Single quote
`\b`	Backspace	`\"`	Double quote
`\n`	Newline	`\\`	Backslash
`\r`	Carrige return	`\ooo`	Character in octal notation
`\t`	Horizontal tab	`\xhh`	Character in hex notation

In addition to escape characters, the format string may contain placeholders for printing integers, strings, and floating point numbers by specifying parameters:

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

int
main(void) {
  printf("integer=%d float=%f string=%s\n", 4711, 3.141, "Hello, World!");
  return 0;
}
```

A placeholder for an integer, string, etc. has the following format:

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

where the specifier is the more significant part as it defines the type and the interpretation of the value of the corresponding argument.

This gives quite a powerful little-language for describing output formats. For example, the following program prints some fictional student records so that everything is nice and aligned:

```c
#include <stdio.h>

char *names[] = {"Bob", "Jane Doe", "John Smith"};
char *sids[] = {"12345678", "23456789", "98765432"};
double marks[] = {99.8763, 78.29561, 62.11345};

int main(void) {
  int i;
  printf("Name       SID      Mark\n");
  for (i = 0; i < 3; i++) {
    printf("%-10s %s %.2f\n", names[i], sids[i], marks[i]);
  }
  return 0;
}
```

The `"%10s"` tells printf to print all strings into a 10-character wide field, with any room filled in by spaces. The `"%-10s"` gives it left-alignment. The floating point specifier `"%.2f"` says to only print 2 decimal places, and performs appropriate rounding.

The output of the program is:

```
Name       SID      Mark
Bob        12345678 99.88
Jane Doe   23456789 78.30
John Smith 98765432 62.11
```

You should experiment around with different output formats, as well as check out the man page (**man 3 printf**) for more awesome little features.

Format	Output	Example
`%c`	character	`a`
`%d` or `i`	signed decimal integer	`392`
`%e`	scientific notation	`3.9265e+2`
`%E`	scientific notation	`3.9265E+2`
`%f`	decimal floating point	`392.65`
`%g`	use the shorter of `%e` or `%f`	`392.65`
`%G`	use the shorter of `%E` or `%f`	`392.65`
`%o`	signed octal	`610`
`%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	`%`

Exercise 2: `scanf(3)`

To read from standard input, C provides the **`scanf`** function, and it can read various types of input such as characters, strings, and integer numbers.

```c
#include <stdio.h>

int
main(void) {
  int x;
  if (scanf("%d", &x) != 1) {
    fprintf(stderr, "no number given\n");
    return 1;
  }
  printf("%d\n", x);
  return 0;
}
```

Function **`scanf`** works with format strings as well. However, instead of passing on the value of a variable as done in **`printf`**, you need to give the address of the variable so that **`scanf`** can manipulate the value of the variable **`x`**. This is similar to a reference in Java which you pass on as an argument so that the caller can manipulate the object. To obtain an address of an object, you need to use the operator **`&x`** that delivers the pointer/address of **`x`**.

Consider following example that reads in a variety of different variable types (integer, floating point number, and string):

```c
#include <stdio.h>

int
main(void) {
  int x;
  double y;
  char buf[20];
  if (scanf("%d %lf %s", &x, &y, buf) != 3) {
    fprintf(stderr, "incorrect input\n");
    return 1;
  }
  printf("%d %f %s\n", x, y, buf);
  return 0;
}
```

The `"%lf"` here means “long float”, i.e. a `double`. **`printf`** doesn’t distinguish between `float` and `double`, but **`scanf`** does.

Note that **`scanf`** is not a safe input method for strings because no string length is specified, i.e., if the input is large enough it can overwrite the whole memory and your program crashes or even worse – it can expose a security risk. In the example above, what happens if the user inputs more than 19 characters?

Exercise 3: Pointers 101

A pointer is a data type which contains an address for some other data.

```c
#include <stdio.h>

int main(void) {
  int a = 2, b = 3;
  swap(&a, &b);
  printf("%d %d\n", a, b);
  return 0;
}
```

- Write a **swap** function that will swap two integers. Your function should modify two existing integers a and b such that when the function returns, b holds the value previously in a and a holds the value previously in b. Include the swap function in the above code snippet, and verify that it works.
- Discuss with the person next to you why it is important that the function accepts pointers to integers rather than integers. Does this help you understand why we commonly use the address-of operator `&` when we call `scanf`? You should write a program to demonstrate your point and confirm your logic.
- What happens if instead of being of type `int`, `a` and `b` are instead some sort of `struct`? Try making a struct which holds ints (or anything you like, really) and passing them to an appropriate `swap` function, first by calling `swap(a, b)`, then modifying the function to take pointers and call `swap(&a, &b)`. What can you deduce about how structs are passed into functions?

Since we are learning about pointers, now is also a good time to learn about *segmentation faults*[1]. A segmentation fault occurs when a program tries to access a memory address that it shouldn't. When this happens the operating system will send a special signal to your program which causes it to terminate. Often a segmentation fault is called by trying to dereference a pointer which is either `NULL` or for some reason contains a value that is not a valid memory address, or at least isn't the memory address that you thought it was.

- Download the file `break_me.c` from Blackboard or Piazza, and copy it to your current working directory. Can you find a set of inputs that will cause the program to reliably fail? Can you fix it?

[1] http://en.wikipedia.org/wiki/Segmentation_fault

Exercise 4: A statistics program

The given program will read numbers from standard in into an array indefinitely (it doesn't know when to stop). Make it stop when it recieves the number -1, and output the mean and sample standard deviation of the numbers read so far.

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

#define MAXLEN (1024)

int main(void) {
  double nums[MAXLEN];
  int pos = 0;
  for (;;) {
    scanf("%lf", &nums[pos++]);
  }

  double mean = 0, stdev = 0;
  printf("mean: %.2f\n", mean);
  printf("stdev: %.2f\n", mean);
  return 0;
}
```

An example session with your program should look like:

```
1 1 1 3 5 -1
mean: 2.20
stdev: 1.79
```

Recall that the mean $\bar{x}$ of a list of numbers $x_1, x_2, \ldots, x_n$ is

$$\bar{x} = \frac{1}{n}(x_1 + x_2 + \ldots + x_n)$$

and that the sample standard deviation is computed using the mean, and is

$$s = \sqrt{\frac{1}{n-1}\left((x_1 - \bar{x})^2 + \ldots + (x_n - \bar{x})^2\right)}$$

Here functions from `<math.h>` are included, so the math library needs to be linked when compiling, using the `-lm` flag:

```
$ gcc -o stats stats.c -W -Wall -std=gnu99 -lm
```

A problem with this program is that it can't handle more than `MAXLEN` numbers, and if `MAXLEN` is set too large, it will exceed the stack memory. Dynamic memory allocation is the way to go here (you'll be learning it in a week or two).

Exercise 5: Aggregate Data Types

The C programming language allows you to construct your own aggregate data structures, such as *structs* and *arrays*. A struct is for storing related data items of different types: for example you could have the following struct for storing the name, species and level of your Pokémon:

```c
struct Pokemon {
    char nickname[11];
    char type[11];
    int level;
};
```

You could create an instance of this type as follows:

```c
struct Pokemon first = {"Bill", "Bulbasaur", 3};
```

To save ourselves some typing, will add a `typedef` mapping the type `struct Pokemon` to the shortcut `Pokemon`:

```c
typedef struct Pokemon Pokemon;
```

- Start a new file, including the struct definition above and the typedef.
- Write a function called `printPokemon` which should take a pointer to a `Pokemon` and print out some relevant information to `stdout`. Test your function with a few different Pokémon. (Quiz: where in memory are these structs located?)

An array is for storing a collection of data items where the items all have the same time. For example we could store an array of integers, floats, chars or pointers. Because the C programming language does not include a "string" data type, a string is commonly represented as an array of characters where the end of the string is marked with the special char `'\0'`. This is called a *null-terminated string*, and there are many useful functions in the C standard library to help you cope with these.

Arrays can also be used to store items of custom types, such as our Pokémon struct.

- Declare a fixed size array of `Pokemon` and write a loop to read in the nickname, type (species) and level of a few Pokémon from `stdin`.
- Modify your program so that after the user has entered his or her list of Pokémon your program will print out the details of the Pokémon with the highest level.
- Create a text file containing the list of the 151 original Pokémon species (download from the Internet and use `bash` to clean), one on each line.
- Read the original species into memory (you may assume no name is longer than 10 characters), and use this to check that the type inputted by the user is a valid Pokémon.

Exercise 6: Shell scripts

One of the great things about learning syntax for **bash** is that all of your commands can be put into a script, and reused again and again. Try creating a new file called **hello.sh** containing the following:

```
#!/bin/sh

echo 'Hello, world!'
```

You can run your script after you make it executable using the **chmod** command:

```
$ chmod +x hello.sh
$ ./hello.sh
```

The first line, starting **#!** is called a "hash-bang" or a "shebang", and tells the operating system to invoke the program on that same line to run the script. **/bin/sh** is usually a symbolic link to **/bin/bash**, but in principle can be any POSIX-compatible shell (on Android devices, it links to **/bin/mksh**, a different shell). If you are programming with any bash-specific features, it's safer to use **#!/bin/bash** as your shebang line.

Shell scripts are handy for automating common tasks. For example, you can count the number of lines of C code in the current directory:

```
#!/bin/sh

total="0"

for cfile in *.c
do
  nlines=$(wc -l < $cfile)
  total=$(($total + $nlines))
done

echo "$total lines of code found."
```

And run it like:

```
$ ./countclines.sh
53 lines of code found.
```

Loops are able to be used in command lines too: for example try entering the following line:

```
$ while true; do echo "Hello"; sleep 1; done
```

Of course, you can use `Ctrl+c` to abort the loop.

Exercise 7: Shell scripts for testing

Testing code can be done manually on the command line. For example, to test the stats program from a previous exercise, first create a directory **`tests`**, and a pair of files **`tests/test1.in`** and **`tests/test1.out`**. Into **`tests/test1.in`**, place

```
1 1 1 3 5 -1
```

and into the **`tests/test1.out`**, place

```
mean: 2.20
stdev: 1.79
```

Now with the compiled program, diff the output of the program against the expected output.

```
$ ./stats < tests/test1.in | diff - tests/test1.out
```

(The **-** as an argument to **`diff`** tells it to use standard input rather than a file in that argument).

This method of testing is very effective, but also very cumbersome. A lot of test cases means a lot of typing. Your task is to write a shell script to automate this: run every test in the **`tests/`** folder against the program, and display alerts to the user if any of the testcases differ in their output.

You might like to look up the **`basename`** command, which can help getting rid of the file extension on files, and also how **`if`** statements work. Another handy thing from bash is the **`&&`** and `||` operators. **`&&`** will execute the following command only if the previous command had a success (zero) return code. `||` will execute the following command only if the previous command had a fail (nonzero) return code. Used together, the two can work like very compact single-program **`if`** statements:

```
$ true && echo 'Woohoo!' || echo 'Awwww'
Woohoo!
$ false && echo 'Woohoo!' || echo 'Awwww'
Awwww
```

Diff will return zero if the files were exactly the same, and nonzero otherwise. It also has some very handy flags you might like to use: be sure to check the manual.