C CODING

Ultimate Step-By-Step Guide To Learning C Programming Fast

ROBERT ANDERSON

C
Coding

Ultimate Step-By-Step Guide To Learning C programming fast

Robert Anderson

TABLE OF CONTENTS

Chapter 1 5

Introduction and Installation 5

History of C 5

Running C on Windows 5

Running C on Linux 8

Running C on Mac 8

Running C online 9

Prerequisites 10

Chapter 2 11

Building Blocks 11

Variables and Types 11

Arithmetic operations 14

Conditionals 14

Iteration 21

Functions 29

Recursion 34

Arrays 35

User input 39

Quiz chapter 2 42

Chapter 3 43

Advanced Basics 43

Pointers 43

Pointer Arithmetic 46

Function pointers 49

Storage Classifications 51

File I/O 53

Exercise 55

Recursion Continued 56

Exercises chapter 3 58

Chapter 4 59

Custom Structures 59

Structures 59

TypeDef 63
Enums 64
Exercise 65
Unions 67
Variable argument lists 68
Exercise 70
Exercises chapter 4 72
Chapter 5 73
Advanced Features 73
Header files 73
Pre-Processor Directives 74
Error Handling 76
Type casting 79
Memory management 80
Exercise 84
Exercises chapter 5 86
Answers chapter 2 87
Answers chapter 3 88
Answers chapter 4 89
Answers chapter 5 90

Chapter 1
Introduction and Installation

History of C

C is a general-purposed computer programming language, that first appeared as a concept in 1972. The lead developer of C was Dennis Ritchie. The origin of C is closely related to the creation of the Unix OS.

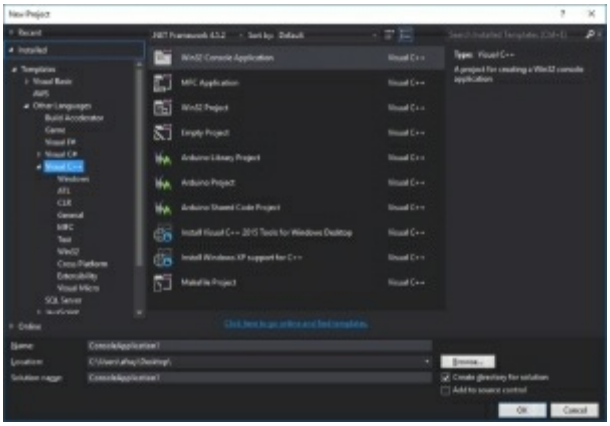
In this tutorial, we will go through the very basics of C right up to the intermediate level sections. The tutorials is designed to appeal to the first-time learners of a programming language.

Running C on Windows

Running C on Windows can be done with Visual Studio 2015. Visual Studio can be downloaded directly from Microsoft, through the installation process C++ needs to be selected as language. The menu should look like this:

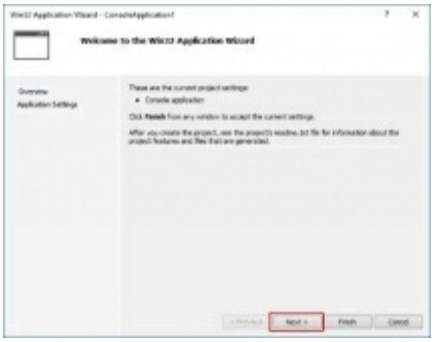
After you select ‘Visual C++’ fully install the program.

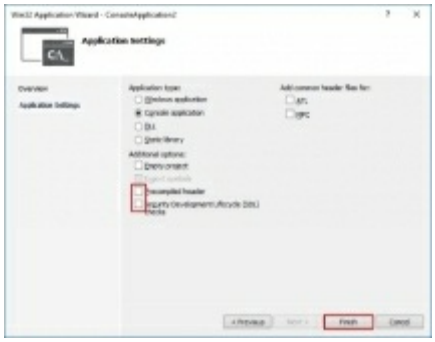
You should loads Visual Studio up and go to FILE -> New Project, you’ll be met by a Window like so:



Select ‘Win32 Console Application’ and click “Ok”

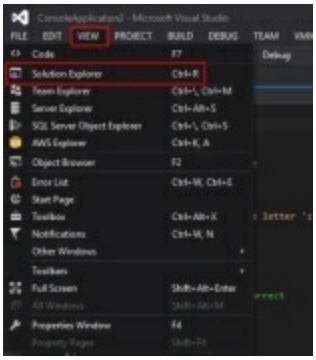
A few menus will now appear, follow them through with the highlighted sections. (The highlighted tix-boxes need to be *unchecked)*:

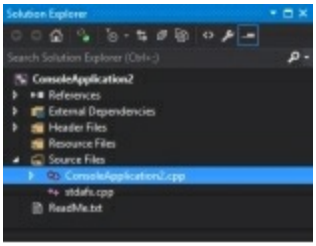




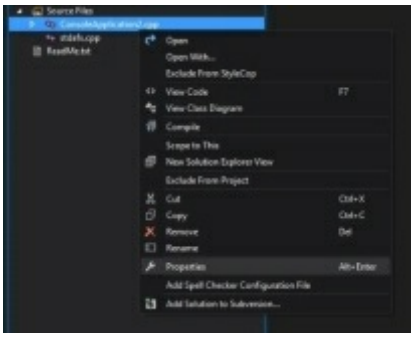
When the program template loads up you should notice you have not been provided with a C environment but a C++ one, you need to convert it to C:

Go to the “Solution explorer” and it will open the “Solution Explorer” Menu

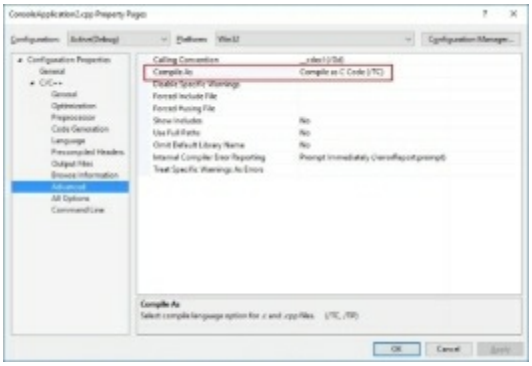




Right click on the **.cpp** file and click “Properties”



A menu will appear, expand the **C/C++** menu, go to “**Advanced**”, loop for the “**Compile As”** section and select from the drop down menu “**Compile as C Code (/TC)”**



The code is now ready to run! When the code you want to compile is reader click the green arrow on the top menu to run your console application.

Running C on Linux

Developing basic code on Linux is relatively easy first you'll need the '*gcc*' compiler, if you don't have it installed run:

```
apt-get install gcc-5
```

The next stage is to create your program, open up a text pad program (Leafpad in this case) use this basic program for testing purposes:

```
#import <stdio.h>
int main()
{
   printf("Hello, World!\n");

   return 0;
}
```

Save this file with the **.c** extension.

To compile and use the program you'll use *gcc* like so:

```
gcc -o <OutputProgramName> <C_FileName>
```

Where;

OutputProgtamName is the name of the executable you want

C_FileName is the name of your C file

To run the program just type the executables name followed by "./"

```
./<OutputProgramName>
```

Running C on Mac

Running C on a Mac is relatively easy, open up the *terminal* and run

```
clang --version
```

Clang is a compiler built by Apple and with deal with all the aspects of translating code into machine code (1's and 0's)

Start your notepad, write down your C code use this to start:

```
#import <stdio.h>
int main()
{
   printf("Hello, World!\n");

   return 0;
}
```

Save this code with the **.c** extension.

Open up a terminal in the folder where you saved you .c file, and type

```
make <filename>.c
```

Where filename is the name of the file you just saved. To run the code just run it like so:

```
./<filename>.c
```

Running C online

One of my favourite online IDE is https://www.codechef.com/ide it provides a clear clean place to produce code.

Prerequisites

Throughout this tutorial you will see code snippets dotted around. There's a few points that need explaining so you can get the most out of this tutorial:

Comments

Comments in code are signified by a "//" and the green colour, these are ignored by the computer and commenting is an important part of keeping computer code readable and easy to maintain. Starting coders tend to discard commenting and regret it later, don't be that programmer.

Basic structure

There is also a required basic structure of a C programme, the structure is below:

```
#include <stdio.h>

int main()
{
        //Code goes here
}
```

This is the structure to make your C programs run, don't worry we'll learn what it all means.

Chapter 2
Building Blocks

Variables and Types

The basis of a program is data, in the form of numbers and characters. The storing, manipulation and output of this data gives functionality. Variables hold this data, think of it as a box used to store a single item.

C is a heavily typed language where each variable must have a defined type. A type is an identifying keyword that defines what the variable can hold. The first type we will come across is the integer, this can hold real numbers (numbers without a decimal place), an integer is defined below:

```
int integerName = 3;
```

- **int** is the defined type keyword, we will learn about the difference possibilities later.

- "**integerName"** is the ID for the variable, this can be anything you want to call it, this is used to allow a variable to have a meaningful name, the variable could be defined as **"int pineapple = 3;"**, but it's good practise to make them relevant. However, there are a few exception to this as a variable cannot be a single digit i.e. '4' or cannot contain special characters (!"£$%^&*) etc.

- **"= 3;"** Is the assignment section, where the values of 3 is placed in the integer box for use later**. This also ends with a semi colon, this is used to signify the end of a line.**

This variable can now be used in valid areas of the program, like so:

```
int newInteger = integerName
```

The value of **intergerName** defined earlier (3) will now be placed in the **newInteger** variable and they will now both have the value of 3. The value of **integerName** doesn't change as it is just being copied and placed into **newInteger**.

String

The String is another crucial variable, this variable type is used to store a series of characters, an example could be the word “batman”, the word is composed of characters that are stored in the String variable. It’s an array of characters (More on arrays later), but effectively it’s the single characters of the word stored next to each other in memory. The ‘\n’ is a special character that stands for a newline.

```
char word[] = "dog\n";


//And it can be used and printed it like so:
printf(word);
```

Boolean

Booleans are used as expressions, their values can only be **true (1)** or **false (0)** and are used to signify certain states or flag certain events, they can also hold the result for a conditional expression (More on conditional expressions later). These will make more sense when we go thought conditional statements.

```
_Bool falseIsDetected = 0; //False


_Bool trueIsDetected = 1; //True
```

Float/Double

Floating point variables are decimal values created with float point precision math, the technical elements of how this works are outside of the scope of this tutorial but can easily be explained through resources on the internet, just search “floating point precision”. Floats allows for a higher level of accuracy

of a value by providing decimal precision. You can specify a ‘float’ value by putting an ‘f’ and the end of the value.

```
float decimalValue = 3.0f;
```

Void

This data type is special and is used to specify no value is available. This sounds counter intuitive, but we’ll see where this is used later.

Notable keywords/terms

const – This keyword turns the variable read-only, meaning it’s value cannot be changed after it is initialised. The keyword is used like so:

```
const float pi = 3.14f;
```

Global variable – This is a term describing a variable definition outside the main function. For example, the **int** friendCount is a global variable and the **int** currentMonth is not. Note the positions they’re defined:

```
//Global
int friendCount = 0;
int main()
{
        //Not Global
        int currentMonth = 5;
}
```

This means the global variable can be used in **any** location in the program and can be dangerous if not used correctly. One correct way it to use it in conjunction with the **const** keyword above, this means functions can only reference the value and not change it.

Recap

- int Holds real numbers
- float / double Hold decimal numbers
- void Specifies there is not type
- boolean Holds either true or false
- string An array of characters making up a word or sentence
- global A variable that can be accessed anywhere
- const Means after a variable has been initialised its value cannot be changed

Arithmetic operations

Symbol	Use
+	int result = 1 + 2;
-	int result = 1 – 2;
/	float result = 1 / 2; (This is stored in a float because of the decimal)
*	Int result = 1 * 2;
%	Modulus operator returns the remainder of a division int remain = 4 % 2;

Conditionals

Comparing values to other values in a meaningful way is fundamental for conditional statements, below is a list of comparisons of values.

Name	**Symbol**	**Detail**
Greater than	>	Returns true if left-hand side is larger than the right
Greater than or equal than	>=	Returns true if the left-hand side is larger or equal to the right
Less than	<	Returns true if the left-hand side is smaller than the right side
Less than or equal than	<=	Returns true if the left-hand side is smaller or equal to the left side
Equal	==	A double equal sign is used to compare if the value of either side is equivalent and returns true if equivalent
Not Equal	!=	Returns true if the two values are not equivalent

The list of conditionals above can be used in certain circumstances to control the flow of the program.

<u>If statement</u>

There are situations where you need something to happen if a certain condition is the case, this is the role of the If statement and where conditional statements come into the mix.

```
if (Condition)
{
    //Code will run here if Condition is True
}
//If the condition is false, the If statement is ignored and the program jumps here
```

A real-world example:

```
int compare = 10;
if (compare > 5)
{
    //printf() will print the string in the brackets
    printf("Code here will run!");
}
```

Output:

```
> Code here will run!
```

These allow a programmer to control the flow of the program and choose situations to happen when a possibility is true, we'll go onto more examples later.

Else Statement

Else's are optional but these can be added to the end of an if-statement and will only run if the if-statement condition is false

```
if (Condition)
{
    //Code will run here if Condition is True
}
else
{
    //Code will run here if Condition is False
}
```

Else statements can also become an else/if statement where a new if statement is attached, this look like this:

```
if (Condition1)
{
    //Code will run here if Condition1 is True
}
else if(Condition2)
{
    //Code will run here if Condition1 is False and Condition2 is True
}
//If neither are true no code will run
```

You can also chain another else statement onto an else statement effectively creating an infinite chain. If any conditions along the chain are true, the ones below are not checked, I'll demonstrate this below:

```
 int main()
{
        _Bool false = 0;
        _Bool true = 1;

        if (false)
        {
        printf("Condition 1");
        }
        else if (true) //This condition is true!
        {
        printf("Condition 2");
        }
        else if (true) //Ignored, due to previous else statement being true
        {
        printf("Condition 3");
        }
        else if (true) //Also ignored due to condition 2 being true
        {
        printf("Condition 4");
        }
}
```

Output

> Condition 2

After the body of condition 2 is hit and the **printf** statement is executed, the program does not go onto to check the other else statements.

Exercise

Create a program that prints out if a value is bigger or smaller than another, use this skeleton below program to get you started.

So in this case it should print “Value 1 is larger” (Note: Use **printf()** for printing

```
#include <stdio.h>

int main()
{
	//Values you change
	int value1 = 10;
	int value2 = 5;

	//PUT CODE HERE
}
```

Solution

This solution could be something like this:

```
#include <stdio.h>

int main()
{
	//Values you change
	int value1 = 10;
	int value2 = 10;


	if (value1 == value2)
	{
	printf("Values are equal!");
	}
	else if (value1 > value2)
	{
	printf("Value 1 is bigger!");
```

```
        }
        else
        {
        printf("Value 2 is bigger!");
        }
}
```

Just change the values of value1 and value2 before you run the program to test it.

Using multiple conditions

You can use more than one condition in a single statement, there're two ways of doing this **AND** signified by **&&** and **OR** signified by || (Double vertical bar).

AND checks if both conditions are true before triggering the body of the statement

```
if (Condition1 && Condition2)
{
    //Code will run here if Condition1 AND Condition2 are True
}
```

OR checks if one of the conditions are true before triggering the body of the statement

```
if (Condition1 || Condition2)
{
```

```
    //Code will run here if Condition1 OR Condition2 are True (Works if both are true)
}
```

Recap

- If statement Deals with conditional statements, code within it's body will run if the condition is true
- else statement Used as an extension to an if statement that is only checked if the if statement is false
- && Used to string two conditions together and will only return true if both are true
- || (Double Bar) Used to string two conditions together and will only return true if one of the conditions are true

Switch-Case

Switch cases are used to test a variable for equality with a constant expression without the need for multiple if-statements. One use for this structure is check users input string. Below is the basic structure of the switch case:

```
//Switch-Case
switch (expression)
{
//Case statement
case constant-expression:
        break; //Break isn't needed

//Any number of case statements
// |
// |
//\ /
// .
```

```
default:
        break;
}
```

The switch starts off with an ‘expression’, this is the variable that is to be compared to the ‘constant-expressions’, these constants are the literal values of the variable such as ‘1’ or ‘Z’. Breaks are optional but without them the code will ‘flow-down’ into other statements. There is an example below using a switch-case statement, the user inputs a character if the char is N or Y, ‘No’ and ‘Yes’ is output respectively but there’s a default case that applies for all situations, this needs to be at the end.

```
char character;

//Reads in user input (Explained in more detail later)
scanf("%s", &character);

        //Switch-Case
switch (character)
{
         case 'N':
        printf("NO\n");
        break;
        case 'Y':
        printf("YES\n");
        break;
        default:
        printf("Do not understand!\n");
        break;
}
```

If there is no **break** statement a ‘flow-down’ will occur, below is an example just like above but without the break statements and we’ll what the output is like:

```
char character;

//Reads in user input (Explained in more detail later)
scanf("%s", &character);

//Switch-Case
switch (character)
{
	case 'N':
	printf("NO\n");
	case 'Y':
	printf("YES\n");
	default:
	printf("Do not understand!\n");
}
```

If ‘N’ is the user input the Output will be:

```
>NO

>YES

>Do not understand!
```

This is because when one case statement is triggered it will continue down until a break statement is found to stop running the case body code.

Iteration

Iteration means looping, and looping quickly gives programs the ability to perform lots of similar operations very quickly, there're two types of iteration: 'for loops' and 'while loops/do-while loop'

For Loop

The for loop is given an end value and loops up until that value, while keeping track of what loop number it's currently on, here is an example below:

```
for(int x = 0; x > 10; x++)
{
    printf("Looped!");
}
```

```
for(int x = 0; x > 10; x++)
```

Each part of the for-loop has a role

Red

This is the *Declaration* section to define the loop counter variable, this defines the start point of the counter

Green

This section is called the *Conditional* and it contains a conditional statement that is checked at the end of the loop to see if the if-statement should continue looping, so in this case the loop should continue looping if **x > 10,** if this condition becomes false the loop will not continue.

Blue

The blue statement is the *Increment* section where the loop counter is incremented (increased in value), the **x++** is shorthand for **x = x + 1** . This can also be **x--**, if there was a case requiring the counter to decrease.

- Declaration section defines the loop counter
- Conditional section continues the loop if true
- Increment section is where the loop counter is incremented

Conditional Loops

Conditional loops work like the for loop but don't have a loop counter and will only loop while a condition is True. This means you can create an infinite loop, like so:

```
while (TrueCondition)
{
    printf("This will not stop looping");
}
```

Note:

An infinite loop is normally constructed using a naked for-loop:

```
for(;;)
{
    printf("This will also never stop looping");
}
```

This loop will never end and your program will get stuck within the loop.

The example below shows if the condition is false the program will never reach the code within the brackets

```
while (FalseCondition)
{
    printf("This code will never run");
}
```

To use this loop effectively you can use it with a conditional statement (like the if statement) or you can use it with a bool variable, examples are below

```
int count = 0;
while (count > 10)
{
    printf("loop");

    //Remember this, it's the same as "count = count + 1;"
    count++;
}
```

As it says above you can also use a while loop directly with a Boolean variable:

```
int count = 0;
_Bool keepLooping = 1; //Bool is true (1) is true
while (keepLooping)
```

```
{
    printf("loop\n");

    //Remember this, it's the same as "count = count + 1;"
    count++;

    if(count == 3)
    {
        keepLooping = 0; //keepLooping is now false, and the loop will stop
    }
}
```

This is very similar to how a for loop operates, but it is important to understand the different uses for a while loop.

Note:

The section in the brackets of the while loop is checking if that condition is true, you can also write it like so:

```
while(!stopLooping) {}
```

This is effectively saying, keep looping while stopLooping is "not true", the "!" is symbol means "not".

Do-While

A Do-While loop is almost exactly the same as While loop but with one small difference, it checks if the condition is true after executing the code in the body of the loop, a while loop checks the if the condition is true before executing the body. The code snippet below shows this difference:

```
_Bool falseCondition = 0;


while(falseCondition)
{
   printf(“While Loop\n”);
}


do
{
   printf(“Do Loop\n”);
} while(falseCondition)
```

Output:

```
> Do Loop
```

Because even though the Boolean is false, the Do loop executed a single time because the check was at the end of the body.

Using a Do-Loop

A real-world example of a do-loop cool be checking for user input, it prints out and asks for input if all is okay there is no need for looping if not it will loop. An example looking for the user to enter a ‘z’ is below

```
#include <stdio.h>
int main()
{
	//Will loop if this is false
	_Bool correct = 1;
	do
	{
	printf("Please enter the letter 'z': ");

	//Takes in user input
	char z;
	scanf("%s", &z);

	//Checks if answer is correct
	if (z != 'z')
	{
	//Incorrect
	correct = 0;
	printf("Incorrect!\n");
	}
	else
	{
	//Correct
	correct = 1;
	}
	} while (!correct);

	//If the user has completed the task
	printf("Correct!");
}
```

Loop control keywords

Sometimes there are situations where you want to prematurely stop the entire loop or a single iteration (loop), this is where loop control keywords come into use.

You have either a **break** or **continue** keyword:

break – Will stop the entire loop, this can be useful if an answer has been found and the rest of the planned iterations would be pointless.

```
for (int x = 0; x < 5; x++)
{
        if (x == 3)
        {
        break;
        }

        //The technicalities of this statement will be explained later
        printf("Loop value: %d", x);
}
```

Output:

> Loop value: 0

> Loop value: 1

> Loop value: 2

Now if the code is changed to not include the break:

```
for (int x = 0; x < 5; x++)
{
        //The technicals of this statement will be explained later
        printf("Loop value: %d", x);

}
```

Output:

> Loop value: 0

> Loop value: 1

> Loop value: 2

> Loop value: 3

> Loop value: 4

Continue – If we use the code from above but replace it for a **continue** the code will look like so:

```
for (int x = 0; x < 5; x++)
{
        if (x == 3)
        {
        continue;
        }

        //The technicals of this statement will be explained later
        printf("Loop value: %d", x);
}
```

The output it like so:

Output:

> Loop value: 0

> Loop value: 1

> Loop value: 2

> Loop value: 4

This shows that when ***x = 3*** the **continue** is executed and the loop is skipped and so is the **printf** statement is also skipped so there is no “Loop value: 3”

Nested loops

You can also place loops within loops to perform specific roles, in the example we are using for-loops but this can also be done with the while/do loop.

The example is printing out a 2D grid, the nested for-loop gives another dimension:

```
//Prints a 5x5 grid
for (int y = 0; y < 5; y++)
{
        for (int x = 0; x < 5; x++)
        {
        //Prints an element of the row
        printf("X ");
        }

        //Moves down a row
        printf("\n");
}
```

Output

```
> X X X X X
> X X X X X
> X X X X X
> X X X X X
> X X X X X
```

Functions

Functions are the building blocks of a program, they allow the reuse of code, the ability to keep it readable and stops the programmer repeating code. Repeating code is heavily advised against because bugs will be repeated multiple times and changes to code also need repeating. Functions give a centralised controlled area that deals with the distinct roles of the program.

A method has two elements, **parameters**:(the items passed into the function) and the **return type** (the variable being returned) these are both optional and you can have a function with neither.

A function must be defined above its call, like so;

```
//Function definition
void Print_Smile()
{
	printf(":)\n");
}

int main()
{
	//Method call
	Print_Smile();

	return 0;
}
```

And this way round would be incorrect:

```
int main()
{
	//Method call [ERROR HERE]
	Print_Smile();

	return 0;
}

//Function definition
void Print_Smile()
{
	printf(":)\n");
}
```

Function Parameters

Sometimes it might be useful to pass data into a program, there are two types of parameter passing, by **reference** and by **value.** Passing by reference is what it sounds like, it passes a direct reference to a variable not a copy, so any changes to that passed variable effect it back in the calling function. Passing by value is the passing of a copy of that variable, so any changes to that variable do not effect that passed variable.

Passing by reference is not directly supported by C, but the effect is possible when dealing with pointers (We will talk about this in the advanced section)

- Passing by reference means changes to the parameter effects the passed variable
- Passing by value means changes to parameter does not effect the passed variable

For the time being passing by value is done below;

```
void Add(int num1, int num2)
{
	//Adds values together
	int newValue = num1 + num2;

	//Prints result
	printf("The result is: %d", newValue);
}

int main()
{
	//Method call
	Add(10, 4);

	return 0;
}
```

Output:

```
> The result is: 14
```

In this case the function has two integer parameters that it takes both in and prints the result. Don't look to deep into the **printf** statement, we'll go into why "%d" is used and how to use **printf** later.

Below is another example but this time a string is used a parameter:

```
void PrintStr(char printData[])
{
	printf(printData);
	printf("\n");
}
```

“char printData[]” is the parameter variable and allows data to be used in that function, in this case it’s printing out the char string. The method call looks like this:

```
PrintStr("Flying Squirrel");
```

This is an incredibly useful feature that allows us to make general purpose code and change its function output by what is put in as a parameter.

Returning values

Returning allows us to return data from a method, this lets us do computation within a function and get the function to automatically return the result. Let's take the one of the previous examples and adapt it so it returns the result instead of printing it:

```
int Add(int num1, int num2)
{
	//Adds values together
	int newValue = num1 + num2;

	//Returned keyword
	return newValue;
}

int main()
{
	//Method call
	int storeResult = Add(10, 4);
	return 0;
}
```

The areas that changed have been highlighted. When a value is to be returned, the "return" keyword is used, after this line has run it returns to the **line where the method was called** so any code under the return **will not run.**

The returned result is then stored in 'storeResult' to be used later. Returning can happen with any variable type. I'll show you an example below that checks if the number is an even value (Using the modulus operator talked about above that finds the remainder of a division):

```
 _Bool EvenNumber(int value)
{
	if (value % 2 == 0)
	{
```

```
        //Return true
        return 1;
        }
        return 0;
}

int main()
{
        if (EvenNumber(2))
        {
        printf("Even number!");
        }

        if (EvenNumber(5))
        {
        printf("Even number!");
        }

        return 0;
}
```

Output:

```
>Even number!
```

This program uses the return variable form the method EvenNumber() as a conditional for the if-statement and if it's true it will print "Even number!". As you can see from the output the first one prints but the second does not.

Recursion

Recursion is a difficult concept and will only be lightly touched on here and its real-world uses and functionality explained in the advanced section.

Recursion is a definition of a functions commands involving a reference to itself, yes very confusing I know, but I use some examples to explain.

```
const int maxLoops = 5;
void Sequence(int previous, int now, size_t loopCount)
{
	//Works out next value
	int next = previous + now;
	//Prints new value
	printf("New value: %d\n", next);

	//Increments counts
	loopCount++;

	//Stopping condition to make sure infinite looping doesn't occur
	if (loopCount < maxLoops)
	{
	//Recursive call
	Sequence(now, next, loopCount);
	}
}

int main()
{
	Sequence(1, 1, 0);
}
```

Output:

```
>New value: 2
```

```
>New value: 3
>New value: 5
>New value: 8
>New value: 13
```

There’re a few things to note, the lack of iteration loops, recursion in its essence causes looping. The second thing to not if the if statement labelled ‘stopping condition’, if recursive set-ups don’t have conditions that stop them looping they will loop forever, so this is a crucial element for using recursion effectively. If you don’t fully understand yet, don’t worry we will go into more detail later in the advanced section of the tutorial.

Arrays

Arrays have been mentioned previously, it is a data structure that holds a set number of variables next to each other in memory. The array will be given a *type*, for example 'int'. Arrays are used to quickly define lots of variables and keep relevant variables together. An array is defined below:

A static size, with the size in the square brackets:

```
int lotsOfNumber[20];
```

Or you can define values at the definition, **Note**: a size does not need to be defined because it's automatically determined by the number of values you specify:

```
int lotsOfNumbers[] = {1,3,4};
```

- **Arrays start at 0**, so the first index has an identifying value of 0, the second is 1 and so on. This means when accessing values, you need to remember it is always one less than the number of values it contains

You can access an index like so:

```
int var = lotsOfNumbers[0];
```

This will grab the first index of the array and place it in 'var'.

Arrays are very useful to access the tightly related data very quickly, you can use a for-loop to loop through the indexes and use them according. An example is below:

```
int lotsOfNumbers[] = { 1, 3, 4, 10};
for (int x = 0; x < 4; x++)
{
        printf("%d\n", lotsOfNumbers[x]);
```

```
}
```

Output:

```
> 1
> 3
> 4
> 10
```

In C you cannot get the length directly and need to work it out, this can be done simply using the **sizeof()** keyword, this returns the size of the elements in the brackets, so for example on a 64bit machine a **int** should be represented using 4bytes so **sizeof** will return 4. The length of an array can be worked out as so:

```
int arrayLength = sizeof(lotsOfNumbers) / sizeof(lotsOfNumbers[0]);
```

This takes the entire size of the array, and divides it by the first element and the division gives how many indices the array holds.

Multi-dimensional arrays

You can also define a second dimension in the array or even a third, this gives more flexibility when working with arrays. For example, a 2D array could be used to store coordinates or positions of a grid. A 2D array is defined as so:

```
int arrayOfNumbers[][2] = { {1,1}, {1,1}};
```

You access an element by putting the values in the square brackets for the x and y coordinate

```
int element = arrayOfNumbers[X][Y]
```

The obvious difference is that the second dimension **needs** a value, this cannot be automatically resolved when creating an array and that the initialization requires nested curly brackets. Below is the 3D example:

```
int arrayOfNumbers[][][2] = {{{1,1},{1,1}},{{1,1},{1,1}}};
```

But at this stage it is starting to lose readability and it's much better practise to lay it out like so:

```
int arrayOfNumbers[][][2] =
{
        {{1,1},{1,1}},
        {{1,1},{1,1}}
};
```

Passing Arrays in functions

As we saw when we learned about functions above, passing variables in as parameters can be very useful and so can passing in lots of variables stored as an array. However, we learnt above how to determine the size of an array above but this **does not work** with an array passed as a parameter, so we must pass in a variable that represents the number of elements that array is holding. There is a special variable type used to hold count variables, it's called **size_t** and it is an unsigned integer value (Meaning it cannot become negative) used to hold values for a count.

An example of an array being passed as a parameter is below:

```
void Print_SingleDimenArray(size_t length, int ageArray[])
{
	//Length is used to dynamically determine the for loop length
	for (int i = 0; i < length; i++)
	{
	printf("%d\n",ageArray[i]);
	}
}


int main()
{
	//An array of ages
	int ages[] = { 32, 11, 12, 1, 8, 5, 10 };

	//Size is determined as shown previously
	size_t ageLen = sizeof(ages) / sizeof(ages[0]);

	//Method call
	Print_SingleDimenArray(ageLen, ages);
}
```

Output

>32

>11

>12

>1

>8

>5

>10

The array has been passed and used to print values. The length of the array is crucial to the program as it allows the for loop to work for arrays of varying length.

- Any changes to an array in any method will change the array variable in the calling method. This is what was mentioned earlier as **passing by reference,** the whole array isn't copied and passed by reference with a little trickery occurring.

Extra – Advanced

This isn't crucial to understanding but I'm going to explain why determining the size of an array after it's passed as a parameter will not give the correct result.

When an array is passed as a parameter the first element's memory position address is passed (This is a pointer, promise we will touch on these later) and when you use **sizeof** to grab the size of the 'array' you are just getting the size of the first element returned.

User input

Sometimes it is necessary to get an input from the user, in the form of a choice or name etc. There are a few ways of doing this but the most straight forward method is the **scanf()** and **printf()** functions.

Before we go into how they work we first need to learn the basic type identifiers for **scanf()** and **printf(): (This is not a comprehensive list)**

Formatting ID	**Valid Input**	**Type needed for argument**
%c	Single characters, reads the characters and so on so on	char
%f	Float values,	float
%d	**Decimal integer**	**int**
%o	Octal	int
%u	Unsigned decimal integer (Does not have a sign and is just a positive int)	int
%x	Hexadecimal	int
%s	**String of characters and will continue reading until a whitespace is found**	**char[]**

scanf() works by grabbing user input and placing in a variable:

```
char name[20];
scanf("%s", name);
```

And an example of grabbing an integer value:

```
int userInputInt;
scanf("%d", &userInputInt);
```

Note: The use of the “&” for userInputInt. This is saying that it’s an exact reference to the memory location of userInputInt and the value read in from the user is placed into that integer variable by adding it to that memory location.

printf is used to display information to the console for debugging and user interfaces. **printf** also uses the same formatting ID’s as **scanf** and these formatting ID’s are used as placeholders, the example below shows printing a string and integer respectively:

Integer

```
int number = 10;
printf("%d", number);
```

String

```
char word[] = "Lemon";
printf("%s", word);
```

These formatting ID work as place holders and **printf** can take multiple parameters depending on the number of formatting ID present in the print string. For example:

```
int number = 10;
char word[] = "Lemon";
printf("A fruit is: %s and here is a number: %d", word, number);
```

Each formatting ID is replaced in order by the parameters, in this case “%s” is replaced by *word* and “%d” is replaced by *number*. This can work with as

many number of formatting ID’s as is needed.

fputs()

fputs() is the much simpler alternative when you just want to **print a string of characters** to the console. **fputs()** does not use the formatting IDs that **printf()** uses so it doesn’t need to check for formatting and is slightly quicker. It also automatically puts a ‘**\n**’ onto the end of the printed line.

Quiz chapter 2

This section is designed to keep you on your toes about the previous content, there will be 10 questions that you can answer to test your knowledge, below in the neighbouring section will be the answers.

1. Name one data type that is used to hold decimal numbers?
2. What two values can a boolean hold?
3. What is a condition statement?
4. Name the three sections that make up the for loop
5. What does the **continue** keyword achieve?
6. Can a valid function not return a value, and if so what datatype is used?
7. How many parameters can a function have?
8. What is the difference between **printf** and **puts**?
9. Which type is a valid input for this formatting id “%d”?
10. What code would you use to find how many values an array can hold?

Chapter 3
Advanced Basics

Pointers

Pointers have been mentioned previously and they are the memory address of a variable, this is much like a house address for a person. These can be passed around a parameter and allow the effects of **passing by reference** mentioned previously.

The program below will show the address of a variable:

```
#include <stdio.h>

int main()
{
	int var;

	printf("The address is: %x\n", &var);

	return 0;
}
```

Output

> The address is: 10ffa2c

Note: This will be different almost every time you run

Note the highlighted statement, the **'&'** (Reference Operator) is used to return the memory location of the variable and allows certain statements to access the data in that location. The '%x' is used because a memory location is in hexadecimal.

To create a pointer, we use the dereferencing operator (*), this will create a pointer variable that is designed to hold a memory locations address. Below

is a program that creates a pointer and uses reference operator (&) to store another regular variable's address in the newly created pointer. The dereferencing operator (*) is also used to access or change the actual data in that memory location.

- Dereferencing operator (*) is used to create a pointer, it is also used when changing the actual value
- Referencing operator (&) is used when obtaining the memory location of a pointer

```
//Regular variable
int var = 10;

//Pointer
int* pointer;

//Storing of var memory location
pointer = &var;

//Pointer is now effecivly 'var' so
//things like this can happen
*pointer = 20;

printf("Var's value is now: %d", var);
```

Output

```
>Var's value is now: 20
```

NULL pointers

When you create a pointer is it initially not given anything to point at, this is dangerous because the pointer when created it references random memory and changing this data in the memory location can crash the program. To prevent this, when we create a pointer we assign it to NULL like so:

```
int* pointer = NULL;
```

This mean the pointer has an address of ‘0’, this is a reserved memory location to identify a null pointer. A null pointer can be checked by an if statement:

```
if (pointer){}
```

This will succeed if the pointer **isn’t** null.

Using pointers

Now some real-world uses of pointers are passing them as parameters and effectively passing them **by reference**. The example below will show the effect:

```
void Change_Value(int* reference)
{
	//Changes the value in the memory location
	*reference = 20;
}


int main()
{
	//Creates pointer to variable
	int var = 10;
	int* pointer = &var;

	printf("The value before call: %d\n", var);

	//Method call
	Change_Value(pointer);
	//Prints new value
	printf("The value after call: %d\n", var);

	return 0;
}
```

Output

```
>The value before call: 10
>The value after call: 20
```

This passes the memory location not the value of the variable meaning you have the location where you can make changes.
Note the parameter is **int*** this is the pointer type, so for an example of a pointer to a char would be **char*.**

Pointer Arithmetic

There is times when moving a pointer along to a another memory location might be useful, this is where pointer arithmetic comes into use. If we were to execute say **ptr++** and the **ptr** was an integer pointer it would now move 4bytes (Size of an int) along, and we were to run it again, another 4bytes etc. This can mean pointer (if pointing to valid array structures) can act much like an array can. An example is below:

```
 int arrayInt[] = { 10, 20, 30 };
size_t arrayInt_Size = 3;

//Will point to the first array index
int* ptr = &arrayInt;

for (int i = 0; i < arrayInt_Size; i++)
{
        //Remember, *ptr gets the value in the memory location
        printf("Value of arrayInt[%d] = %d\n", i, *ptr);
        ptr++;
}
```

Output

```
>Value of arrayInt[0] = 10
>Value of arrayInt[1] = 20
>Value of arrayInt[2] = 30
```

This shows that a pointer to the first address of the array can be incremented along the addresses of the array (Remember each value of an array is stored in neighbouring memory locations)

You can do the opposite and decrement a pointer i.e. make the pointers value decrease.

There is also way to compare pointers using relational operators such as ==, < and >. The most common use for this is checking if two pointer point to the same location:

```
int value;

//Assigns ptr1 and prt2 the same value
int* ptr1 = &value;
int* ptr2 = &value;

//ptr3 is assigned another value
int* ptr3 = NULL;

if (ptr1 == ptr2)
{
	printf("ptr1 and ptr2 are equal!\n");
}

if (!ptr1 == ptr3)
{
	printf("ptr1 is not equal to ptr3\n");
}
```

This checks if the various pointers are equal. The same can be done with > and <.

Function pointers

Much like you can do with variables you can also do the same with functions, below is a snippet of code that shows a function pointer being defined. Key sections will be highlighted:

```
void printAddition(int value1, int value2)
{
	int result = value1 + value2;

	printf("The result is: %d", result);
}

int main()
{
	//Function pointer definition
	//<retrunType>(*<Name>)(<Parameters>)

	void(*functionPtr)(int, int);
	functionPtr = &printAddition;

	//Invoking call to pointer function
	(*functionPtr)(100, 200);

	return 0;
}
```

The basic structure for defining a function pointer is like so

```
<Return_Type> (*<Name>) (<Parameters>)
```

Where in this case:
<Return_Type> = void
<Name> = functionPtr
<Parameters> = int, int

This function pointer can now be passed as a parameter and used in situations

where you would want to change the behaviour of code but with almost the same code.

An example could be dynamically choosing what operation a calculator should perform (Note: this is complex code and should be used a rough example, so don't worry if you don't fully understand)

```
void calculator(int value1, int value2, int(*opp)(int,int))
{
	int result = (*opp)(value1, value2);
	printf("The result from the operation: %d\n", result);
}

//Adds two values
int add(int num1, int num2)
{
	return num1 + num2;
}

//Subtracts two values
int sub(int num1, int num2)
{
	return num1 - num2;
}

int main()
{
	calculator(10, 20, &add);
	calculator(10, 20, &sub);

	return 0;
}
```

Here what is happening we are passing the function 'add' and 'sub' as parameters for the function calculator, as you see from the highlight the function parameter is defined like it is above with the return type, name and parameters being defined, all that is passed into calculator is &add and &sub

for the function pointers. The calculator function then goes on to invoke the pointer and passes in the values and returns the result.

Storage Classifications

In C each variable can be given a storage class that can define certain characteristics.

The classes are

- **Automatic variables**
- **Static variables**
- **Register variables**
- **External variables**

Automatic variables

Every variable we have defined so far has been an automatic variable, they are created when a function is called and automatically destroyed when a function exits. These variables are also known as *local variables*.

```
auto int value;
```

Is the same as

```
int value;
```

Static variables

Static is used when you want to keep the variable from being destroyed when it goes out of scope, this variable will persist until the program is complete. The static variable is created **only once** throughout the lifetime of the program. Below is an example of a static variable in use:

```
void tick()
{
	//This will run once
	static int count = 0;

	count++;
	printf("The count is now: %d\n", count);
}
```

```
int main()
{
	tick();
	tick();
	tick();
}
```

Output

```
>The count is now: 1
>The count is now: 2
>The count is now: 3
```

The area highlighted section is the static definition and will only run once.

Register variables

Register is used to define a variable that is to be store in register memory opposed to regular memory, the benefit register memory has is it is much much quicker to access however there is only space for a few variables.

Defining a register variable is done like so:

```
register int value;
```

External variable

We touched on this before but a global variable is a variable not defined in a scope and therefore can be used anywhere. An external variable Is a variable defined in a separate location like another file and the **extern** keyword is used to signify that the variable is in another file. You would include another file as reference by placing this at the top of your file:

```
#include "FileName.c"
```

This is to tell the program to reference this file as well. Note the files need to

be in the same location.

Program 1 [File_2.c]

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

int main()
{
	extern int globalValue;
	printf("The global variable is: %d", globalValue);
}
```

Program 2 (It is small) [C_TUT.c]

```
#include <stdio.h>

int globalValue = 1032;
```

The **Output** of running File_2.c:

>The global variable is: 1032

This shows that the **globalValue** is referenced from C_TUT.c and used in another file by using the extern keyword.

File I/O

There will be cases where you'll need to retain data past the duration of the programs life time, this is where saving a loading to file comes into use. There are two types of main files used

- Text Files
- Binary files

To start you will need to create a pointer of type 'FILE' tile will allow communication between file and program. This is done like so:

```
FILE* ptr;
```

Opening a file

If this file already exists you can open it and read it's contents, this would be done by using:

```
ptr = fopen(char* filename, char* mode)
```

Where:

filename = to the filename of the file to open

mode = this is the mode to open the file in, the comprehensive list is below

Mode	Description
r	Opens the file for reading
w	Opens the file for writing, if the file does not exist a new file is created. The program will begin writing content from the start of the file.
a	Appending mode, if the file does

	not exist it is created. Any changes to an existing file will be added onto the end.
r+	Opens the file for reading and writing
w+	Opens the text file for both reading and writing. It first truncates (chops) the file length to zero if it exists, if it doesn't create a new file
a+	Once again opens the file for reading a writing. If the file does not exist a new one is created. Reading starts at the very start but writing is only appended onto the end of the file.

Closing a file

This one is nice and simple, just need to run:

```
fclose(ptr);
```

Writing to a file

There are two different ways of writing to a file **fprintf()** and **fputs()** (Like the console printing commands **printf()** and **puts()**. The only difference really is that **fprintf()** allows you to use the formatting ID like '%s' and **fputs()** does not. This mean **fputs()** does not need to check for formatting and just prints out the exact string making it faster. **fputs()** also automatically adds a newline character (\n) to the character string it is given, much like **puts()**. The example below shows to open, write-to and close a file:

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

```
int main()
{
	//Creates the file pointer
	FILE* ptr;
	//Opens the file
	ptr = fopen("C:/C_IO/example.txt", "w+");

	//Writes to the file
	fprintf(ptr, "fprint()\n");
	fputs("fputs()\n", ptr);

	//Closes file
	fclose(ptr);
}
```

It creates a file in the ‘C_IO’ directory directly on the C: drive (make this before running the program). It uses the ‘w+’ mode so every time the program is run the data in file is overridden.

You can also save values only using **fprintf** like so:

```
fprintf(ptr, "%d\n", value);
```

- Opening a file fopen()
- Writing to a file
    - fprintf()
    - fputs()
- Closing a file fclose()

Exercise

- Write a program to save the output of an add function save it to the C_IO on the C (Or your main drive) you made earlier and call it “addSave.txt”, use 10 and 25 and your values.

Solution

Something like this, doesn’t have to be exact there are always different ways of doing things.

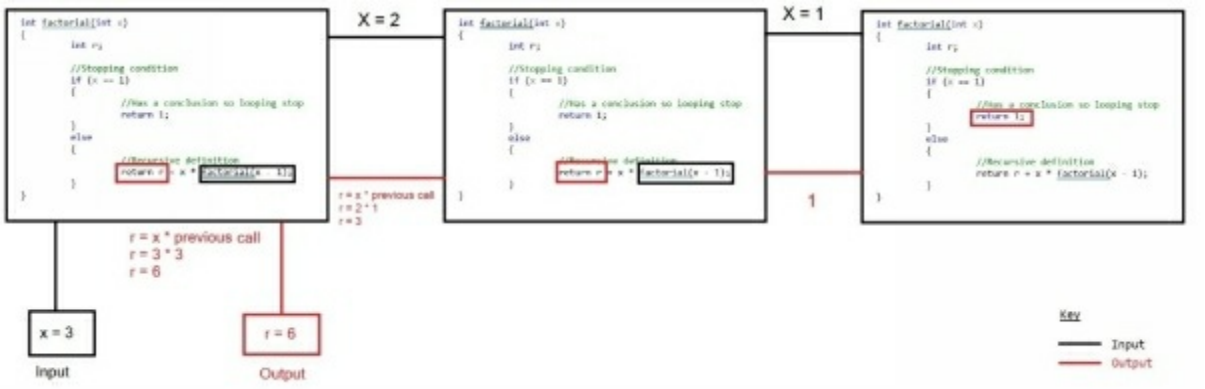
```
#include <stdio.h>

int add(int num1, int num2)
{
	return num1 + num2;
}

int main()
{
	//Grabs the value to save
	int saveValue = add(10, 25);

	//Creates the file pointer
	FILE* ptr;
	//Opens the file
	ptr = fopen("C:/C_IO/addSave.txt", "w+");

	//Writes to the file – This is how printf works but without the ptrn
	fprintf(ptr, "%d\n", saveValue);

	//Closes file
	fclose(ptr);
}
```

Recursion Continued

We touched on recursion in the basic section of the tutorial, and again it's the definition of a functions tasks with definition to itself. As promised there are a few more examples of recursion explained below:

```
int factorial(int x)
{
	int r;

	//Stopping condition
	if (x == 1)
	{
	//Has a conclusion so looping stop
	return 1;
	}
	else
	{
	//Recursive definition
	return r = x * factorial(x - 1);
	}
}

int main()
{
	puts("Please enter a number: ");

	//Reads in user input
	int a, b;
	scanf("%d", &a);

	//Starts the execution
	b = factorial(a);
	printf("The factorial is: %d", b);
}
```

This is the world famous example of recursion that is used to find a factorial of a number (3 factorial is 3x2x1). It works by the recursive return statement above, it stops by having a return statement without a recursive definition, i.e. when x = 1 the function just returns 1, this means the stack can unwind and find an answer. There is a flow diagram below:



Exercises chapter 3

1. What does a pointer variable hold?
2. What operator is used to signify a pointer?
3. What will "++" do to a pointer of type int where an integer is 4bytes?
4. How many times is a static variable initialised throughout the life of a program?
5. What happens when you define a variable as 'register'?
6. What are the two main functions used to write to a file?
7. What does the fopen() mode **w+** do?
8. In what case will the code below succeed. Pointer is an integer pointer

```
if(pointer)
{


}
```

9. What is '&' called and what does it do?
10. When you pass an array as a parameter what do you also need to path with it?

Chapter 4
Custom Structures

The customizable aspect of programming languages allows them to perform any role under the sun and the programmer can manipulate and craft structures designed to store and process data.

Structures

The first user-defined object we'll come across is the 'structure', this allows custom storing of data and are defined like so:

```
struct Structure-Name
{
        //Statements
};
```

Where "statements" is the variables the structure is to hold. The structure allows custom storing of data in a meaningful way, it much like the array, the only difference being that an array stores lots of variables of the **same type** the structure allows storing of several types, even other structures.

A real example is below:

```
struct Person
{
        int age;
        char firstName[15];
        char lastName[15];
        char favoriteColour [10];
};
```

This structure is designed to hold data about a person, a new person can be created like so:

```
struct Person p1;
```

It is much like defining a new variable, the example below shows that two newly created people do contain separate and independent values from each other:

```c
#include <stdio.h>

struct Person
{
	int age;
	char firstName[15];
	char lastName[15];
	char favoriteColour[10];
};

void PrintPerson(struct Person p)
{
	printf("First Name:      %s\n", p.firstName);
	printf("Last Name:       %s\n", p.lastName);
	printf("Favourite colour: %s\n", p.favoriteColour);
	printf("Age:             %d\n", p.age);
	puts("");
};


int main()
{
	//Person 1
	struct Person p1;

	p1.age = 10;
	strcpy(p1.firstName,"John");
	strcpy(p1.lastName, "Doe");
	strcpy(p1.favoriteColour, "Red");

	PrintPerson(p1);

	//Person 2
	struct Person p2;

	p2.age = 25;
	strcpy(p2.firstName, "Lucy");
```

```
        strcpy(p2.lastName, "Brown");
        strcpy(p2.favoriteColour, "Yellow");

        PrintPerson(p2);
}
```

Output:

```
>First Name:        John
>Last Name:         Doe
>Favourite colour:  Red
>Age:               10


>First Name:        Lucy
>Last Name:         Brown
>Favourite colour:  Yellow
>Age:               25
```

To copy a string value into a structure variable you need to use **strcpy()**. This code shows that each time a new Person is created so does a whole new set of variables that go along with that Person. This give a very easy 'cookie-cutter' way of creating lots of meaningfully variables very quickly. Also note how each variable is accessed, it's by using a full stop (Access operator) :

```
personName.Variable
```

```
p1.age = 10;
```

Nested structures

Structures can also hold other structures, they can either be: internally defined or externally defined.

Internal definition

This is where you define a structures definition inside another's structures variable definition. If we use the person example and turn the first and last name into a structure.

```
struct Person
{
	int age;
	char favoriteColour[10];
	struct Name
	{
	char firstName[15];
	char lastName[15];
	} name;
};
```

Where the first name is access like so:

```
p1.name.firstName;
```

External definition

This is like the same but the definition is not within another a structure:

```
struct Name
{
	char firstName[15];
	char lastName[15];
};
```

```
struct Person
{
	int age;
	char favoriteColour[10];
	struct Name name;
};
```

Where first and last name are accessed exactly the same as the internal definition.

- Internal definition means you cannot recreate the structure in other locations but can use it internally
- External definition means you can use it internally and in other locations

This gives a much more modular and readable code.

TypeDef

Typedef is a keyword provided so you can customize the name of build in and user defined variables and structures. I show an example below:

```
typedef int INTEGER;
```

This is taking the data type **int** and giving it another persona as INTEGER, now an INTEGER is defined like normal:

```
INTEGER value = 10;
```

Real-World Example

A real world example of where this could be useful is backwards compatibility in situations where integer values are different. So you would define the int like so:

```
typedef int int32;
```

Use int32 like int normally and on the other machine where you would need a larger int value you would change the definition to be:

```
typedef long int32;
```

And the code would still work with a very small amount of maintaining.

Enums

Enums stands for Enumerated types. An enum is a user defined type that allows creations of customs data types that hold custom values. An example is below:

```
enum Condition
{
	Working,
	NotWorking,
	Finished,
	Unknown
};
```

This defines an enum called Condition where each of the possible types are defined as possible states. You create and use an enum like this:

```
#include <stdio.h>

enum Condition
{
	Working,
	NotWorking,
	Finished,
	Unknown
};

int main()
{
	//Created
	enum Condition programCondition;

	//Assigned a value
	programCondition = Working;
	//Comparison
```

```
        if (programCondition == Working)
        {
        puts("All is good!");
        }
}
```

In this program is shows how you can create an enum, how you can assign it a value and how you can compare it’s value.

Enums are most commonly used like boolean with extra context and features. It improves readability and adds extra states opposed to the binary nature of booleans values. Enums are very good at keeping track of data with limited well-defined values, like the current month or the day of the week.

Exercise

Create an enum that deals with the days of the week and prints out whatever day has been assigned (Hint: A switch-case is very useful for checking current day)

Solution

Something like this:

```
#include <stdio.h>

enum WeekDay
{
	Monday,
	Tuesday,
	Wednesday,
	Thursday,
	Friday
};
enum Condition currentDay;

int main()
{
	currentDay = Friday;

	switch (currentDay)
	{
	case Monday:
	puts("It's Monday!");
	case Tuesday:
	puts("It's Tuesday!");
	break;
	case Wednesday:
	puts("It's Wednesday!");
	break;
```

```
        case Thursday:
        puts("It's Thursday!");
        break;
        case Friday:
        puts("It's Friday!");
        break;
        }
}
```

Unions

Unions are a special data type that allow the programmer to store different data types in the **same memory location**. You can define a unions with many data members, **but only one** variable can hold a value at one time. These are an efficient way of using the same memory location for different uses.

The structure of a union is like so:

```
union Name
{
        //Variables
        int i;
        int y;
        char word[20];
        _Bool alive;
};
```

And multiple copies can be created much like how you can make versions of a structure or an enum. This is done like this:

```
union Name t1;
```

This **size** of the unions is as **large as the largest variable**, not as big as all the values together giving an efficient way to store variables one at a time opposed to a structure. Comparing the two the union above is *20 bytes* in size, a structure with exactly the same variables would be *32 bytes*. This would amount to a big space reduction if used on a much larger scale.

The variables in a union are accessed using the member access operator (.) and used like this:

```
union Name example;

example.i = 10;
printf("i before assigned another variable a value: %d\n", example.i);

example.y = 25;
printf("i after assigned another variable a value: %d\n", example.i);
printf("y value for reference: %d\n", example.y);

strcpy(example.word, "Hello!");
printf("i after assigned a string a value: %d\n", example.i);
printf("y after assigned a string a value: %d\n", example.y);

printf("String variable: %s", example.word);
```

Output

```
>i before assigned another variable a value: 10

>i after assigned another variable a value: 25

>y value for reference: 25

>i after assigned a string a value: 1819043144

>y after assigned a string a value: 1819043144

>String variable: Hello!
```

As you can see changing one value effects every other variable, this is the downfall of unions if not used correctly and need to managed very carefully so errors do not occur due to changing variables.

Variable argument lists

You might stumble upon a problem that might require a function with many parameters, this is where variable arguments comes it because it allows a dynamic amount of variables passed as a parameters.

Note:

A new included is required for this:

```
#include <stdarg.h>
```

Add this when using variable arguments.

You design the method like so:

```
void function(int noOfVariables, ...)
..
..
```

Where there is always one variable defining the number of variables followed by the variables. The extra variables are assigned to something called a *va_list,* these lists are manipulated with these functions:

```
va_start(va_list valist, int numberOfVaribles);
```

va_start effectively takes the extra variables and places them in the va_list

```
va_arg(valist, type);
```

va_arg takes the next variable and returns it, it doesn't know if the current integer is the last so the program setup needs to set up to make sure it doesn't overflow.

```
va_end(valist);
```

va_end simply cleans up the memory for the va_list.

Below is an example that finds the largest value:

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

int max(int n, ...)
{
	int largest = 0;
	//Creates an assigns
	va_list valist;
	va_start(valist, n);

	//loops through variable list
	for (int i = 0; i < n; i++)
	{
	//Grabs the next arg
	int nextVar = va_arg(valist, int);

	//Compares size of values || The first value is assigned to be the largest value
	if (nextVar > largest || i == 0)
	{
	largest = nextVar;
	}
	}

	//Frees up memory
	va_end(valist);

	return largest;
}
```

```
int main() {
        printf("Largest: %d\n", max(6, -2,3,4,5,66,10));
         printf("Largest: %d\n", max(3, 7, 2, 1));
}
```

Output

> Largest: 66

> Largest: 7

This program shows how the dynamic nature of the variable list can be very useful.

Exercise

Create a program using variable lists that returns the averages (product of all numbers/how many numbers) of **int** variables provided.

Create a method called ‘average’ that returns a *float*, this will be your variable function. And don’t forget to include:

```
#include <stdarg.h>
```

Solution

Something like this:

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

double average(int n, ...)
{
	double total = 0.0;

	//Creates variable list
	va_list vaList;
	va_start(vaList, n);

	//Grabs each variable
	for (int i = 0; i < n; i++)
	{
	//Same as: "total = total + va_arg(vaList, int);"
	total += va_arg(vaList, int);
	}

	//Clean up!
	va_end(vaList);

	double avg = total / n;
	return avg;
}
```

```
int main()
{
	//%f for a float
	printf("Average: %f\n", average(4, 1,2,77,4534));
}
```

This is very similar to the max value example above, it has the va_start, va_arg and va_end.

Exercises chapter 4

1. What function is needed to copy a string into a suitable variable?
2. What are the two ways a structure can be nested?
3. The keyword ‘typedef’, what is it used for?
4. What is Enum short for?
5. What is the drawback of a union structure?
6. What determines the size of a union?
7. What does a va_list hold?
8. What is used to show a function is to use an argument list?
9. What operator is used access members of a structure?
10. How does a structure and array differ?

Chapter 5
Advanced Features

Header files

A header file is a list of function, variable and macro definitions that can be included and used in different files. A header file is specified by the **.h** filename extensions. They are included in other files by using the **#include** (pre-processor directives) and the name of the file ("header.h"). There are custom header files that programmers can create and also built in header files that come with the compiler, much like "**stdio.h**" that we have seen in the previous section.

To create a header file all you need to do is create a file with the **.h** file extension, this can be done in something as simple as notepad, this header file can now be included with another file to link and use the features of the header file.

Below is an example:

header.h

```
int x = 10;

int Function()
{
	return x;
}
```

Can he included and used like so:

```
#include <stdio.h>
#include "header.h"

int main()
{
	printf("%d", Function());
}
```

Output

>10

This allows programmers to make modules that can be reused between projects.

A possible use of headers is below:

```
int add(int num1, int num2)
{
	return num1 + num2;
}
int sub(int num1, int num2)
{
	return num1 - num2;
}
int div(int num1, int num2)
{
	return num1 / num2;
}
int mul(int num1, int num2)
{
	return num1 * num2;
}
```

Once again, the calculator example returns, a header could be a list of functions like this that could then be included to allow the use of these functions.

Pre-Processor Directives

Pre-processing are used to give the compiler a type of command, you come across a command quite a few times already is the '#include' directive that tells the compiler to include a certain header file.

Below is a list of the processor directives, we will go through how they all work:

Directive	Description
#define	Used to define a macro
#include	Inserts a particular header from another file.
#undef	Removes the effect of #define
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends pre-processor conditional.

#define and #undef

These commands are used when global values are used to increase readability, for example:

```
#define LOOP_NUMBER 2;
```

Could be defined and used like so:

```
for (int i = 0; i < LOOP_NUMBER; i++)
{
	//Loop code
}
```

And the for loop will loop twice, but you cannot change the value like you would with a normal variable, it's **constant through the life of the program** unless **#undef** is used that will undefine the value set.

#undef is normally used to overwrite build in directives where you would undefine a value and then set a value of your choosing.

#ifdef and #if

#ifdef and **#ifndef** will return true and false respectively if a macro is defined, this most common use for this is checking if the *-DDEBUG* flag has been set and running code in DUBUG mode that has no use in the finished program. It's used like this:

```
#include <stdio.h>

int main()
{
#ifdef DEBUG
	puts("DEBUG!");
#endif
```

```
}
```

This code will only run if the code is flagged as debug mode. This can be very useful for a programmer.

#if, #else and #elif work exactly the same as if statements but are used to check the macro with arithmetic expressions rather that checking for its existence.

Error Handling

Error handling is a very important section, because a programmer needs to make sure his program is prepared to deal with expected and unexpected errors. C does not provide direct support for error handling but allows access to some low level functions. Most functions will return -1 or NULL if there is an error and set the **errno** error code, **errno** is a global variable that holds the last returned error code.

You will need to include the <errno.h> header file to use these error handling functions.

```
#include <errno.h>
```

There are a few functions that allow you to use an understand error codes.

perror()

Function displays the string you pass and attaches to the end the textual representation of the error code stored in **errno.**

strerror()

Returns the pointer to the textual representation of the **errno** value. This can be used to save the error feedback.

stderr file stream

stderr is used to output an error to the console.

Usage is below:

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

extern int errno;

int main() {

	FILE * file;
	int errnum;

	//Looks for file
	file = fopen("youWillNotFindMe.txt", "rb");

	//If the file returned an error
	if (file == NULL)
	{
	//Grabs error code
	errnum = errno;

	//Prints error code
	fprintf(stderr, "Value of errno: %d\n", errno);

	//Returns string provided + : and Error code message
	perror("Error printed by perror");

	//Grabs the error text
	char* errorMsg[] = { strerror() };
	printf("Error msg test print: %s", *errorMsg);
	}
	else
	{

	fclose(file);
	}
```

```
        return 0;
}
```

Above is usage of some of the error handling examples. Go over and understand it error handling is very important.

Below is another example of preventing the classic divide by zero error:

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

extern int errno;

int divide(int x, int y)
{
        if (y == 0)
        {
        //Prints error
        fprintf(stderr, "Diving by zero error..!");

        //Returns error code
        return -1;
        }
        else
        {
        //If valid returns even
        return x / y;
        }
}

int main()
{
        int returnCode = divide(0, 2);
}
```

Type casting

Variables has defined types, but there are situations where you'll need to convert from one type to another. Very commonly would be the conversion between *integer* to a *float* or *double*

The general format is like so:

```
(type)varToCast
```

This can be used like so:

```
int main()
{
	int integer = 3;
	float decimal = 1.5f;

	int result = (int)decimal + integer;
	printf("%d\n", result);
}
```

Output

```
> 4
```

What happens is any decimal values are truncated (removed) and the remainder is added to the integer value.

You can also upgrade a value, this example shows when a integer is upgraded to a float:

```
int main()
```

```
{
        int integer = 3;
        float decimal = 1.5;

        //Explicit casting
        float result = decimal + (float)integer;
        printf("%f\n", result);

         //Implicit casting
        float result = decimal + integer;
        printf("%f\n", result);
}
```

This can be done either implicitly or explicitly. Implicit is when the complier **automatically** converts the variable and explicitly is when you use the casting operator. It is however very good practice to specify a cast wherever it is necessary.

Memory management

C allows programmers to dynamically manage memory. This functionality is provided by:

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

Memory management requires the use of a few functions

- **void *calloc(int num, int size);**
  - This function allocates an array (size specified by **num**) and the size of each allocated specified by **size**
- **void free(void *address);**
  - This function releases memory, the location is specified by **address.**
- **void *malloc(int num);**
  - This function works like **calloc** but leaves the locations uninitialized.
- **void *realloc(void *address, int newsize)**
  - This function re-allocates memory to the size specified in **newsize**

Below is an example of using **malloc** or **calloc** to allocate memory for a string:

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

```c
#define WORD_SIZE 20

int main()
{
	char* word;

	//Allocates memory
	word = malloc(WORD_SIZE * sizeof(char));

	//or

	word = calloc(WORD_SIZE, sizeof(char));

	//Copies values over
	strcpy(word, "Hello nice to meet you!");

	//Printing
	printf("The string is: \"%s\"", word);

	//Deletes memory
	free(word);

}
```

This can be used to do quite complex things, like take in a user input and store it in an array exactly the size for that string:

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

#define WORD_SIZE 20

char* getInput()
{
	//Will be deleted
	char temp[50];

	//User input
	printf("Please enter your first name: ");
	scanf("%s", temp);
	puts("");

	//strlen() finds lenght of a string
	int len = strlen(temp);
	//Dynamic allocation of memory
	char* perfectSizeWord;

	//Allocates memory!
	perfectSizeWord = calloc(len, sizeof(char));

	//This command would not work, it takes the memory location of temp and just copies it so when
	//this function goes out of scope so does perfectSizeWord
	///perfectSizeWord = temp

	//Copies the values properly
	for (int i = 0; i < len; i++)
	{
	perfectSizeWord[i] = temp[i];
	}
```

```
	//Size comparason
	printf("Size of temp var:		%d\n", sizeof(temp));
	printf("Size of new perfect var:	%d\n", sizeof(perfectSizeWord));

	return perfectSizeWord;
}


int main()
{
	//Gets returned variables
	char* word = getInput();

	//Prints
	printf("The word is:		%s\n", word);
	//Delets memory
	free(word);
}
```

The program above is large, take your time and look through. Try and implement it and understand how it works.

Exercise

Use error handling techniques above to handle the error for allocating memory with **malloc** or **calloc**. To simulate a an error just make *word = NULL;* after the dynamic memory allocation.

Use this program frame below to help:

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

int main()
{
	char* word;

	//Allocates memory
	word = malloc(20 * sizeof(char));

	 //ERROR SIMULATE
	word = NULL;

	 //-----------------PUT ERROR CODE HERE----------------------

	//Deletes memory
	free(word);

}
```

Solution

Very simply all that needs including is:

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

int main()
{
	char* word;

	//Allocates memory
	word = malloc(20 * sizeof(char));

	//ERROR SIMULATE
	word = NULL;

	if (word == NULL)
	{
	fprintf(stderr, "Error: Unable to allocate the memory!");
	}
	//Deletes memory
	free(word);

}
```

Exercises chapter 5

1. How do you tell the compiler to use other files and libraries?
2. What file extension does a header file use?
3. What is #define used for?
4. What #include is required for error handling?
5. What is contained in the global variable **errno**?
6. What does strerror() return?
7. What file stream is required to print out an error?
8. Typecasting; what is it used for?
9. What is the different between **calloc** and **malloc**?
10. What normally will a function that encounters an error return?

Answers Chapter 2

1. Float or Double
2. True (1) and False (0)
3. A statement designed to check if a condition is true or false
4. Declaration, Conditional and Iteration.
5. Skips a full iteration
6. Yes it is, and by using the void keyword
7. Theoretically unlimited
8. Printf uses formatting id's, puts does not using formatting id's and puts automatically adds a '\n' add the end of the string
9. *Integer value*
10. *int arrayLength = sizeof(lotsOfNumbers) / sizeof(lotsOfNumbers[0]);*

Answers chapter 3

1. A memory location for a certain variable type.
2. The dereferencing operator (*).
3. It will move 4bytes along to the next memory location.
4. Only once.
5. The variable is stored in the much faster to access register memory.
6. fprintf() and fputs().
7. Opens the file for both reading and writing and chops the file size down to zero if it exists and creates it if it does not exist.
8. If the pointer is not NULL.
9. *Is called the reference operator and is used to access or return the raw memory address.*
10. *The size of the array preferably as a **size_t** variable.*

Answers chapter 4

1. strcpy() is needed.
2. Internally and externally defined.
3. Used to give variables and user defined structures custom names.
4. Enumerated type.
5. Only on variable in a unions can hold a value at one time.
6. A union is as big as its biggest variable.
7. The list of variables passed into a variable function.
8. An ellipsis at the end of the parameter list.
9. *The member operator denoted by a full stop (.)*
10. *An array stores variable of the same type, a structure stores variables of any type.*

Answers chapter 5

1. Using the #include directive
2. A header file uses “.h”
3. #define is used to create global definitions of constant values
4. #include <errno.h>
5. The last error code that was thrown will be stored there.
6. The textual representation of the error code stored in errno
7. The file stream is: “stderr”
8. Used to change one variable into another
9. *When **calloc** allocates memory it initialises values, **malloc** does not*
10. *NULL or -1*