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>

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. For a more humorous take on semicolons see the following guide:

Comments

Obviously, as the weeks go on we'll be cerating 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 Bob said \"Hello World\" too.");

    return 0;
}
```

The `printf()` function will print out every character it sees inbetween 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

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, in other words **declare** it, so that you can refer to it elsewhere in your code. Finally you need to give your variable a value, or **initialise** it, somewhere in your code, otherwise you will get an *unused variable* error message from the compiler.

Types of Variables

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

Conversion Characters

Many functions, such as `printf()`, will allow us to substitute in varaibles rather than having to hard code in variables. Take the following example:

```
#include <stdio.h>

int main(){
  int myInt = 42;

  printf("Your integer is %d", myInt);

  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`. The `myInt` parameter tells the compiler which variable that should be. There are different **conversion characters** depending on which type of data you want to substitute into your function. The following table contains a complete list.

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 the first example we see how you can output a specific number of digits from the float pi. 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:

```
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. This is another example of where variables are extememly useful. The following code snippet has improved the previous example by including variables:

```
int main(){

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

  //initialise pi by assigning it a value of 3.141593
  pi = 3.141593;

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

  return 0;
}
```

Now, if we have to make any change to our program we simply have to change the varible once and all the functions that rely on it will get the updated value. To make your life easier you can also **declare** and **initialise** a variable in one line like so:

```
float pi = 3.141593;
```

Arrays

We'll explore arrays in more detail in chapter 4 so this is just a quick introduction. As you may know 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]);
```

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>` header, 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 name and then greets them personally.

Summary

Arithmetic & Logic

Domhnall O'Hanlon

2015

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	
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 operands. 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

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.

The `else` condition is included in the snipped below:

```
 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` clauses. 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##

Loops

Domhnall O'Hanlon

February 24, 2015

Part 3: Loops

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.

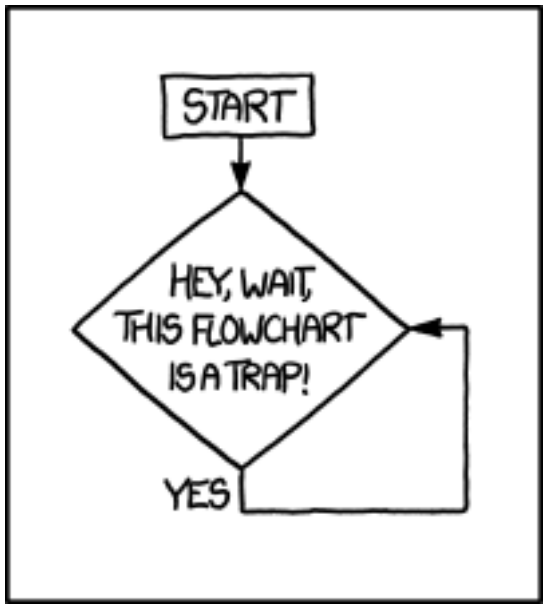


Figure 1: flowchart

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:

```
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

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.

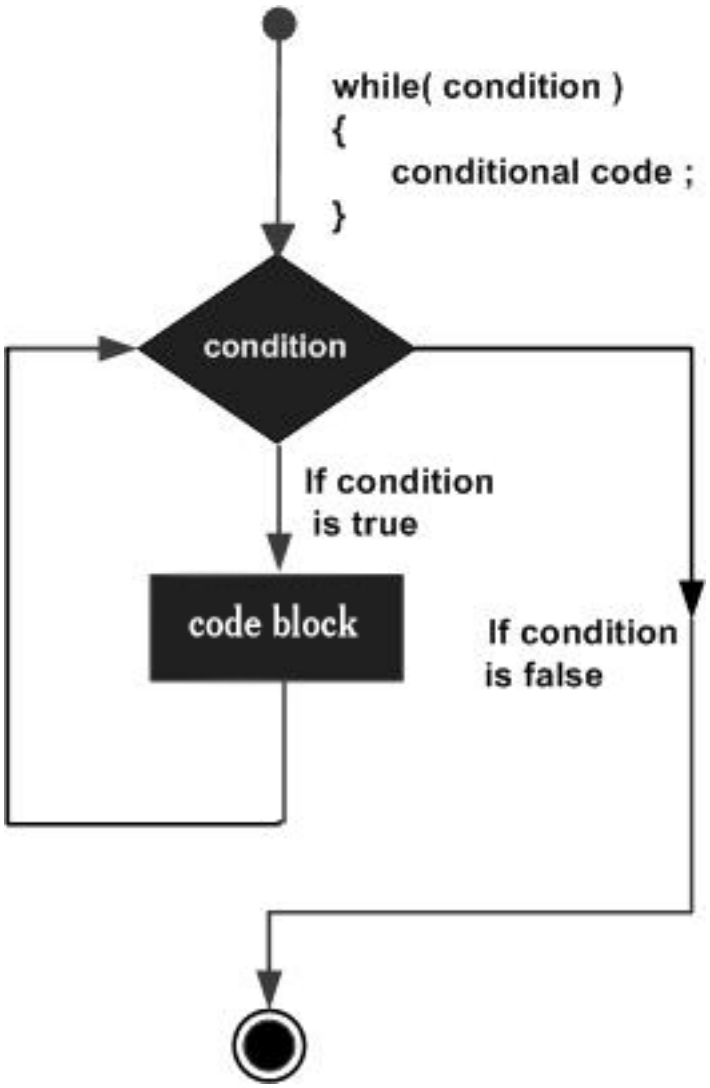


Figure 2: while loop

Do While Syntax

```
 int loopCounter = 1;

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

Visualisation

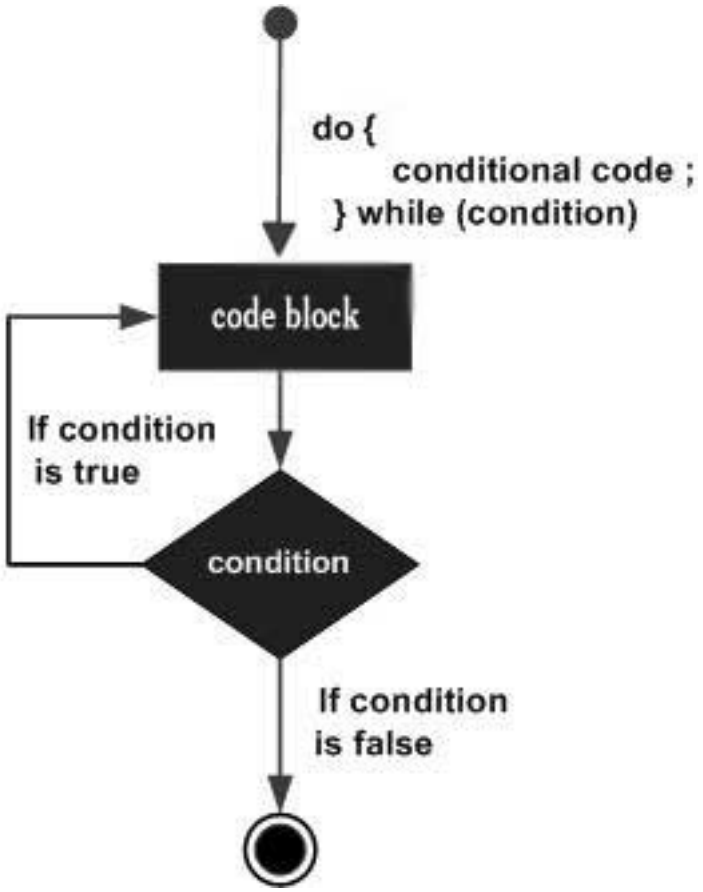


Figure 3: Do While loop

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

Example

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

     int i;

     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.

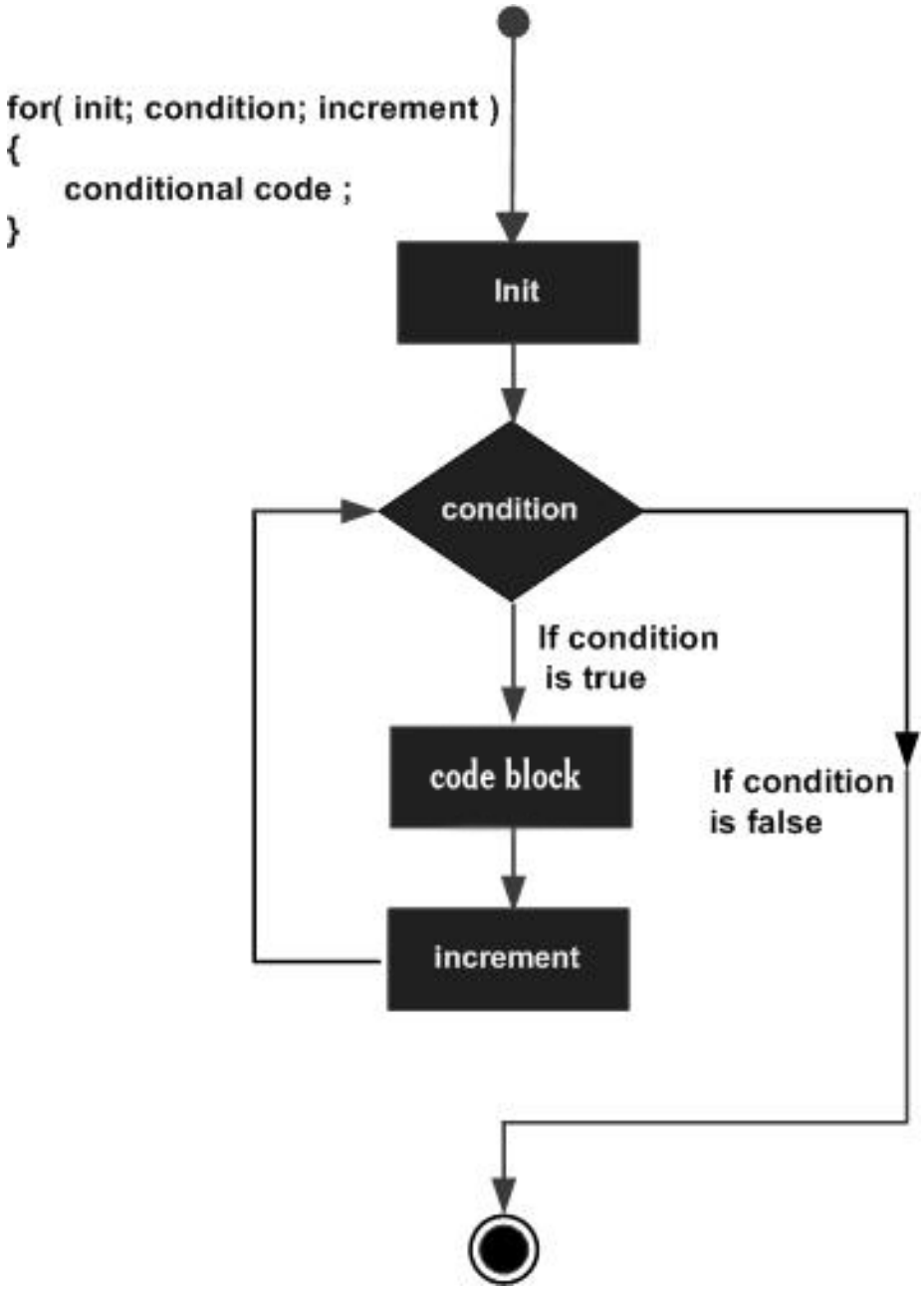


Figure 4: for loop

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

Example 1

```
int main(){
```

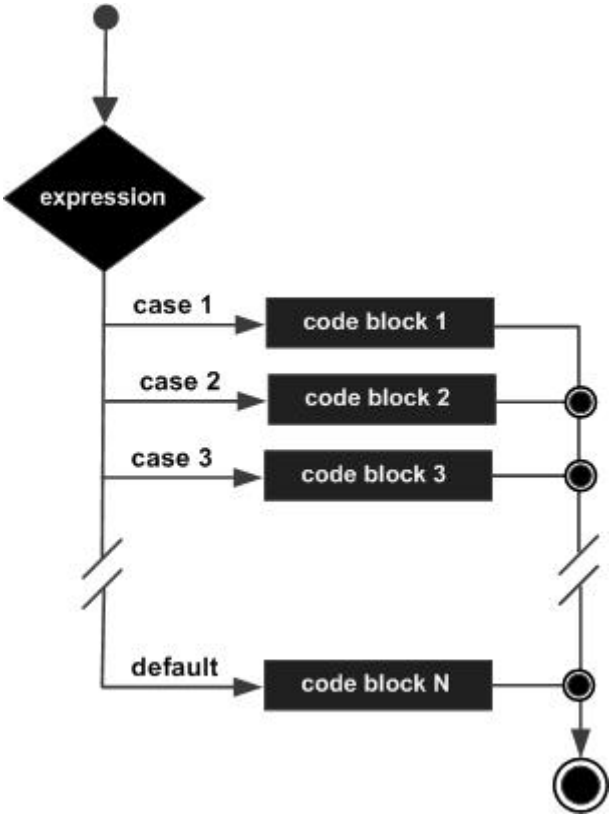


Figure 5: switch

```
int input;
int type;

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

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

```
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

Arrays

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 your 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:

```
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:

```
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.

Download Code

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.

Download Code

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 can 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 your message 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 gist will be?

Download Code

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.

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**. `strcat()`

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 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*.

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.

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

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

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:

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

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

Maths

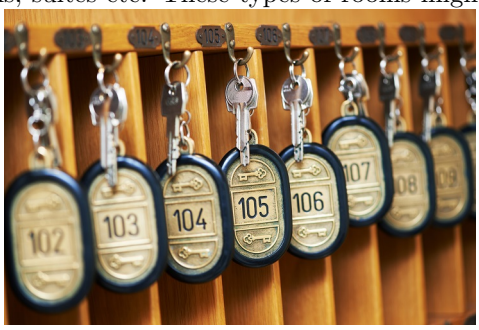
Create functions to calculate the area of a square, circle, cube and sphere.

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 `*`

```
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:

```
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:

```
    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;
}
```

Structs

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.

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.

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:

Typedef

With other data types

Example Code

Here is the complete code for this section:

And the code for changing values:

Using pointers to modify values globally, rather than confined to the scope of the `getTopSpeed()` function.

Reading & Writing to Disk

Motivation

Types of Files

Sequential

Random Access

Modifying Files

To modify a give file you must have the appropriate permissions.

r - read w - write a - append

Working with Files

`fopen()` opens an existing file or creates one if you tell it too "w"

`fprintf()` print data to your file.

`fclose()` close the files and frees up system memory once your finished writing to your file.