Hello C...TYI!

Domhnall O'Hanlon

February 7, 2017

Part 1: Input > Process > Output

In this secion you will write your first C programs, and learn about the differnt bits and pieces that are required to compile and run software written in C. By the end of this section you should be able to

- Write a simple, interactive, exectuable program.
- Understand essential C syntax
- Be aware of different data types in C

Hello World!

"Hello World" is typically the first program that students learn to write, regardless of what language they're learning. It's origins are as old as C itself so let's follow in the footsteps of all the great programmers and write it for ourselves.

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

int main(){
  printf("Hello World");

  return 0;
}
```

Analysis

The first line of our Hello World programs has `#include <stdio.h>` which **includes** commands for inputting and outputting data. We will cover these functions in greater detail in the libraries section. The next line `int main()` creates a function called main that returns a whole number (integer) when it finishes. Again, we will also cover functions in much greater detail but for now

it is sufficient to know that every program that you write must contain a `main()` function. The contents of any function are placed between braces (sometimes called curly braces) like these `{ }`. In the Hello World example, out main function only does two things; firstly it prints the message *Hello World* and then it exits successfully by returning 0. Each of these steps that the program takes is called a **statement**. Note that every statement you write in C (and many other langauges) will always end with a semicolon.

Comments

Obviously, as the weeks go on we'll be creating more and more sophisticated programs, and we'll also be collaborating with other programmers. In both of these instances it is useful to have have some human-readable text that gives some clue as to what the program or function we are looking at is supposed to do. C has comments for this very purpose. There are two types of comments:

Single line comments that are indicated with two forward slashes `//`

```
//this is a single line comment.
```

Multi line comments, which can span one or more lines. A multi line comment begins with `/*` and ends with `*/`.

```
/*
This is a
multiline
comment
*/
```

Originally C only supported multi line comments and support for single line comemnts was introduced after the release of the C++ language. For full backwards compatibility with older compilers it is recommended that you always use multiline comments.

Escape Characters

What if you wanted print the words *Hello* and *World* on seperate lines? How would you insert a line break into your text? In the example above, the text you want to display is contained between double quotes, but what if you wanted to display some dialog, like, *Domhnall said "hello world"*, how do you display double quotes without inadvertently closing one string and opening another by accident? Try it if you like.

The solution is to use **escape characters**. This means the symbol you want to display, or keyboard character you want to enter, is escaped by placing a backslash, \, in front of it. Try the following example

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

int main(){
  printf("Alice said \"Hello World\" \n");
  printf("Bob said \"Hello World\" too. \n");

  return 0;
}
```

The `printf()` function will print out every character it sees in between the open and closing double quotes. In the example above, to display quotes in the console you have to escape them by preceding the quotes with a backslash. You also have to explicitly tell the complier when you want to go on to a new line, and this is acheived using the new line escape character, `\n`. A complete list of escape characters is included in the table below.

sequence	output
`\a`	Alarm (Beep, Bell)
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline (Line Feed);
`\r`	Carriage Return
`\t`	Horizontal Tab
`\v`	Vertical Tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\nnn`	A character where nnn interpreted as an octal number
`\xhh`	The character where hh interpreted as a hexadecimal number

Conversion Characters

Many functions, such as `printf()`, will allow us to substitute information into a `printf()` statement, rather than having to hard code it directly into the output.

```
#include <stdio.h>

int main(){

  printf("Your lucky number is is %d", 42);

  return 0;
}
```

The `printf()` function is now getting two pieces of information, seperated by commas. The first piece of information is a string or message to display and the second piece of information is the name of the variable we want to display. At the end of the string there is a new sequence of characters `%d`. This tells the compilers to go and find a varaible and substitute in its value in place of the `%d`. There are different **conversion characters** depending on which type of data you want to substitute into your function. In this case, since we are substituting in a whole number we use the integer data type. We'll learn more about data types later in the chapter.

The following table contains a complete list of conversion characters.

Character	Argument to Display
`%c`	Single character (char)
`%d`	Signed decimal integer (int)
`%e`	Signed floating-point value in E notation
`%f`	Signed floating-point value (float)
`%g`	Signed value in %e or %f format, whichever is shorter
`%i`	Signed decimal integer (int)
`%o`	Unsigned octal (base 8) integer (int)
`%s`	String of text
`%u`	Unsigned decimal integer (int)
`%x`	Unsigned hexadecimal (base 16) integer (int)
`%%`	(percent character)

Examples

In this first example we see that the integer conversion character can substitute in the result from a mathematical calculation and the substitutions are performed in order. Try this for yourself by changing the digits and the type of calculation.

```
#include <stdio.h>

int main(){

  printf("When you add %d and %d you get %d", 2, 3, 2+3);

  return 0;
}
```

In the next example we see how you can output a specific number of digits from the a **floating point** number. By default a float will display six decimal places but you and add values to the conversion character to change this. `.5` means that five decimal places should be printed, `.4` means display four decimal places, `.3` measn 3 and so on. Take a look at the following example, which outputs pi to 6, 4 and 2 decimal places - but with questionable precision!

```
int main(){
  printf("I ate some %f", 3.141592);
  printf("I ate some %.4f", 3.141592);
  printf("I ate some %.2f", 3.141592);

  return 0;
}
```

In the example above there is a (deliberate) rounding error. The first 8 digits of Pi are 3.1415926 so we should have rounded our float up to 3.141593. In order to rectify this mistake we have to make three changes - but in larger programs a small change in specification might require you to make hundreds or thousands of changes. As you can probably imagine, hard-coding values like this directly into `printf()` statements is a bad idea. It makes code harder to maintain and much more error prone. If you continue programming like this you'll end up with slow, inefficient software that is full of tiny mistakes.

As a courtesy to your future self, now is the time to learn as much as you can about variables.

Variables

A variable is simply a placeholder where we are going to store some information in the computer's memory. There are three things you need to do to create a variable in your program. First you need to tell the compiler what **type** of data you'll be working with - i.e. is it a number or letters etc. Then you need to **name** your variable, so that it can refer to it elsewhere in your code. Finally you need to give your variable a **value**. The following code snippet has improved the previous example by creating a **variable** called pi and assigning it a **value** of 3.141593.

```
int main(){

  //declare a floating point variable and name it "pi"
  float pi;

  //assign a value of 3.141593 to the pi variable
  pi = 3.141593;

  printf("I ate some %f", pi);
  printf("I ate some %.4f", pi);
  printf("I ate some %.2f", pi);

  return 0;
}
```

To make your life easier you can **declare** and **initialise** a variable in one line like so:

```
float pi = 3.141593;
```

You can even declare several variable, of the same data type, all at once on the same line. Just use commas to sepearate each varaible name.

```
//some irrational numbers
float e, pi, tau;

//a few floating point inputs
float price, height, temperature;
```

Variable Types

When you started learning maths in secondary school you were probably introduced sets for the first time. You will no doubt remember that there are different *types* of numbers. For example the set $\mathbb{Z}$ is that set which contains all the positive and negative whole number, or integers. Similarly, the set $\mathbb{R}$ contains all the rational and irrational numbers.

In many computer programming languages, including C, you have to specifically declare what type of number you wish to work with.

Integer Types

Integer data types are for storing whole numbers. There are signed and unsigned variants of these, where the allocated size is the same but the range oof possible values is changed.

Variable Name	C Identifier	Size	Range
Character	char	8 bit	0 - 255
Integer	int	16 bit	± 32,000
Short	short	16 bit	± 32,000
Long	long	32 bit	± 2 billion

Floating Point Types

Floating point data types allow increasing levels of presicsion for calculations by providing more and more decimal places. For most engineering applications 15 decimal places will usually suffice.

Variable Name	C Identifier	Size	Precision
Float	float	32 bit	6 decimal places
Double	double	64 bit	15 decimal places
Long	long	80 bit	19 decimal places

Arithmetic

You will recall that a computer is defined as a machine that can perform arithmetic and logic operations. So far we've done a little bit of arithmetic, but let's look at how we can use variables to make our calculations more robust.

```
int main(){

  //declare an integer variable
  int a = 7;

  //return the int squared
  printf(a*a);

  return 0;
}
```

Great, let's try one more example. You code has compiled, it will execute from top to bottom. This means that new values can be assigned "on-the-fly" limiting the need for too many variables. In the following example, the `age` variable is first user to calculate Mark Zuckerberg's age, and then it is reused to calculate how old Bill Gates is.

```
int main(){

  int currentYear = 2017;
  int zuckerBorn = 1984;
  int babyGates = 1955;
  int age;

  //calculates how old Mark Zuckerberg is:
  age = currentYear - zuckerBorn;
    printf("Mark Zuckerberg is %d years old \n", age);

  //here we reuse the age variable to compute how old Bill Gates is.
  age = currentYear - babyGates;
    printf("Bill Gates is %d years old \n", age);

    return 0;

}
```

Input!!

So far we've been manipulating predefined data - this is all well and good but it doesn't make for particularly interactive programs. In this section we will look at the `scanf()` function, which is used for reading (or scanning!) in data from the console.

scanf() syntax

Both `scanf()` and `printf()` are part of the `<stdio.h>` library, or header file, so hopefully they will look quite similar to you. As we saw before, when printing data we needed to tell the function both the **type** of data we are working with and a **value** for that data to have. Take a look at the following simple example:

```
int myInt;

scanf("%d", &myInt);
printf("%d", myInt);
```

If you build and run the above example you will just see a console with a blinking cursor - not particularly intuitive for the end user. Typing a number will result in that number being echoed, or repeated back to the console. Notice that in the `scanf()` function that we are looking for an integet input and that the input should be stored in the *myInt* variable. This is specified by using the *address of* operator, in this case `&myInt`. Don't worry about it too much for now, we'll learn more about these operators in the next chapter.

Let's improve the user interface a bit but prompting the user for the type of input we'd like from them. Before scanning for data, lets use a `printf()` function to tell the user what to type:

```
int age;

printf("What year were you born in ? \n");
scanf("%d", &age);
printf("You were born in %d.", age);
```

Challenge

Improve the snippet above so that it asks the user for their year of birth and then returns their (approximate) age to them.

Summary

In this chapter we learned some of the fundamentals of C programming. At this stage in the course you should be able to:

- Use `printf()` and `scanf()` to ask the user for information and return a result to them,
- Include escape characters in your `printf()` statements,
- Recognise conversion characters and know which character correspond to which data type.

Working with Strings

Unlike other (primative) variable types, strings are instantiated using `String` with a capital S. This is because a string is actually an array of `chars` - in other words a collection of characters.

String Terminator

When you go to run your code the C compiler needs to know where every string ends, or terminates. To do this, the compiler parses your program and everywhere it finds closing quotation marks it insterts a character known as the **string terminator**. The string terminator is simply `\0` and takes up one additional byte of memory. This means that if you create a String variable to store your name, and if your name is 8 characters long, the name variable will actually take up 9 bytes of memory.

Overwriting Strings

To overwrite an existing string use the `strcpy()` function. This function takes two **arguments** or parameters. Remember, the `strcpy()` function can be used to overwrite *any* string, so the first thing you need to tell is is what string to replace, then you need to tell it what to replace it with.

Arrays.

Let's explore arrays in a bit more detail. As mentioned already, an array is simply a collection of data. The only syntactic difference between a primative data type and an array is an extra set of square brackets e.g.`int integerArray[];` Each new element of an array is seperated by a comma. For example, you might use an array to store your lotto numbers: `int luckyNums[] = [4,8,15,16,23,42];`

Indexing

What is we want to know,for example, what the third element of an array is? Well, we can look it up using it's **index**. Just like with a book, an index is used to look up information you want to find. Now intuitively you might think that the third element of an array would be found at index 3 - however C, like that vast majority of programming languages, start indexing arrays at 0. This means the the first elements is at index 0, the second is at index 1 and so on. So to print our third lottery number to the console we could write:

```
printf("%d", luckyNums[i]);
```

Preprocessor Directives

Up until now we haven't really paid any great attention to the `#include` code at the beginning of our programs. The `.h` files that are included are known as header files or preprocessor directives. This is because the contents of these files are called *before* the main.c is compiled.

Writing a simple header.

Lets create a header with some commonly used mathematical constants

```
DEFINE PI = 3.141593
DEFINE EULER = 1.6
```

If you're using an IDE you'll see that this new .h file is saved in a new folder called "headers". Typically your compiler will expect your headers to be in a predefined location —— and this is implied in the code by using brackets `#include <someHeader.h>` however, our hearder is stored in the same directory as our source code so acccess it we use quotation marks instead: `#include "myHeader.h"`

Input!!

So far we've been manipulating predefined data - this is all well and good but it doesn't make for particularly interactive programs. In this section we will look at the `scanf()` function, which is used for reading (or scanning!) in data from the console.

scanf() syntax

Both `scanf()` and `printf()` are part of the `<stdio.h>` library, or header file, so hopefully they will look quite similar to you. As we saw before, when printing data we needed to tell the function both the **type** of data we are working with and a **value** for that data to have. Take a look at the following simple example:

```
int myInt;

scanf("%d", &myInt);
```

If you build and run the previous example you will just see a console with a blinking cursor - not particularly intuitive for the end user.

```
int age;

printf("What year were you born in ? \n");
scanf("%d", age);
```

Challenge

Improve the snippet above so that it asks the user for their year of birth and then returns their (approximate) age to them.

stretch goal use system time and users D.o.B to get exact age, or to create a birthday countdown.