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.

```c
#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

```c
#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.

```c
#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.

```c
#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!

```c
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.

```c
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:

```c
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.

```c
//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	$\pm$ 32,000
Short	short	16 bit	$\pm$ 32,000
Long	long	32 bit	$\pm$ 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.

```c
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.

```c
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:

```c
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:

```c
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.

Hello C...TYI!

Domhnall O'Hanlon

February 14, 2017

Part 2: Arithmetic and Logic

Maths Operators

Also known as binary operators, these mathematical operators require two operands to produce an output. For example, $40 + 2 = 42$ requires two inputs to produce an output. As we will see later, programming languages also support unary opeartors.

Operation	Symbol
addition	$+$
subtraction	-
multiplication	$*$
division	
tabularnewline modulo	$\%$
equivalence	$==$

A simple incrementer

One of the most common tasks you will encounter in programming (in any language) is incrementing a value i.e. increasing it by a specific amount. Take for example scoring points in a game. Every time you hit a pig in Angry Birds 5000 points are added to your score.

Try the following code snippet:

```
//initialise a variable to store the number of times prinf is run
int numPrints = 0;

//increment the variable by one
numPrints = numPrints + 1;
  printf("The printf function has run %d times \n", numPrints);
```

```
numPrints = numPrints + 1;
  printf("The printf function has run %d times \n", numPrints);

numPrints = numPrints + 1;
  printf("The printf function has run %d times \n", numPrints);
```

i++

The most common amount to increment a variable by is one, in fact it's so common that there's a shorthand for it. All you have to do is write it in the form `myVariable++`, so the example above becomes:

```
int numPrints = 0;

numPrints++;
  printf("The printf function has run %d times \n", numPrints);

numPrints++;
  printf("The printf function has run %d times \n", numPrints);

numPrints++;
  printf("The printf function has run %d times \n", numPrints);
```

Unary Operators

Unlike binary operators, unary operators only require one operand. They allow you to write your code more concisely. For example, `i++` or `i--` will incrememt or decrement the integer `i`.

There are serveral other ways of concisely expressing arithmetic operations, but they are binary rather than unary. For example, if we want to implement the Angry Birds scoring system we could use `myScore+=5000` to increase the score in steps of 5000.

Table of Unary Operators

In the previous chapter we saw the **Address Of** unary operator when we learned about the `scanf()` function. We were able to scan information from the console and store it in a variable called myInt using the following line of code:

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

The table below sets out the rest of the unary operators.

Operation	Symbol	Also
Increment	x++	++x
Decrement	x–	–x
Address	&x	
Pointer	*x	
Positive	+x	
Negative	-x	
Ones Complement	~x	
Logical negation	!x	
Variable Size	sizeof x	sizeof(type-name)
Type casting	(type-name)	

Performing Calculations in printf()

You can create functions to perform maths operations almost anywhere in your program. It is also worth knowing that you can perform calculations within other functions such as `printf()`. Try the following example:

```
printf("Pi is approximately %f", 22/7);
```

Working with Modulo

You should be familiar with the first four mathematical operations, but modulo may be new to you. Put simply, modulo tells you the remainder of dividing one number by another. For example `7%2` would return `1`, since 2 goes into 7 three time with a remainder of 1. Similarly `6%3` would return `0` since 3 divides evenly into six. In general terms, the modulo operator will always return a number between 0 and d-1, where d is the divisor (or denominator, if you prefer to think in fractions).

Order of Operations

Multiplication is just a concise way of saying that you want to repeatedly add a number to itself, and similarly division is just a simpler way of saying you

want to repeatedly subtract a number. Then when multiplication and division become too cumbersome to work with you can use exponents and radicals (roots) to indicate repeated multiplication and division, respectively. This heirarchy is the reason you had to memorise some acronym for the order of mathematical operations.

PEMDAS	BOMDAS	BIMDAS
Parentheses	Brackets	Brackets
Exponents	Orders	Indices
Multiplication	Multiplication	Multiplication
Division	Division	Division
Addition	Addition	Addition
Subtraction	Subtraction	Subtraction

Order of Operations

Unlike many calculators, C is intelligent enough to understand this order and will apply it when making calculations. This means that if you want some particular step of your calculation to happen in a certain order then you will have to make careful use of brackets.

Try the following snippet as an example and see if you can come up with some others.

```
printf(3 + 5 * 7); //returns 38
printf((3+5) * 7); //returns 56
```

Programming Challenges

Here are a few simple scenarios to challenge your understanding of the programming concepts we've covered so far.

Grade Point Average

Write a simple command line application that accepts 6 integer grades and returns the average of these numbers.

Tip Calculator

Write a command line application that accepts a bill amount and return to the user both the value of a 10% tip and the combined amount of initial bill and tip.

Logic

This section will help you add some "intelligence" or decision making abilities your programs. By the end of this section you will be able to write programs that respond to a variety of different input conditions, and we will conclude this chapter by writing a very simple game.

If Statements

Sometimes refered to as branches, if statements contain code that only executes if a certain condition is met. A nice example of an app that makes decisions based on specific events happening is called "IFTTT" which stands for **If This Then That**. As a more everyday example you can imagine the following scenario: "If it's raining outside then bring an umbrella"

Syntax

A typical if statment will look like so:

```
  if(test if true){
     code to run if test is true;
}
```

Try the following out:

```
int input;

  printf("What is the meaning of life, the universe and everything? \n")?
  scanf("%d", &input);

  if (input == 42){
    prinf("Such learning. Many wisdom. Wow");
}
```

What ELSE can we do?

In the previous example there was only one "correct" answer. When you run the program you only ever get a response if the number '42' is entered. For every other input the program remains silent.

By adding an else condition we can catch all the other alternatives that our if test misses.

```
 if(input == 42){
    printf("Such Learning Many Wisdom. Wow");
```

```
} else{
    printf("Try again");
}
```

Else If

Finally, we can run more than two tests by adding in one (or more) `else if`
cases. For example:

```
if(input == 42){
    printf("Such Learning Many Wisdom. Wow");
} else if(input == 43){
    printf("Close, but no cigar");
} else if(input == 41){
    printf("Close, but no cigar");
} else{
    printf("Try again");
}
```

Ternary Operator

As with most things in programming, there's a more concise way to write your if
statements. We've already seen the a unary operator take one operand, a binary
operator takes two and with a ternary operator we can use three operands to
handle the "If-Then-Else" elements of an if statement.

```
(test) ? trueCode : falseCode;
```

The ternary operator, just like unary and binary operators, can be used as an
argument in other functions such as `printf()`. Try this nice way to print plurals
correctly!

```
int numFriends = 2;

printf("You have %d friend%s", numFriends, (numFriends!=1) ? "s." : "." );
```

Coding Challenge!

Sorting...Sort of.

Early entrance is changing - students will now be divided by surname, write a
program that checks the first character of a lastName string, A-N, M-Z

Truth Tables

Hopefully you will remember truth tables from your electronics studies in EM113. Just in case you've forgotten, here's what the `OR` and `AND` truth tables look like for two inputs.

A OR B

A	B	A+B
0	0	0
1	0	1
0	1	1
1	1	1

A AND B

A	B	A.B
0	0	0
1	0	0
0	1	0
1	1	1

Logic Operators

In your logic tests you can test if more than one condition is met using a logic `AND` operator or you can test to see if either condition is true using the logic `OR` operator.

Here's some pseudo-code to illustrate:

```
  //logic and example
  if (test 1 && test 2){
    code if both are true;
}
```

Logic and is denoted by `&&` which is Shift + 7 on a UK keyboard. Logic or is denoted by `||` which is Shift + \ on a standard keyboard.

Coding Challenges

Try these two challenges to test your knowledge of what we covered in this chapter:

CAO Points Calculator

Write an application that asks for an integer input (between 0 and 100) and converts it the corresponding Leaving Cert grade.

Fizz Buzz

need to introduce for loops first

Print all the numbers from 0 to 30 inclusive. If the number is evenly divisibe by 3 print "FIZZ" instead of the number If it's evenly divisible by 5 then print "BUZZ" instead of that number. If the number is divisible by both 3 && 5 the print FIZZBUZZ

Summary

After this second chapter you should now be able to:

- Understand binary operators and use them in your programs,
- Recognise the unary operators, know where to use them and be able to use the Address Of unary operator,
- Write If-ElseIf-Else statements,
- Use ternary operators instead of IF/Else statements,
- Use logic AND in your programs,
- Use logic OR in your programs.

This page is intentionally left blank

Hello C...TYI!

Domhnall O'Hanlon

February 21, 2017

Part 3: Loops

Overview

We've already seen a few instances where we wanted to repeatedly print information to the screen. Performing repetitive tasks is one of the things that computers are exceptionally good at doing. In this chapter we'll introduce a variety of different loops that you can use to write better programs.

Looping

If you've never done any programming before loops can be a challenging topic due to their unfamiliarity. If you think about it in more general terms, how would you give a computer instruction to do something over and over again? How would you avoid getting stuck in an *infinite loop*?

Start, Middle, End

When you creating a loop in any programming language you will have tell it when and where to start. Starting a loop is known as **initialising** the loop.

Next you will have some code to run - for example increment at counter, print some text etc. - while the loop is running.

Finally you need an end condition. Once this condition is met or exceeded then the loop should exit gracefully.
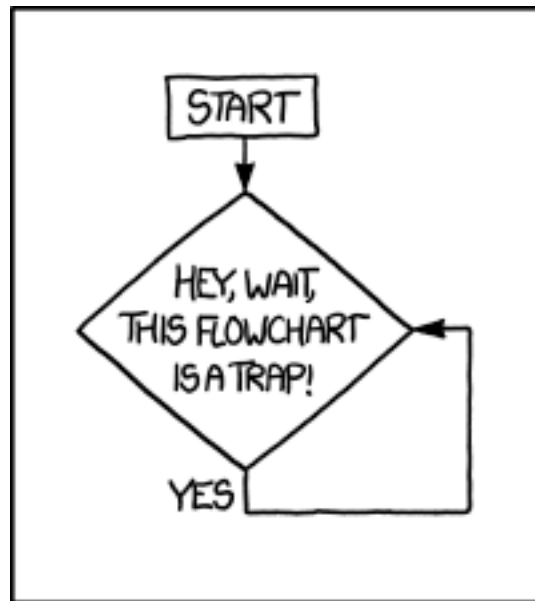
Figure 1: flowchart

While Loops

A `while` loop, as the name implies, executes while a certains condition is true. Once the test condition is no longer true then the loop is broken and code execution moves on to the next line. Try this snippet to get started:

```c
int main(){

    int counter = 0;

        while(counter < 10){
            printf("the value of the counter is: %d \n", counter);
            counter++
        }

    printf("successfully exited the while loop! \n");

    return 0;
}
```

Syntax

A typical `while` loop will begin with a test condition:

```
    while(testIfTrue){
        code to run while true;
        incrementer;
}
```

If the test is true then the code inside the loop (i.e between the braces) will run - keep in mind that each line must end with a semi-colon. Finally, you'll need to have some sort of incrementer that gets updated during each pass through the loop. This is essential so that you don't get stuck in an infinte loop.

Example

Try the previous example with different test conditions. For example using `while (counter <= 30)` will cause the loop to run an extra time. Similarly you could continue executing the loop while the counter is *not* equal to a certain value. `while (counter != 30)`

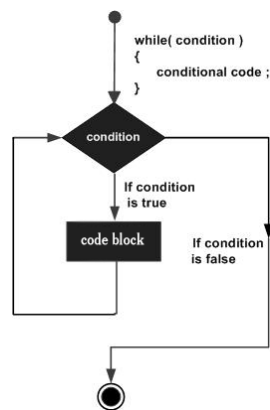Be careful if you are changing the direction of the inequality!

Visualisation

Figure 2: while loop

Programming Challenge

Here's a simple example you can code using `while` loops. Which would you preferm, one million euro today or 1 cent, doubled every day for a month (30

days)?

Your loop should run 30 times, doubling the value of your variable each time.

Do While

With a `while` loop there is always a possibility that the test condition will never be true and that the code within the loop will never run.

A `do while` loop differs from a `while` loop in that it will always run at least once, and the conditional check is performed at the end of the loop, rather than at the beginning.

Do While Syntax

```c
int loopCounter = 1;

/* do loop execution */
do
{
    printf("number of times this loop has run: %d\n", loopCounter);
    loopCounter++;
}while( loopCounter < 10 );
```
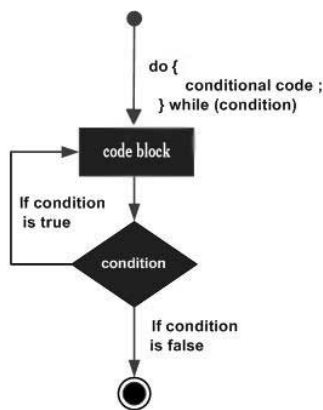
Visualisation

Figure 3: Do While loop

For Loops

The `for` loop contains the starting condition, end condition and incrementer all at the beginning of the loop

Syntax

```
int main(){

    int counter;

    for(counter = 0; counter < 10; counter++){
        printf("Hello World!");
    }

    return 0;
}
```
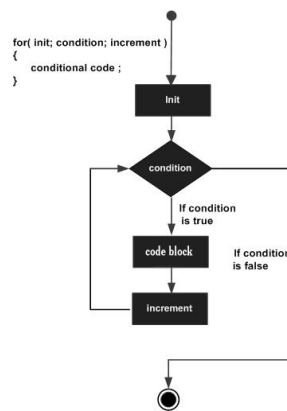
Visualisation

Figure 4: for loop

Example

```
/* Print all the even numbers between 0 and 100 in 3 lines of code */
int main(){

    int i;
```

25

```
    for(i = 0; i <=100; i++){
        if(i%2 == 0){
            printf("%d \n", i);
        }
    }

    return 0;
}
```

Nesting For loops

A quick challenge to really test your understanding so far.

Create a program that has variables to represent the number of rows and number of colums that a table should have. Then use nested for loop to print a 3 x 3 table to the console.

Break

Lets say that we have some condidtion which, if met, should break us out of our loop immediatley. In such a scenario we would use a `break;` statement. Here's a simple example that modifies out doWhile application to exit before it has iterated through the loop 10 times.

```
int main(){

    int loopCounter = 1;

    /* do loop execution */
    do
    {
        if(loopCounter == 7){
            break;
        }
        printf("number of times this loop has run: %d\n", loopCounter);
        loopCounter++;
    }while( loopCounter < 10 );


    return 0;
}
```

Continue

Conceptually this is the opposite of a `break` statement. It's behaviour will differ slightly depending on where it is used. In a `for` loop the `continue` statement will cause the conditional test and increment portions of the loop to execute.

When used in a `while` or `do-while` loop, the `continue` statement causes the program control to pass straight to the conditional tests.

Switch

Switch statements are used to test a variable for equivalence against a list of given values. It is conceptually similar to an `if-else if-else` block of code.
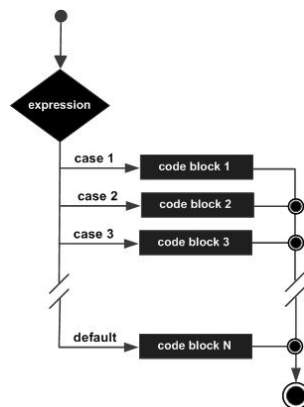
Visualisation

Figure 5: switch

Example 1

```
int main(){
    int input;
    int type;

    printf("Enter a number \n");
    scanf("%d", &input);

    if(input % 2 == 0){
        type = 0;
```

27

```c
    }else{
        type = 1;
    }

    switch(type){
        case(0):
            printf("The number is even\n");
            break;
        case(1):
            printf("The number is odd\n");
            break;
        default:
            printf("Sorry, unknown type of number! \n");
            break;
        }

    return 0;
```

Example 2

```c
switch(grade)
{
case 'A' :
    printf("Excellent!\n" );
    break;
case 'B' :
case 'C' :
    printf("Well done\n" );
    break;
case 'D' :
    printf("You passed\n" );
    break;
case 'F' :
    printf("Better try again\n" );
    break;
default :
    printf("Invalid grade\n" );
}
```

Fizz Buzz Revisited

Using a loop of your choosing:

- Print all the numbers from 0 to 30 inclusive.
- If the number is evenly divisibe by 3 print "FIZZ" instead of the number

- If it's evenly divisible by 5 then print "BUZZ" instead of that number.
- If the number is divisible by both 3 && 5 the print FIZZBUZZ

Summary

After reading this chapter and completing all the exercises in it you should be able to:

- Create a For loop,
- Understand the differences between a While and Do While loop, and how to use both,
- Use a Switch statment,
- Correctly use the break and continue statements.

Hello C...TYI!

Domhnall O'Hanlon

February 28, 2017

Part 4: Arrays & Strings

Up until now we have typically worked with only one piece of information - for example integers, floats, chars etc. In many cases - such as our grade calculator - all the information was of the same type, yet we still created seperate variables to store individual grades. Wouldn't it be much simpler if we could collect similar information like this in the same place?

Collections

An array is just another word of a collection. Examples of arrays can be found in mathematics, astronomy, and even biology. What sort of things do people typically collect? What sort of collections can you think of?

In the case of all of these collections, each consists of the same **type** of thing - stamps, coins etc. In programming that same is true. A collection of data, or an **array** must contain items that all have the same data type. For example, an array of ints could contain people's ages, an array of floats might contain bank balances, or an array of chars could store a student's letter grades.

Initialising

What sort of information do you need to know in order to be able to create, or initialise, a loop?

Like any variable, it will need a name. You also need to know what sort of information you're working with, so you have to know variable type. Finally you need to know how many items should go in the array.

Typically the syntax for an array is as follows:

```c
int numbers[5] = {1,2,3,5,8};

char letters[3] = {'A', 'B', 'C'};

float myArray[10];
```

Accessing Elements

It is important to note that the first item of an array has an index of 0 (lives at index 0?) as this is often the source of off-by-one errors. For example,to print the letter **A** from the array `letters[3]` above, you select the 0th item from the array with the following piece of code:

```c
printf("%c \n ", letters[0]);
```

Combining with Loops

You can quickly scan items to or prints items from an array by using an incrementer to both keep track of the iterations of your loop and the location (index) in the array that you want to access.

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

int main()
{
    int values[5];
    int counter;

    /*read 5 integers into the "values" array*/
    printf("Please enter 5 values\n");
        for(counter = 0; counter < 5; counter++){
                scanf("%d", &values[counter]);
            }
```

```
    /*print out the values from the array*/
    printf("\n The values you entered were: \n");
        for(counter = 0; counter < 5; counter++){
            printf("%d\n", values[counter]);
        }


    return 0;
}
```

Working with unknows

Let's say that you want your user to enter all their test results - the only problem
is you don't know in advance how many exams they've taken. Take a look at
the code below and see if you can understand what's going on.

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

int main()
{
    int inputs;
    printf("How many exams have you taken?\n");
    scanf("%d", &inputs);

    int results[inputs] = {};
    int index;

    printf("Please enter your grades in any order:\n");
        for (index = 0; index < inputs; inputs++)
        {
            scanf("%d", &results[index]);
        }

    for (index = 0; index < (sizeof(results)/sizeof(results[0])); index++){
        printf("Result %d is : %d\n", index+1, results[index]);
    }

    return 0;
}
```

Programming Challenges Using Arrays

Strings

In other lanugages, such as Java, there are dedicated *String* data types, but in C the convention is to use an array of chars, which does essentially the exact same thing. A C string is any array of chars followed by the null character, `\0`. The null character is also known as the string terminator (see figure 1). 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. This section outlines a number of different ways that strings can be created in C, as well as looking at how to manipulate stings with functions from the `string.h` library.

Array of Chars

The long way of creating a string is one character at a time so, if you really want to, you *could* do the following:

```
char str1[20] = {'H','e','l','l','o',' ','W','o','r','l','d', '!'};
```

Note that when working with chars that each element of the array has to be inside single quotes. A more concise way to acheive exactly the same thing is by enclosing the entire string in double quotes like so:

```
char str2[20] = "Hello World!";
```

Try creating a program that includes `str1` and `str2` and prints the both to the console.

string.h

Since the original C language doesn't contain functions for working with strings, much of this functionality is provided by the `string.h` library.

Copying 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 (overwrite), then you need to tell it what to replace it with i.e. the string you wish to duplicate.

What do you expect the output of the following piece of code will be?

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

int main()
{
    /* code */
  char str1[20] = {'H','e','l','l','o',' ','W','o','r','l','d', '!'};

  char str2[20] = "Goodbye World!";

  printf("String 1 is: %s\n", str1);
  printf("String 2 is: %s\n", str2);

  /* Overwrite string 1 with the contents of string 2 */
  strcpy(str1, str2);

  /* display them again */
  printf("String 1 is: %s\n", str1);
  printf("String 2 is: %s\n", str2);



    return 0;
}
```

Measuring Strings

To find the length of a string simply use `strlen()` function. This function only accepts one argument, the string you want to measure the length of, and it returns an integer value with the length of the string.

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

int main() {
    char greeting[50] = "Hello World";
    int buffer_size;

    printf("%s\n", greeting);
    buffer_size = strlen(greeting);
    printf("The greeting is %d characters in length", buffer_size);
```

```c
    return 0;
}
```

Write a program that asks the user for two strings and then tells them which string is longer.

Joining Strings

In many areas, particularly when working with databases, you frequently have to put two (or more) strings of text together. This joining operation is known as **concatenation**. You can concatenate two strings by using the `strcat()` function. Write a program that ask the user for their first name and their second name and concatenates these two strings, with a space in between.

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

int main() {

    char name[100], lastName[100];

    printf("What is your first name?\n");
    scanf("%s", name);

    printf("What is your second name?\n");
    scanf("%s", lastName);

    //add a space after firstName
    strcat(name, " ");

    //concatenate
    strcat(name, lastName);

    printf("Hello %s \n", name);
    return 0;
}
```

Comparing Strings

You can check if two strings are identical or not by using the string compare function `strcmp()`. As you can probably imageine, this function requires two arguments - the two strings you want to compare. If, for example, you want to compare *str1* and *str2* there are three possile outcomes. They're either identical, or string 1 might be smaller than string 2, or string 1 might be bigger than string

37

2. The `strcmp(str1, str2)` function returns 0 if both strings are identical, a negative number if *str1* is less than *str2* or a positive number if *str1* is greater than *str2*.

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

int main() {
    char s1[32] = "apples";
    char s2[32] = "oranges";
    char s3[32] = "apples";
    int result;

    result = strcmp(s1, s2);
    printf("Comparing %s and %s returned %d\n", s1, s2, result);

    result = strcmp(s1, s3);
    printf("Comparing %s and %s returned %d\n", s1, s3, result);

    if(result > 0){
        printf("%s is longer than %s\n", s1, s3);
    }
    else if(result < 0){
        printf("%s is longer than %s\n", s3, s1);
    }
    else{
        printf("%s and %s are identical\n", s1, s3);
    }

    return 0;
}
```

Coding challenge

Create a simple password app. Your code should should include a user-configurable access key. You should also make sure that your user can not enter a password longer than 20 characters.

```c
 char key[20] = "YourPassword";
```

and some way to check if the users password is the same as your stored key.

More About scanf()

In the case of both printf() and scanf() the f stands for *formatted*. When you use `scanf()` the function reads in characters from the console until it encounters a **space** or a **new line**. This means that for things like sentences, or even just someone's full name, the `scanf()` function as we have been using it so far will not work.

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

int main() {

    char fullName[100];

    printf("\nWhat is your full name? \n");
    scanf("%100[^\n]s", name);
    printf("You typed \n%s\n", fullName);

    return 0;
}
```

In the example above, we use `%s` to specify that we are reading in a string, and then send it to the `fullName` buffer, just like with any other string we've seen so far. The two new things added here are actually between the `%` and `s`. The first value, `100`, is simply the buffer size. Since the fullName string can only contain 100 characters we limit the number of characters that the user can enter to 100 to avoid buffer overflow. If you were using a larger or smaller buffer you'd adjust this number accordingly. The second value `[^\n]` tells scanf to keep reading until it encounters a new line, i.e., until someone hits the enter key.

Summary

Having read this chapter and completed the programming exercises you shold now be able to:

Hello C...TYI!

Domhnall O'Hanlon

March 7, 2017

Functions

As you've no doubt seen during your maths studies, a function is something that takes one or more arguments (inputs) and produces some return (output). In C you can of course create lots of different types of functions, using any of the data types that the language supports, but for the purpose learning about functions some mathematical examples are quite useful.

Creating Functions

So, what data type does a function have? Well, that depends on what type of data it returns. If you want to create a function that squares two numbers and returns the result then it will probably have an integer type. If you want to get the average or root of some numbers then it makes more sense to use a float. The process of creating a function is known as **declaring** a function. Typically, functions are declared just below any `#include` calls at the beginning of your code. This declaration tells the compiler not just what return type to expect but all the number and type of arguments the function accepts. This can also be refered to as the function **prototype**. The code associated with how the function actually performs is known as the **definition** of the function. Let's create a simple function that squares an integer input:

```c
#include <stdio.h>

int num, result;
int square(int i);

int main(){

    printf("Please enter a number to square\n");
    scanf("%d", &num);

    result = square(num);
```

```c
        printf("%d squared is %d \n",num, result );
        return 0;
 }

   int square(int i){
      return i*i;
}
```

Maths

Create functions to calculate the area of a square, circle, cube and sphere.

Square

see the first example for squaring number.

Circle

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

#define PI 3.141593

float circle();

float result;

int main() {

    circle();
    printf("The area is %.2f units squared.\n\n", result);
}

float circle(){
    float rad;
    printf("What is the radius of the circle?\n");
    scanf("%f", &rad);
    result = PI*rad*rad;
    return result;
}
```

Cube & Sphere

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

#define PI 3.141593

float circle();
float cube();
float sphere();

float result;

int main() {

    cube();
    printf("The area is %.2f units squared.\n\n", result);

    sphere();
    printf("The area is %.2f units squared.\n\n", result);
}



float cube(){
    float side;
    printf("Enter the side length: \n");
    scanf("%f", &side);
    result = 6 * side * side;
    return result;
}


float sphere(){
    float rad;
    printf("What is the radius of the circle?\n");
    scanf("%f", &rad);
    result = 4*PI*rad*rad*rad;
    result = result/3;
    return result;
}
```

Summary

Having read this chapter, and completed all the programming exercises, you should now be comfortable with creating your own functions and calling them from anywhere in your main method.

Hello C...TYI!

Domhnall O'Hanlon

March 14, 2017

Pointers

The topic of pointers in C is one that many beginners find challenging, but once you get comfortable with them your applications will become more powerful and more efficient.

Welcome to the Hotel California

A pointer is a special type of variable (it's actually a constant!) whose value is the memory address of another variable. To help you visualise this, image a hotel with lots of floors, and many rooms on each floor. The rooms are analogous to locations in computer memory in that they don't ever change their pyhsical location or address. The occupants of the rooms however can change - so your hotel visitors are like programming variables, free to come and go as they please and easily replaced by other variables. ~~We can also extend this analogy a bit by considering twin rooms, double rooms, suites etc. These types of rooms might correspond to different types of data~~

A pointer variable is declared by prefacing its name with an asterisk. The address in memory is a numeric value so it's ok to use an interger as the type, but by starting your variable name with an asterisk (star) rather an a letter you are telling the compiler that this variable is actually a pointer rather than a conventional integer.

```
int main(){
    int *pointer;

    return 0;
}
```

You can still declare multiple varaible on one line, but each pointer has to start with *

```c
int main(){
    /*one pointer and one integer*/
    int *my_pointer, not_a_pointer;

    /*two pointers*/
    int *first_pointer, *second_pointer;
}
```

Convention

In many cases the accepted convention is to start a pointer variable with the letter p, for example:

```c
int main(){
    //create an integer variable
    int age;

    //create a pointer variable
    int *pAge;

    return 0;
}
```

Where are variables stored in memory?

To acces memory locations in C use a pointer thingy:

%p

We can echo a memory address to the console using something like the following snippet:

```c
    int i = 42;

    printf("%p ", i);

    return 0;
```

Some notes on Hex.

Creating Pointers:

Take for example: `int myInt = 42`, this can store any numeric value between x and y. It has a memory address and we can create `int * pMyInt = &myInt`

Dereference Pointer

If you want to retrieve the value that a pointer points to (as opposed to the address that it points to) then you use the unary operator, *, which is used to dereference the pointer.

```
int i = 42;
int *prt = &i;

int main(){
    printf("The address of int i is: %p \n", prt);
    printf("The value stored  at i is: %d \n", *prt);

    return 0;
}
```

Arrays and Pointers

create and array of vars loop through the array and print it's contents, and memory pointer.

int luckyNums[6] = [4,8,15,16,23,42];

```
printf(" Element \t Address \t Value")
for (i=0; i<6; i++){
   printf(" luckyNums[%d] \t %p \t %d", i, &luckyNums[i], luckyNums[i]);
}

//array names are just pointers to the first element of that array.
printf("\n luckyNums %p", luckyNums);
```

Strings and Pointers

Test it Out

```
int main(){
    j = 1;
    k = 2;
    ptr = &k;
    printf("\n");
    printf("j has the value %d and is stored at %p\n", j, (void *)&j);
    printf("k has the value %d and is stored at %p\n", k, (void *)&k);
    printf("ptr has the value %p and is stored at %p\n", ptr, (void *)&ptr);
    printf("The value of the integer pointed to by ptr is %d\n", *ptr);
```

```
    return 0;
}
```

Summary

Hello C...TYI!

Domhnall O'Hanlon

March 21, 2017

Why Structures?

We've seen lots of examples so far of where we would want to store multiple pieces of information. Up until now we've always used an array, however the main drawback of using an array is that it can only store one **type** of information, like an array of ints or an array of chars. The main advantage of using a struct is that it allows you to use different types of data.

Convention

The convention in C is to create your structs in a seperate header `.h` file, but for the time being we'll create them in our main.c files. In this lesson we'll create a basic car structure, make a few different cars and then modify them using a few functions we'll create ourselves.

Creating a struct

In something as complex as car, there are lots of different properties that we might want to assign. We can use strings to represent things like the make, model, features etc. numbers to represent properties such as horse power or top speed and arrays to capture the variety of engine sizes available, or perhaps types of fuel. The properties of a struct are contained between curly braces, and just like when you initialise any other data type, you must end your declaration with a semicolon.

```c
struct car{
    char* make;
    char* model;
    int bhp, topSpeed;
}; /* don't forget the semicolon */
```

Using a Struct

Now that we've created a prototype for all cars in general, we can now go ahead
and make specific cars by using our car struct.

```c
/* Add a new car by using our struct */
struct car bv = {"Bugatti", "Veyron", 1000, 407};
```

The Dot Operator

The dot operator, ., is used to assess specific members (elements) of a struct.
In the example of our car structure, it has properties, or elements corresponding
to the make, model, horse power and, top speed. Each of these can be accessed
using the syntax `name.element` like in the example below:

```c
printf("Make: \t %s\n", bv.make);
printf("Model: \t %s\n", bv.model);
printf("Horse Power: %d\n", bv.bhp);
printf("Top Speed \t %d \n", bv.topSpeed);
```

Typedef

```c
/* TYPEDEF: we don't have to be limited to just ints, chars, arrays etc - now that we can c
can create, or define, our own types using typedef */

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

typedef struct car{
    char* make;
    char* model;
    int bhp, topSpeed;
}car; /* don't forget the semicolon */

/*create a function which prints the top speed of any car */
void getTopSpeed(struct car anyCar){
    printf("The top speed of the %s %s is: %d km\\hr. \n",
        anyCar.make, anyCar.model, anyCar.topSpeed);
}

int main(int argc, char* argv[])
{
```

```c
    /* Creating cars without having to use struct all the time */
    car bv = {"Bugatti", "Veyron", 1000, 407};
    car vg = {"Volkswagen", "Golf", 227,170};

    /* Use the function to get the top speed of the cars */
    getTopSpeed(bv);
    getTopSpeed(vg);

    return 0;
}
```

Example Code

A quick example of sending data to functions and making it globally accessible throughout your application.

Here is the complete code for this section:

```c
/*
Now that we're using pointers the variables will update correctly.
 */

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

typedef struct person {
    char* name;
    int bankBalance;
} person;

void developApp();

int main(int argc, char* argv[])
{
    /* create a few people */
    person mz = {"Mark", 500};
    person es = {"Eduardo", 10000};

    developApp(&mz, 1000);

    /*print out mz.bankBalance here too */
    printf("In main() the balance is: %d ", mz.bankBalance);
    return 0;
}
```

```c
void developApp(person *somebody, int sales)
{
    printf("%s just launched a new app\n", somebody->name);

    somebody->bankBalance += sales;

    printf("%s's new net worth is: %d \n", somebody->name, somebody->bankBalance);
```

Summary