

C Mini Lessons

last update: Mar 3, 2021

From <http://www.onlineprogramminglessons.com>

These C mini lessons will teach you all the C Programming statements you need to know, so you can write 90% of any C Program.

- Lesson 1** **Input, Output and Variables**
- Lesson 2** **Functions**
- Lesson 3** **Structures**
- Lesson 4** **Operators**
- Lesson 5** **Programming Statements**
- Lesson 6** **Arrays**
- Lesson 7** **Pointers and Allocating Memory**
- Lesson 8** **Passing Arrays and Structures to Functions**
- Lesson 9** **Function Pointers**
- Lesson 10** **File Access**
- Lesson 11** **Recursion**
- Lesson 12** **Projects**

Conventions used in these lessons:

bold - headings, keywords, code

italics - code syntax

underline - important words

Let's get started!

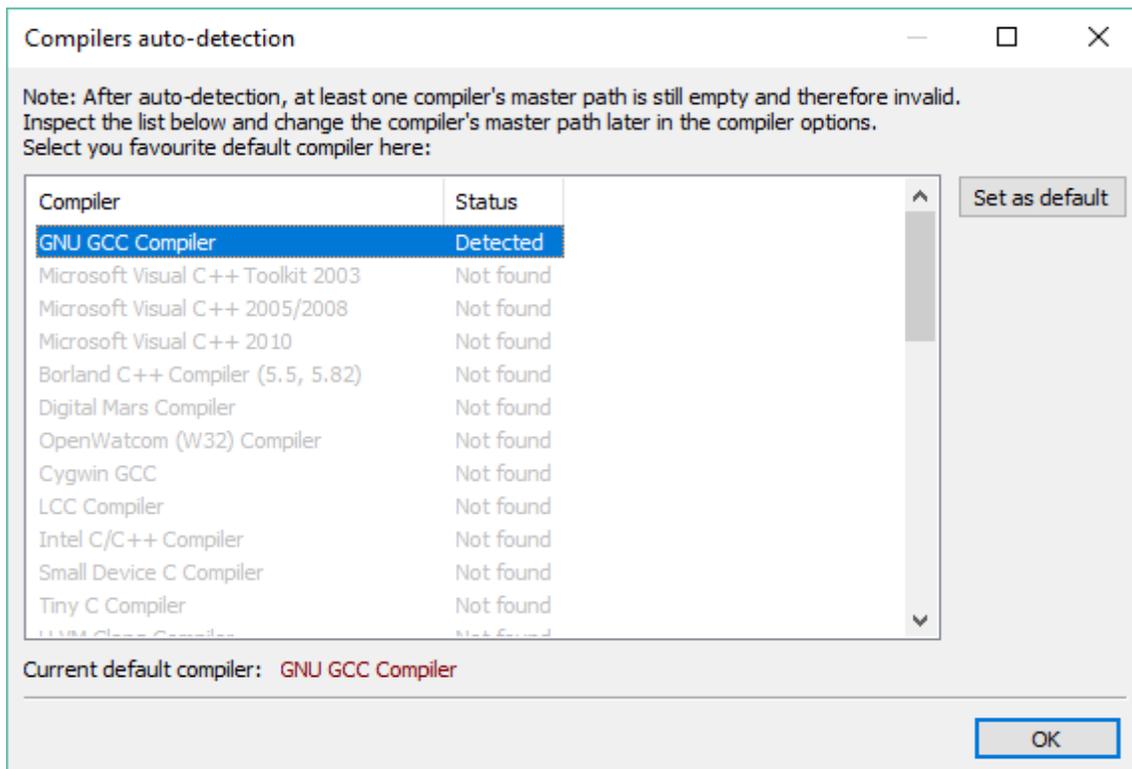
You first need to download CodeBlocks C Compiler. This is needed to edit, compile and run C programs. Alternately you can use your own C compiler or one of the online C compilers.

Download CodeBlocks from this link.

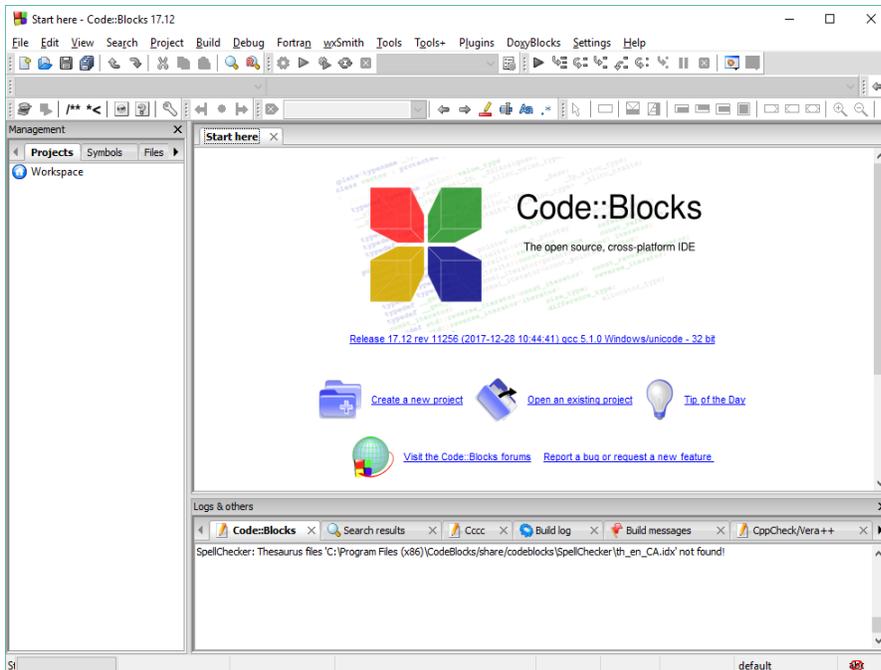
<http://www.codeblocks.org/downloads>

Download and install the **codeblocks-17.12mingw-setup.exe** file that includes the C Compiler

Once you installed and run CodeBlocks you will get this screen that asks you to select the C Compiler to use. We selected the GNU GCC Compiler and then pressed the "Set as Default" button.



The following screen then appears.

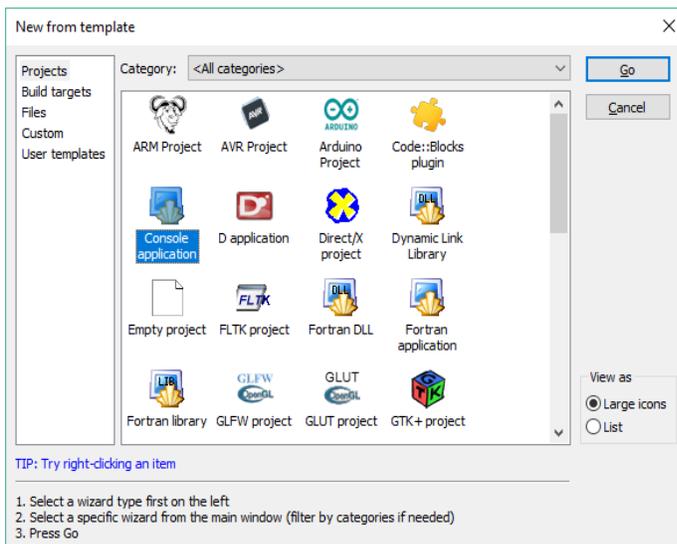


Lesson 1 Input, Output and Variables

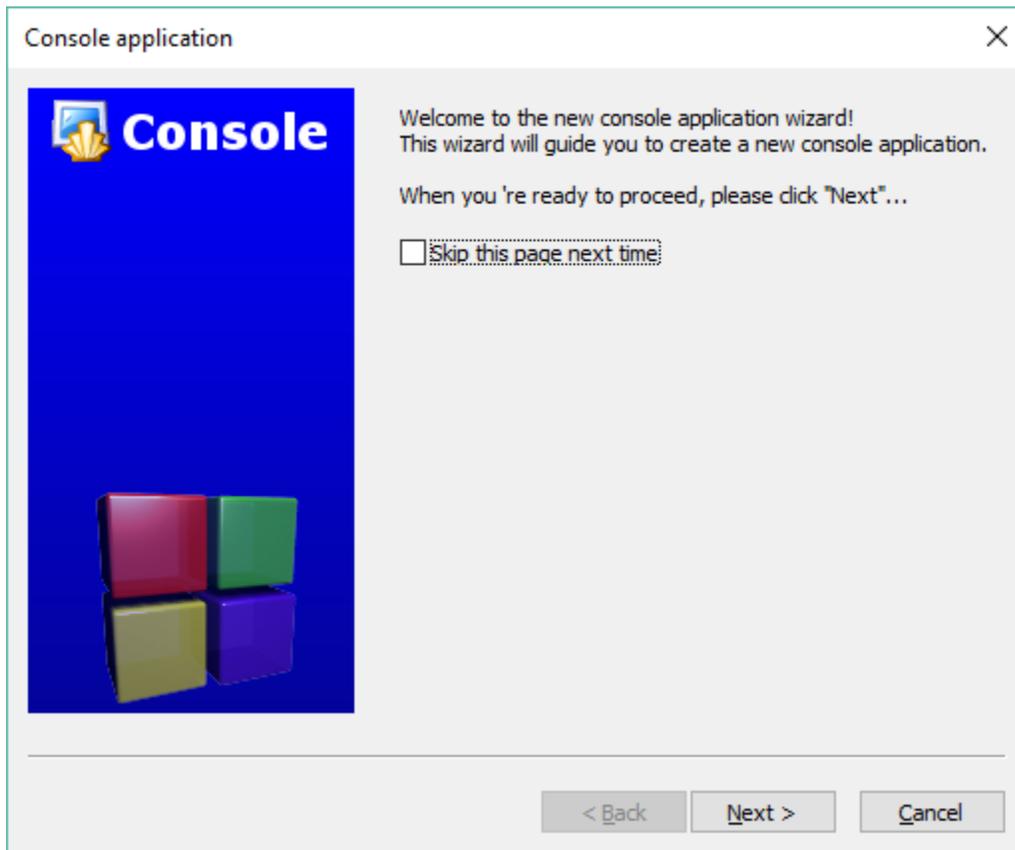
You first need to make a Project to store all your C lesson files.

From the start menu select **Create a new project** or from the file menu select, **Open New Project**.

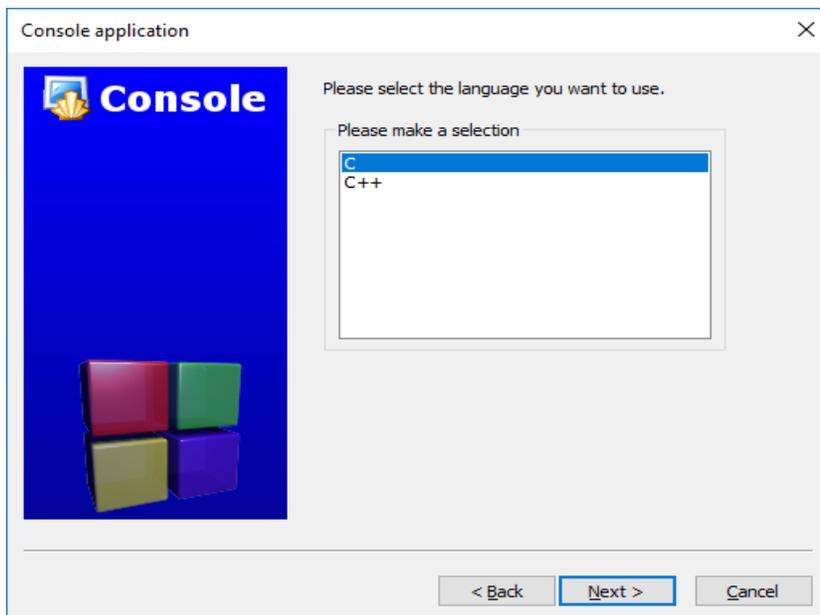
Next Select **Console application** and then press the “Go” button.



This Console screen appears next.

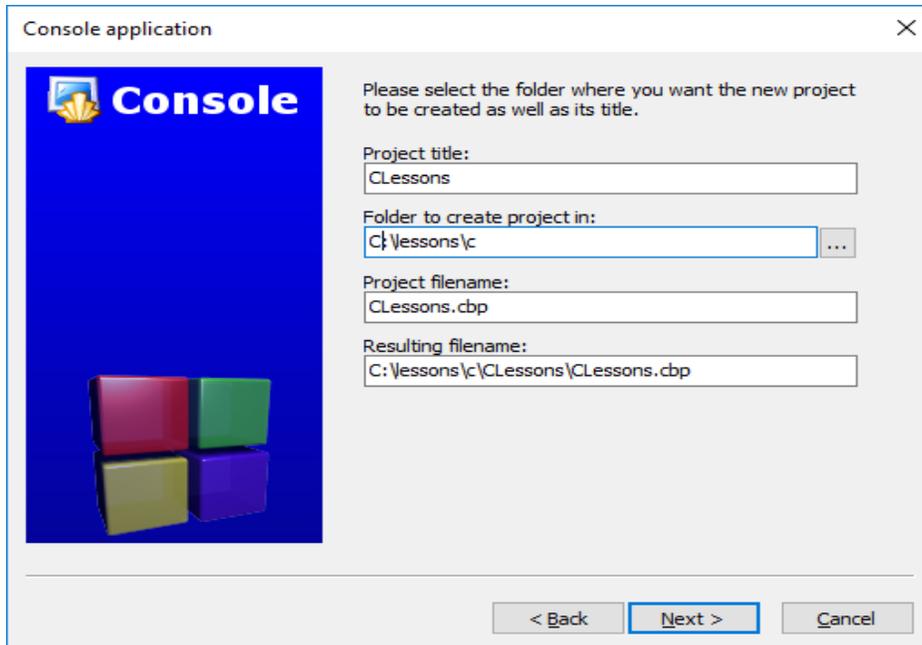


Press Next button

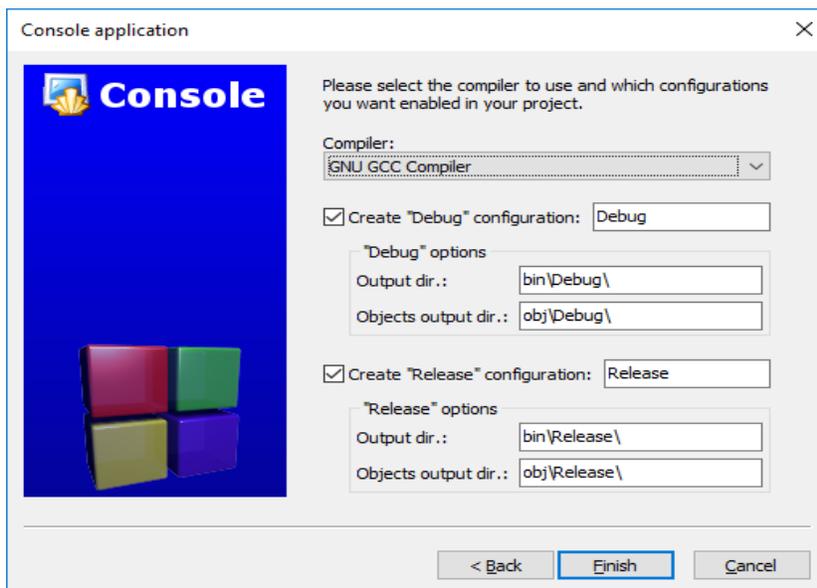


Select C and then press Next button

Using File Explorer make a Folder to hold your C lesson files called C, then enter CLessons as the project name.

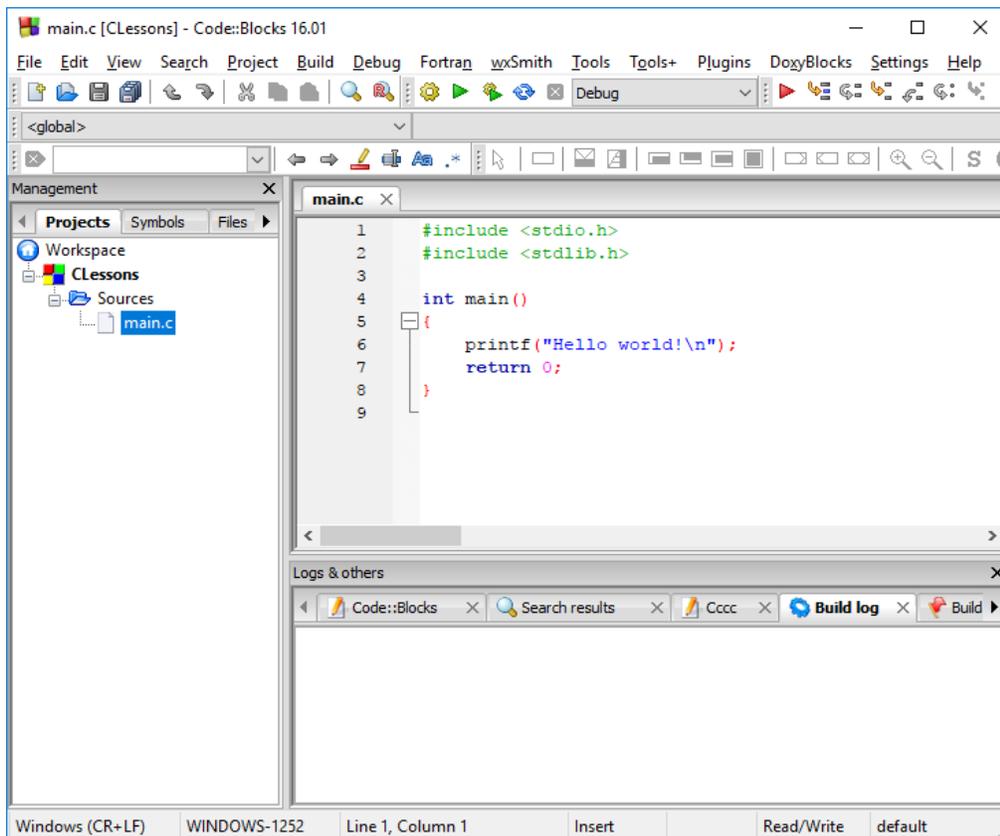


The next screen tells you what C Compiler the CodeBlocks is using.



You can just select Finish.

You should get something like this after expanding “Sources” folder and clicking on main.c in the Management window.

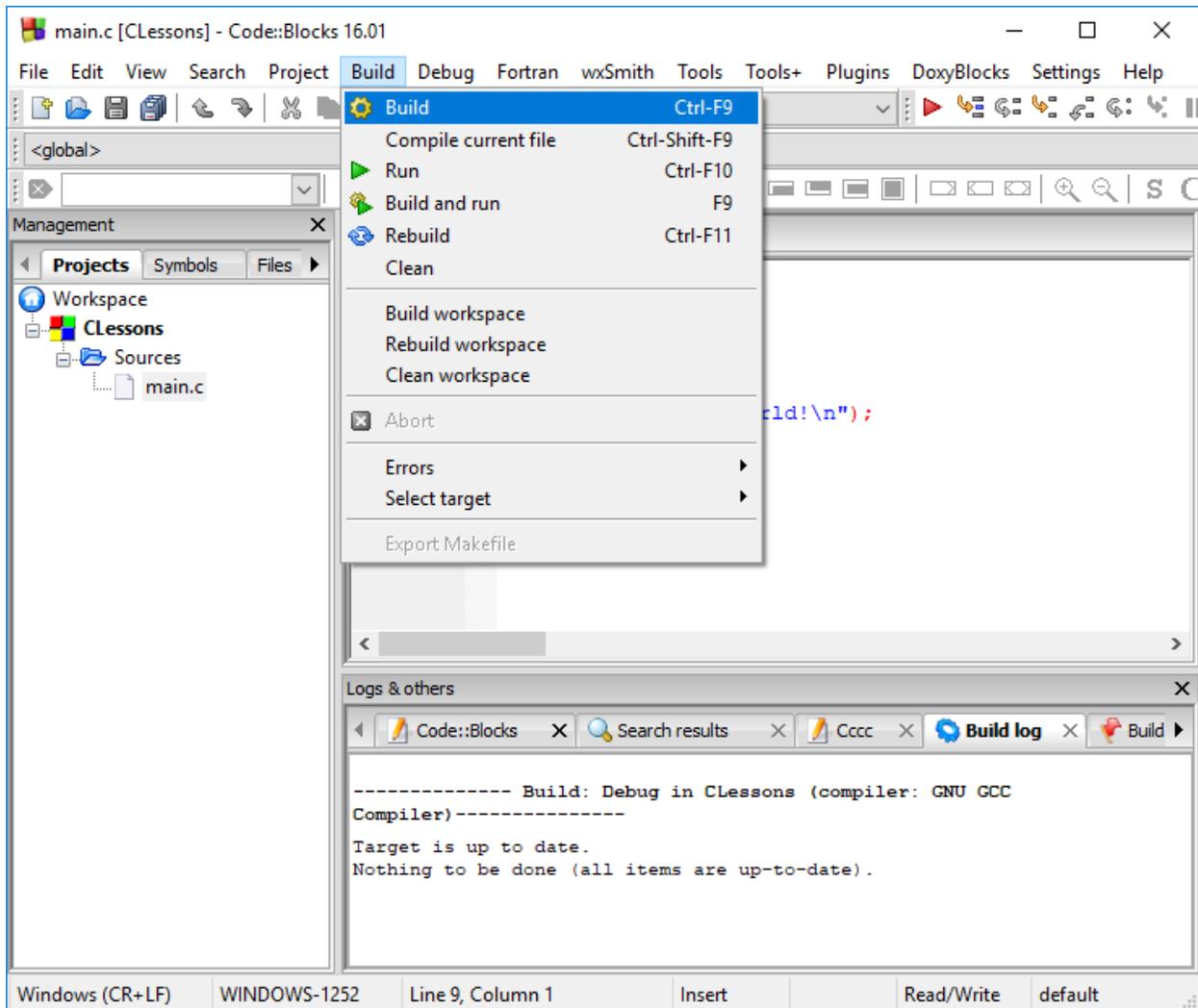


If you are using another C compiler then you need to type in the following code:

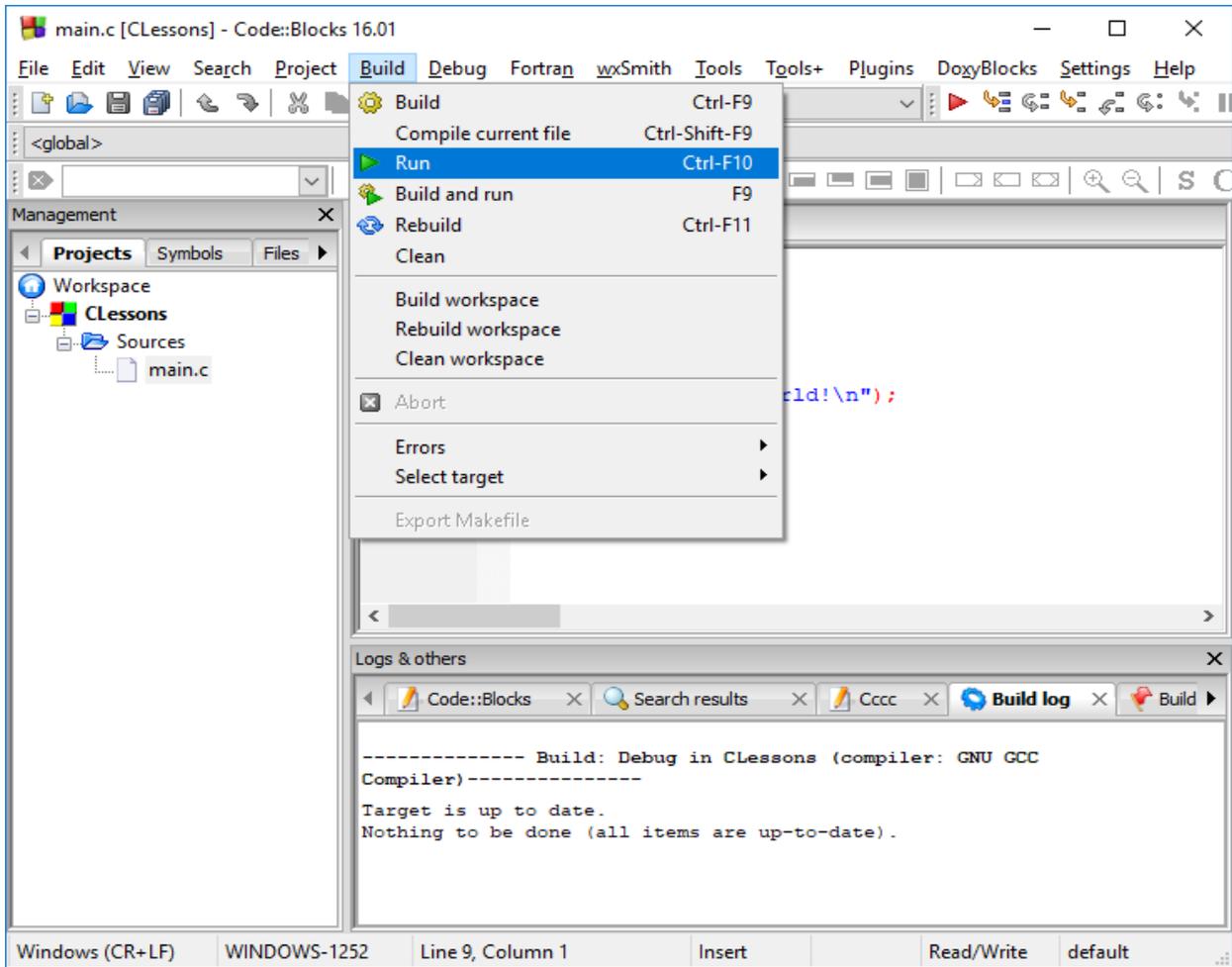
```
#include <stdio.h>
```

```
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

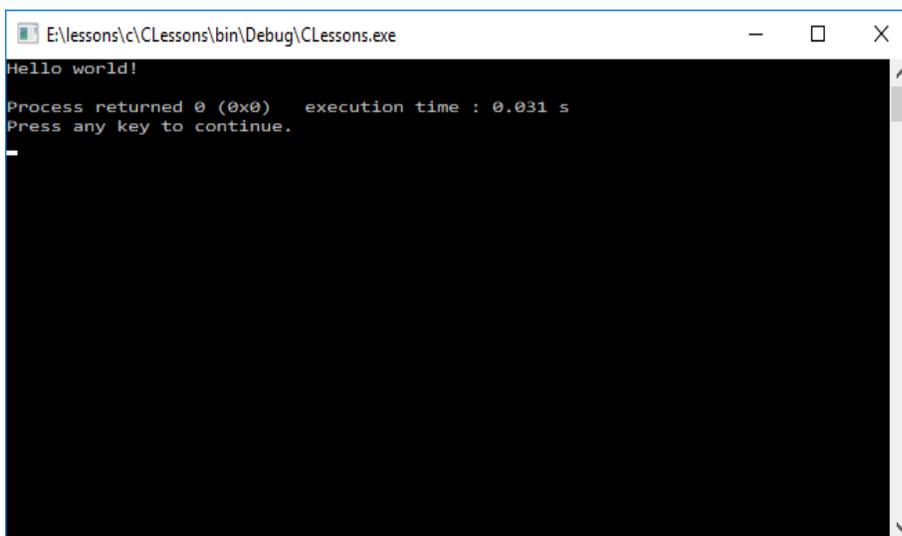
Next you need to build the program before you can run it. From the Build menu select Build.



If you have no errors, then you can run your program.
Select Run from the Build menu



You should get something like this:



A C program contains programming statements that tells the computer what to do. These programming statements are grouped together in a function enclosed by curly brackets, so each programming statement can execute one by one sequentially.

In a C program the main function is the first function to run. Here is the main function:

```
#include <stdio.h>  
  
int main()  
{  
    printf("Hello world!\n");  
    return 0;  
}
```

main is the function name and **printf("Hello world!\n");** is a programming statement printing Hello world! on the screen
Our first C program prints our "Hello World!" on the screen. If your compiler did not generate the sample program above for you then you will have to type it in your self, into your C compiler then build and run it.

A C program starts with the following include statement

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

The **#include** statement is called a **preprocessor** it tells the C compiler to use function definitions from the **stdio.h** file. These functions allow a program to print messages on the screen or get values from the key board.

The other include statement **#include <stdlib.h>** is used for other things we will discuss later and not necessary to use now.

The next programming statement is the main function definition header.

```
int main()
```

Inside the main function definition header we have the programming statements enclosed in curly brackets { } The open { curly bracket means to start the programming statements. The closing } curly bracket means to end the programming statements. Our first programming statement prints "Hello World" string message on the console screen.

```
{  
    printf("Hello world!\n");
```

printf is used to print out the message **Hello world!\n** to the screen. The **Hello world!\n** message is enclosed in double quotes **"Hello world!\n"**. Anything enclosed in double quotes is known as a string value. The '\n' at the end of the string means to start a new line on the output screen. The **printf** statement will print the string message **"Hello world!\n"** that are specified within the round brackets ().

The main function return's a value using the return statement. The **return** is a **keyword** that is used to specify what value is to be returned. The main function usually returns a 0 meaning every thing is okay.

```
    return 0;  
}
```

Here is the main function again with the preprocessor include statement.

```
#include <stdio.h>  
  
int main()  
{  
    printf("Hello world!\n");  
    return 0;  
}
```

The include statement is not part of the main function but is usually at the top of every C program.

Terminology

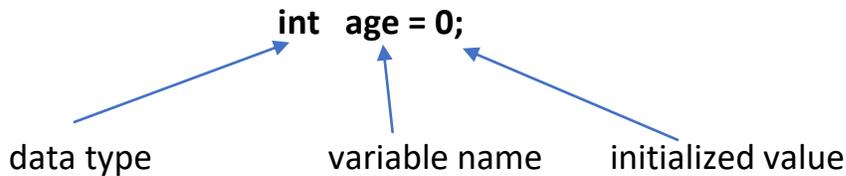
Before we proceed it is important to understand the terminology: data type, variables, functions, programming statements and objects.

data type	States what type of data value a variable is to represent and store.
Variable	Stores a string or numeric value represented by a name. All variables have a specified data type stating what type of data is to be stored.
programming statement	Is an instruction containing commands to perform a desired action, like printing a value on the screen, get a value from the key board or calculate a certain value.
Function	Contains programming statements that are executed sequentially telling the computer what to do.
Preprocessor	Instructions to the compiler to include additional files to be compiled.

variables

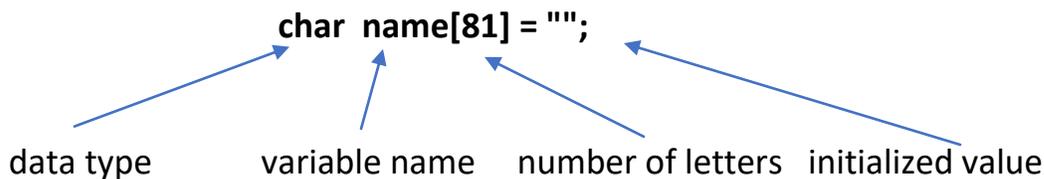
Programing is all about storing values and doing operations on them like addition and subtraction. Values my be whole numbers like 5, decimal number like 10.5 or a text message like "Hello World". Text messages are enclosed in double quotes are also known as strings. A program use variables represented by a identifier name to store values. The value is actually stored in the computer memory when the program runs.

To use a variable you need to declare it first. A variable specifies a **data** type and a name and an optional initialized value. Data types specify what kind of data the variable is going to store. We use **int** data type for whole numbers, **float** or **double** data types for decimal numbers. Double variables have more precision then float numbers (more accurate). **char** data types to represent a single letter or a group of letters to represent a string message. In the following example we declare the following variable **age** that has an **int** data type to store whole numbers.



The **variable** name is age and is initialized to the value of 0, it is also good practice to initialize variables when you declare them to give them a default value. All variable declarations end in a semi colon ;. All variable names should represent the value it is storing. For example **age** would represent somebody's age.

Text string messages are stored in **char** data type variables. A char data type represents a single letter character. To represent a string text message you need many letters, you need to specify how many letters you need. To represent a text message for a whole line we use 80 letters. The following variable represents a name that holds 80 letters. We add 1 more letter for a termination letter. The length of the text message is specified in square [] brackets as follows.



Our initialize value is an empty string represented by 2 quotes side by side with no space between them. We also use the character length of 81 rather than 80 because we need 1 extra character for the end of string character '\0' also known as **NULL** character;

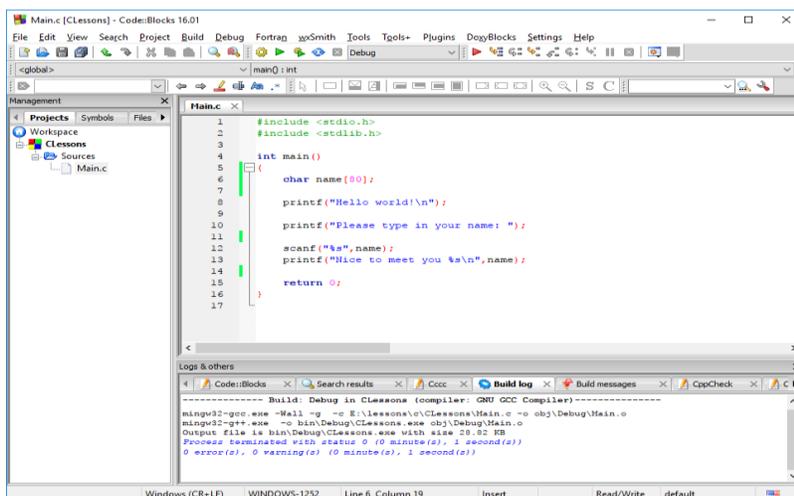
Now we know about variables we can use them in our program. The age variable will represent a person's age and the name variable will represent a person's name. We will get these values from the keyboard. Alternatively you can initialize these variables directly, like this:

```
int age = 24;  
char name[] = "Tom";
```

We can also specify the length as [] which indicates the length is automatically specified by the length of the initialized string. We will now ask the user to type in their name and then greet them. Type in the following statements in the C editor before and after the "Hello World" printf statement.

```
char name[81];
printf("Hello world!\n");
printf("Please type in your name: ");
scanf("%s",name);
printf("Nice to meet you %s\n",name);
```

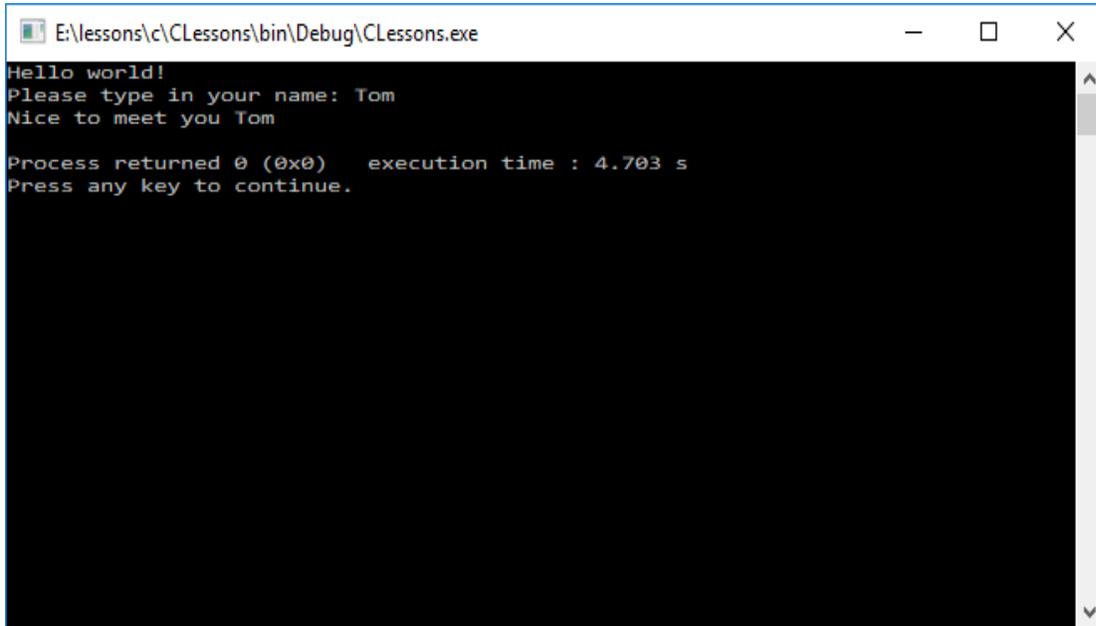
You should have something like this:



The complete program looks like this:

```
#include <stdio.h>
int main()
{
char name[81];
printf("Welcome to my program\n");
printf("Please type in your name: ");
scanf("%s",name);
printf("Nice to meet you %s\n",name);
return 0;
}
```

Now build and run your program, and enter the name "Tom". You will get something like this:



```
E:\lessons\c\CLessons\bin\Debug\CLessons.exe
Hello world!
Please type in your name: Tom
Nice to meet you Tom

Process returned 0 (0x0)   execution time : 4.703 s
Press any key to continue.
```

We will first ask the user to type in their name using the **printf** statement.

```
printf("Please type in your name: ");
```

We then obtain the user name from the keyboard. We first declare a char string variable called **name** that will be used to store the persons name and then use **scanf("%s",name);** statement to read a char string from the keyboard. The value entered from the keyboard is placed in the char string variable name. Variables are used to store values and must be declared before using them.

```
char name[81];
```

Variables in C are written at the top of the function, although some compilers allow you to place them as you use them. Char string variables are used to store string message. The maximum length of the string message must first be specified, which is 80 in this case. The maximum length is enclosed in square brackets []. The data type of the char string variable is **char**. Char stands for a character, our string message can hold up to 80 characters. The number 81

represents the maximum number of characters that can fit in one line of screen, plus 1 extra character for the end of string character.

The **scanf** statement is used to obtain values from the keyboard. The scanf statement must know what kind of data it is supposed to read from the keyboard. A format specifier is used to specify what kind of data is to be read. Format specifiers start with the percent character ‘%’

%c is used for char data like ‘a’

%s is used for char string data like “hello”

%d is used for int whole numbers like 5

%f is used for float decimal numbers like 10.5

%lf is used for double decimal numbers like 10.5

The %s format specifier is enclosed in double quotes followed by a comma and the char string variable name all enclosed in round brackets (). The round brackets introduce the format specifier, comma and the variable name.

```
scanf("%s",name);
```

When the **scanf** statement is executed, the string value that is read from the keyboard is stored in the variable name.

printf is then used to print out the string message “Nice to meet you” and the name of the user that was stored in the variable name.

```
printf("Nice to meet you %s\n",name);
```

Again, we use the format specifier ‘%s’ to tell the print statement what data type to print out, (string data). Notice the format specifier is specified after the “Nice to meet you” message, so that we can print the name of the person right after the message. The ‘\n’ new line is after the format specifier to start a new line. The name of the person is printed at the position where the format specifier is located.

Nice to meet you Tom

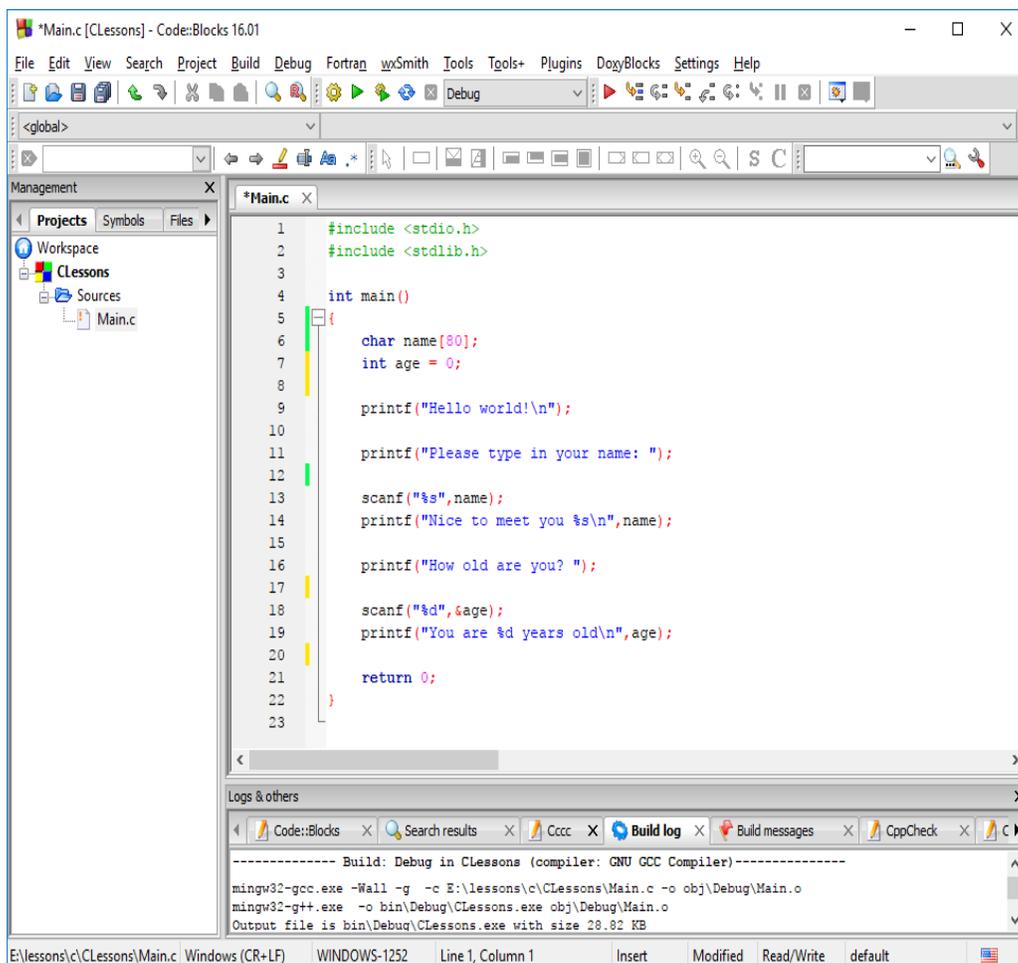
We now continue our program ask the user how old they are. Type in the following statements at the end of your program.

```
printf("How old are you? ");
scanf("%d",&age);
printf("You are %d years old\n",age);
```

Put the variable age after the variable name at the top of your program right after the char name[81] variable.

```
int age = 0;
```

Make sure you save your file before proceeding. You should have something like this:



The screenshot shows the Code::Blocks IDE with a C program named Main.c. The code is as follows:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char name[80];
7      int age = 0;
8
9      printf("Hello world!\n");
10
11     printf("Please type in your name: ");
12
13     scanf("%s",name);
14     printf("Nice to meet you %s\n",name);
15
16     printf("How old are you? ");
17
18     scanf("%d",&age);
19     printf("You are %d years old\n",age);
20
21     return 0;
22 }
23
```

The IDE also shows a build log at the bottom with the following output:

```
----- Build: Debug in CLessons (compiler: GNU GCC Compiler) -----
mingw32-gcc.exe -Wall -g -c E:\lessons\c\CLessons\Main.c -o obj\Debug\Main.o
mingw32-g++.exe -o bin\Debug\CLessons.exe obj\Debug\Main.o
Output file is bin\Debug\CLessons.exe with size 28.82 KB
```

The complete program now looks like this:

```
#include <stdio.h>
int main()
{
    char name[81];
    int age = 0;

    printf("Hello world!\n");
    printf("Please type in your name: ");
    scanf("%s",name);
    printf("Nice to meet you %s\n",name);
    printf("How old are you? ");
    scanf("%d",&age);
    printf("You are %d years old\n",age);

    return 0;
}
```

Build and run the program and enter Tom for name and 24 for age, you should get something like this.

```
E:\lessons\c\CLessons\bin\Debug\CLessons.exe
Hello world!
Please type in your name: Tom
Nice to meet you Tom
How old are you? 24
You are 24 years old

Process returned 0 (0x0) execution time : 12.453 s
Press any key to continue.
```

Recapping:

In our program we declare an **int** variable called age and initializes it with the default value 0.

```
int age = 0;
```

We then ask the user to enter there age using printf statement

```
printf("How old are you? ");
```

We use scanf to enter the age from the keyboard and assign the value to the variable age.

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

We use the %d format because we want to read in an int value. The variable age has a '&' in from of it. It is used to specify the location in computer memory where the value obtained from the keyboard is to be placed. This location is known as a memory address. Without the '&' the value of the age variable is obtained rather than the location. With the '&' preceding the variable age then the address of the variable age is obtained. The scanf function needs to know the address of the variable, so it can place the value from the keyboard into it.

Using the **printf** statement we print out the message "You are", the value stored in the variable age and the message "years old" is printed to the computer screen.

```
printf("You are %d years old\n",age);
```

The %d format specifier is used to print out the value of the variable age. The value of the age is printed after the "You are " message and the "years old" message. Note the age does not have a & because we want to obtain the value from it.

Reading single characters from the keyboard

Reading a single character from the keyboard can be a challenge because the enter key stays in the input stream and is never removed. This causes preceding values not to be read properly.

Example

```
char ch;  
scanf("%c",&ch);
```

There are many ways to solve this dilemma.

Solution 1:

Read the enter key without storing it using %*c

```
scanf("%c%*c",&ch);
```

Solution 2:

Read the enter key with **additional** scanf or the getchar() function

```
scanf("%c",&ch);
```

or

```
ch = getchar();
```

solution 3:

Remove the enter key before hand when reading another variable by placing a **space** in front of a format specifier.

```
scanf(" %s",name);
```

solution 4:

You can clear the input stream with fflush

```
fflush(stdin)
```

reading a whole line from the key board including spaces

The **scanf** %s format specifier only reads individual words in a line. To read a whole line you need to use the square bracket specifier that states what string characters you want to read. To read a whole line you need to read all lines up to the newline \n character. This format **%[^\n]** specifier will read a whole line from the keyboard. The **^\n** means do not read \n, so when a new line '\n' is encountered scanf stops scanning.

```
scanf("%[^\n]",line);
```

There are other C functions that just read string lines like **gets** and **fgets**.

```
gets(line); # read a line of unlimited characters
```

```
fgets(stdin,81,line) # read a line up to 81 characters
```

fgets is the preferred choice since it specifies the maximum character to be read. **stdin** is the keyboard input stream, where as **stdout** is the console screen output stream.

CONSTANTS

Constants let you associate a value with a label or a name. It is not good to put hard codes values in a program because nobody knows what they mean. Constants allows you to have **labels** as an identifier in your program to represent a value. Once the label identifier is set it cannot be changed. Constants allow values to have a meaning represented by a label identifier. Constants represents a value that **has no** memory location. Constant value once set cannot be changed.

A good constant example is the value 81 that we used to represent the maximum characters in a screen line.

```
#define MAX_CHARS 81
```

We would use a constant like this:

```
char name[MAX_CHARS];
```

When the compiler sees the constant label **MAX_CHARS** it substitutes the value 81, it is a direct substitution. Constants represents a value that **has no** memory location.

```
char name[81];
```

There are 2 ways to make constants in C.

Using the **#define** preprocessor.

```
#define MAX_CHARS 81
```

Using the **const** keyword.

```
const int MAX_CHARS = 81;
```

When using the **const** keyword **MAX_CHARS** is known as a **name** identifier because it represents a value that **has** a read only memory location, meaning it cannot be changed when the program is running. Constant labels and name identifier usually start with a capital letter or all capital letters to indicate this label name is a constant. The **const** keyword is the preferred way, but many old C compilers cannot recognize or handle it properly. So, we will still use the **#define** preprocessor. At the top of your program just below the **#include** statements type in

```
#define MAX_CHARS 81
```

Note important: DO NOT PUT A SEMICOLON AFTER THE NUMBER 81 OR ELSE YOUR PROGRAM WILL HAVE MANY ERRORS

Once you have your constant defined then you need to put the constant in the same place where the number 81 is:

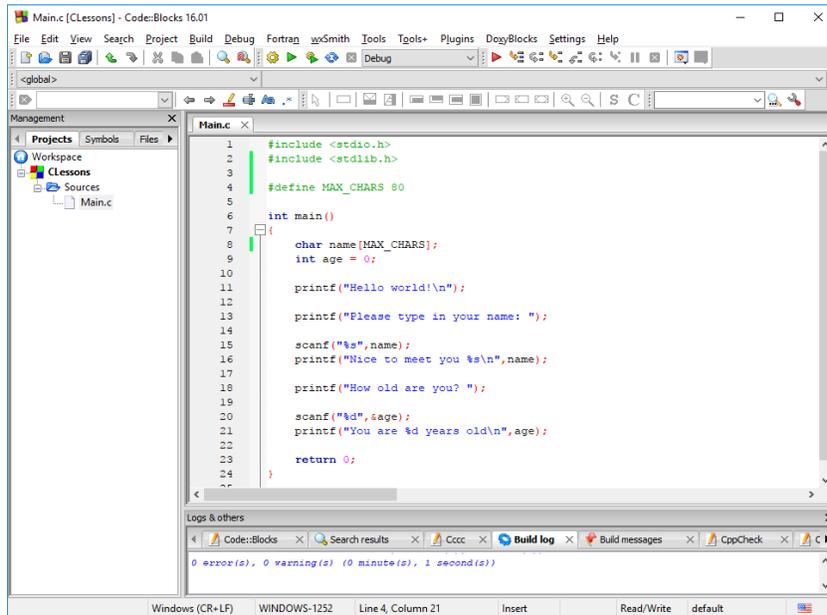
Before:

```
char name[81];
```

After

```
char name[MAX_CHARS];
```

You should now have something like this:



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX_CHARS 80
5
6 int main()
7 {
8     char name[MAX_CHARS];
9     int age = 0;
10
11     printf("Hello world!\n");
12
13     printf("Please type in your name: ");
14
15     scanf("%s",name);
16     printf("Nice to meet you %s\n",name);
17
18     printf("How old are you? ");
19
20     scanf("%d",&age);
21     printf("You are %d years old\n",age);
22
23     return 0;
24 }
```

The complete program is now:

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

```
#define MAX_CHARS 81
```

```
int main()
{
char name[MAX_CHARS];
int age = 0;
printf("Hello world!\n");
printf("Please type in your name: ");
scanf("%s",name);
printf("Nice to meet you %s\n",name);
printf("How old are you? ");
scanf("%d",&age);
printf("You are %d years old\n",age);
return 0;
}
```

Compile and run your program and make sure it still works.

This is a lot to digest. You should need to know all these basics concepts, to understand programming. All programming is based on variables, values, addresses, programming statements and functions. If you do not understand these concepts, just keep on doing Lesson1 repeatedly until it makes some sense.

If you have got this far then you will be a great C programmer soon.

Most people find Programming difficult to learn. The secret of learning program is to figure out what you need to do and then choose the right programming statement to use. If you want to print messages and values to the screen you use a **printf** statement. If you want to get values from the keyboard, you use a **scanf** statement.

You should concentrate on getting your programs running rather than understand how they work. Once you get your programs running and you execute them understanding will be come much easier. Understanding will now be much easier, because you can now make an “association connection” to the program statement that is running that produces the desired input or output action.

C Data Types

Data types state what kind of data a variable is suppose to represent. C has many data types that can be used to represent various kinds of data as follows:

Data Type	Size	Min value	Max Value	Example
Char	8	-128	127	char x = 100;
Short	16	-32768	32767	short x = 1000;
Int	32	-2^{31}	$2^{31}-1$	int x = 10000;
Long	32	-2^{31}	$2^{31}-1$	long x = 10000;
float	32	-1.4E-45	3.4E38	float f = 10.5;
Double	64	-4.9E-324	4.9E-324	double d = 10.57654;

The above data types are signed data type representing both positive and negative numbers. The **double** data type is much more accurate than the **float** data type, it can represent many more decimal digits. (E means exponential)

C also has unsigned data types **unsigned char**, **unsigned short**, **unsigned int** and **unsigned long**.

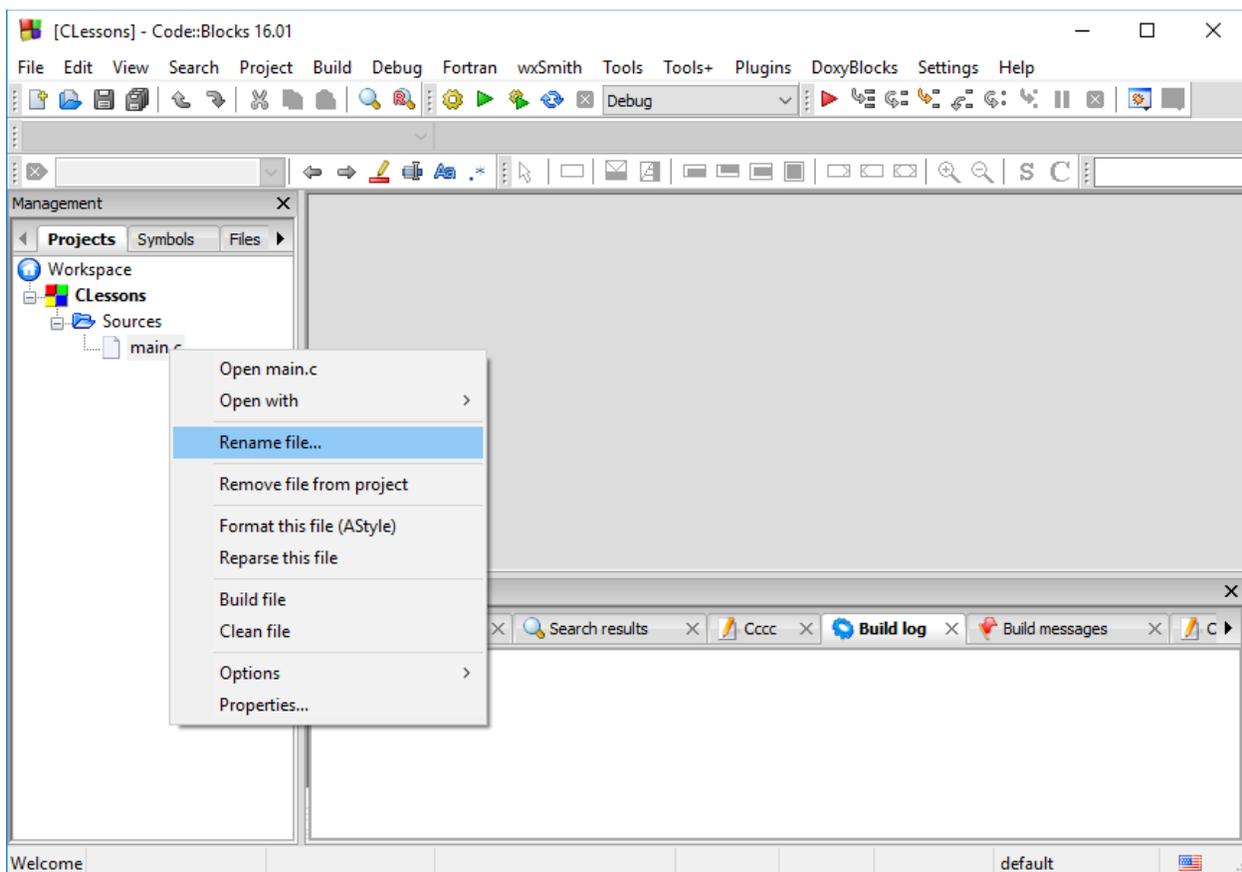
Data Type	Size	Min value	Max Value	Example
unsigned char	8	0	256	char x = 100;
unsigned short	16	0	65535	ushort x = 1000;
unsigned int	32	0	$2^{32}-1$	uint x = 10000;
unsigned long	32	0	$2^{32}-1$	ulong x = 10000;

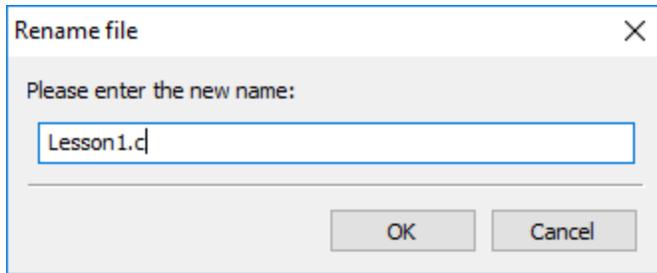
Lesson 1 Homework

Make a C program file called homework1.c that asks someone what their profession title is and annual salary is. Make a char[81] title and a float salary. Then print out a message like this: "I am a Manager and I make \$100,000 dollars per year!". When the program starts print out a welcome message.

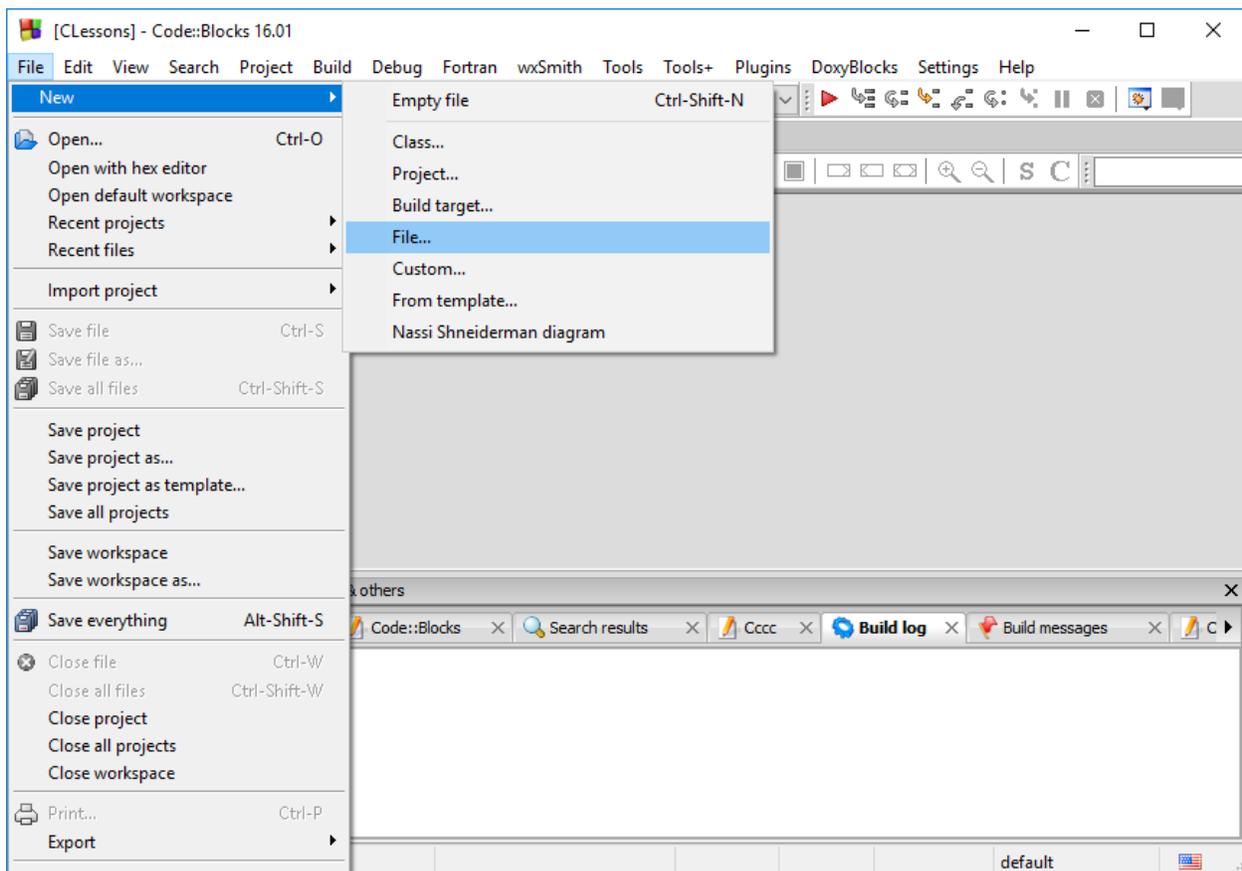
LESSON 2 FUNCTIONS

Functions allow you to group many programming statements together so that you can reuse them repeatedly in your C Program. The most common function is the main function that starts a C program, which we used previously in Lesson 1. A program may have many functions. Each function has a dedicated purpose, some action to perform. Functions usually are defined at the top of the program in order as they are used. The main function is the last one because it will call all the preceding functions. When a function is called in a programming statement it means it is executed. C also has many built in functions that you can use, that make C programming easier to do that you will learn later through these lessons. It is now time to add functions to our previous Lesson 1 program. We will make a **welcome**, **enterName**, **enterAge** and **printPerson** functions. Before proceeding, you may want to save your previous main.c file as Lesson1.c for future reference. Close file main.c and in the Management Window right click on main.c and rename Lesson1.c

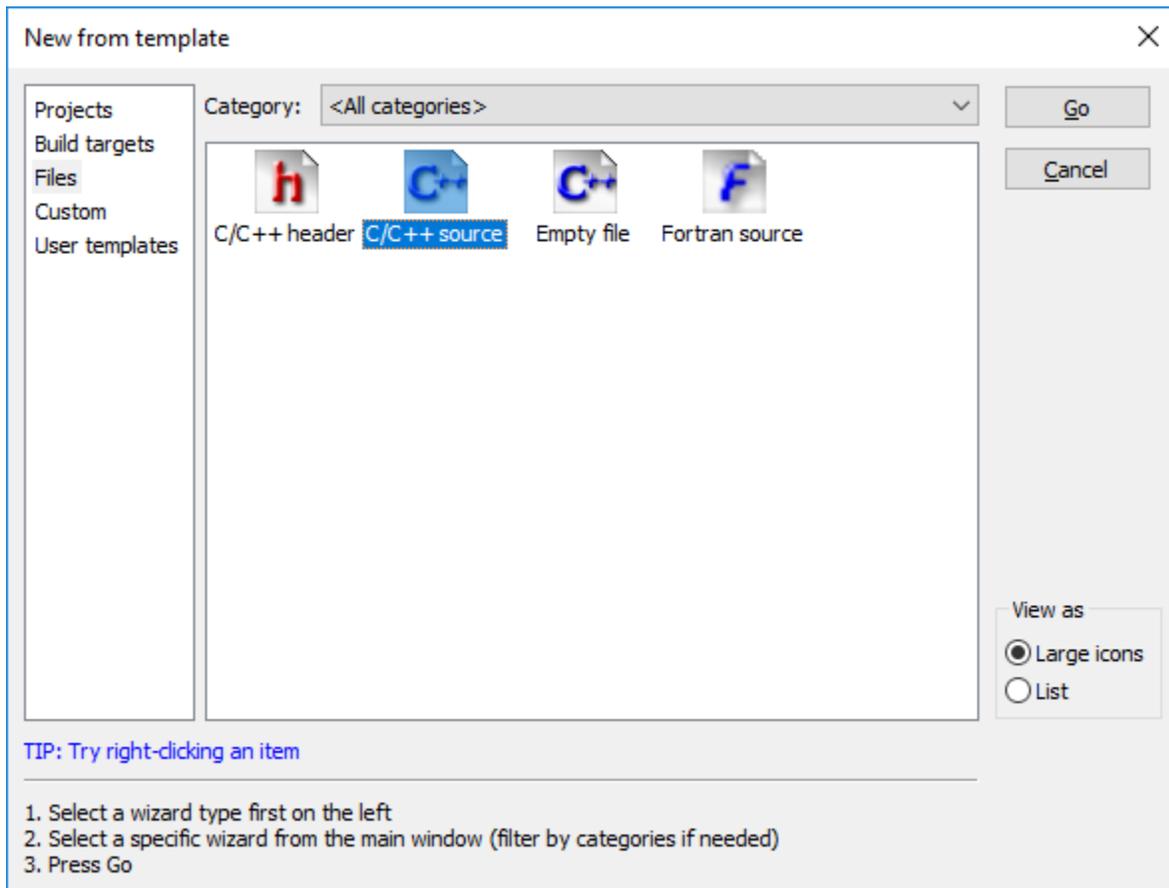




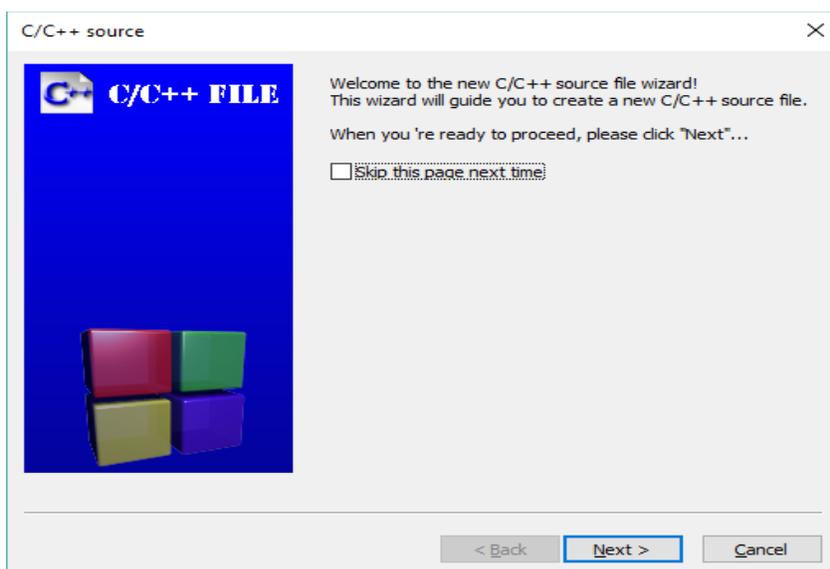
Now make a new C source file called Lesson2.c. From the File Menu select New then File.



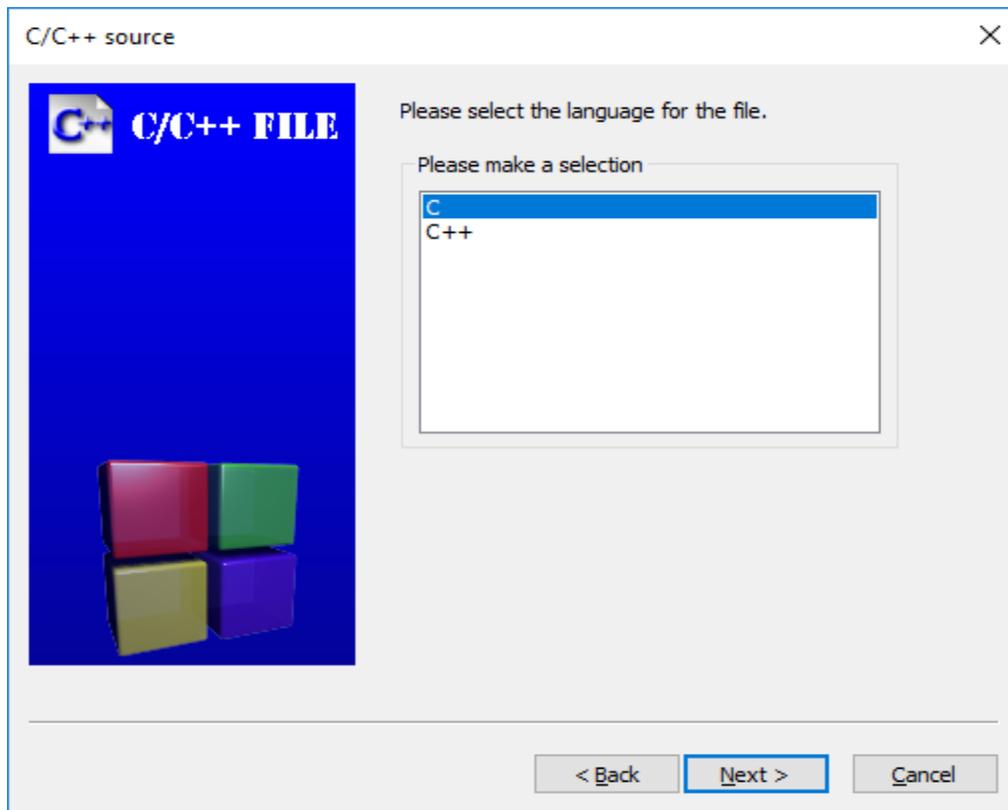
Select C Source File template



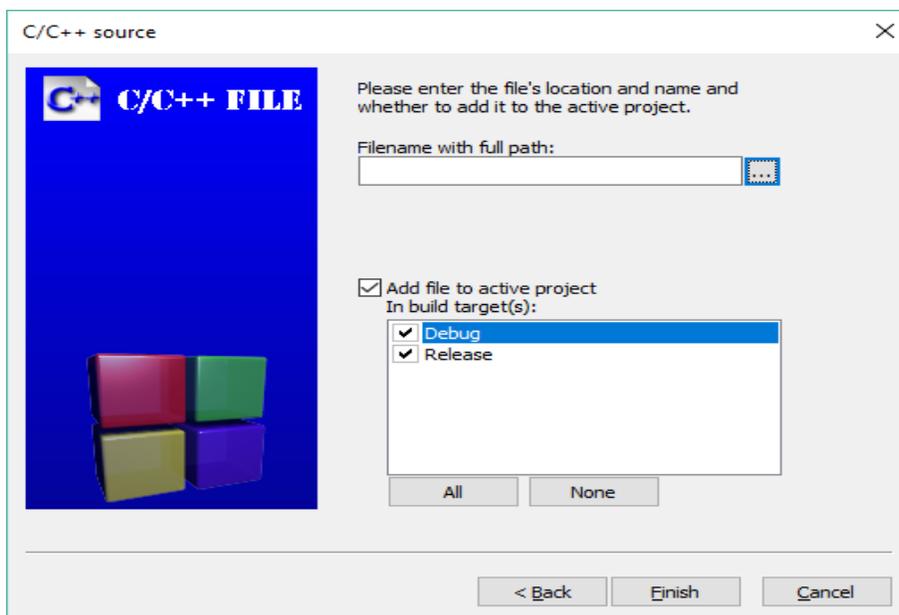
Press GO button



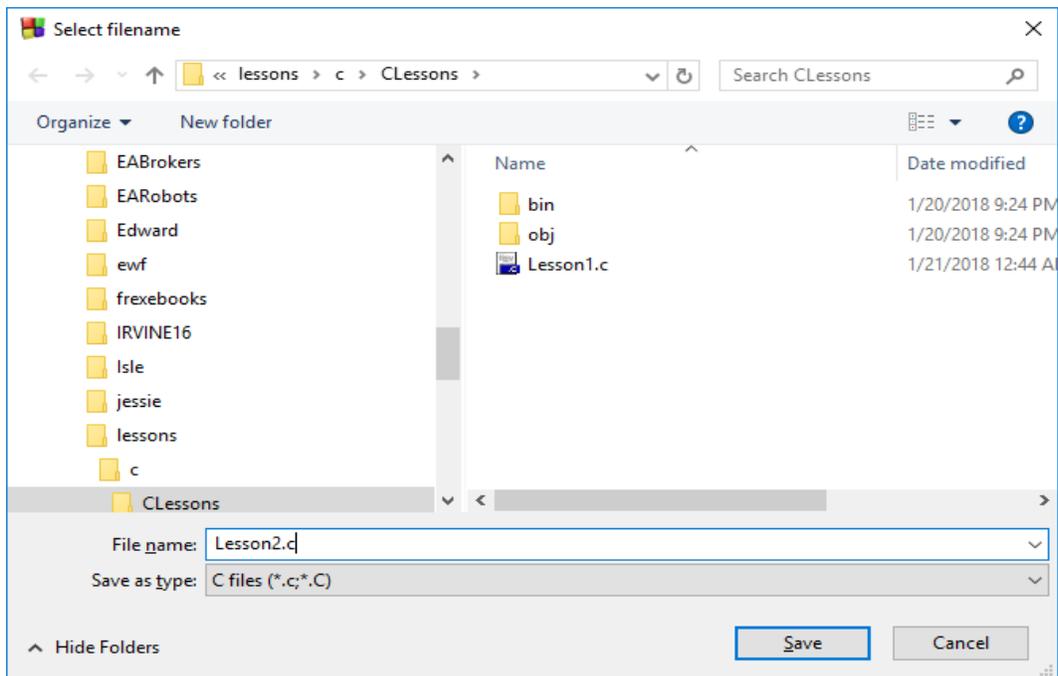
Press Next button
Select C



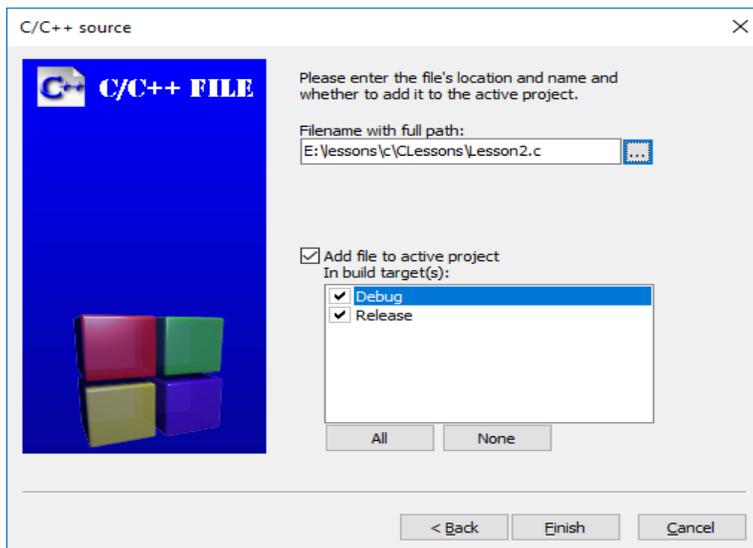
Press Next



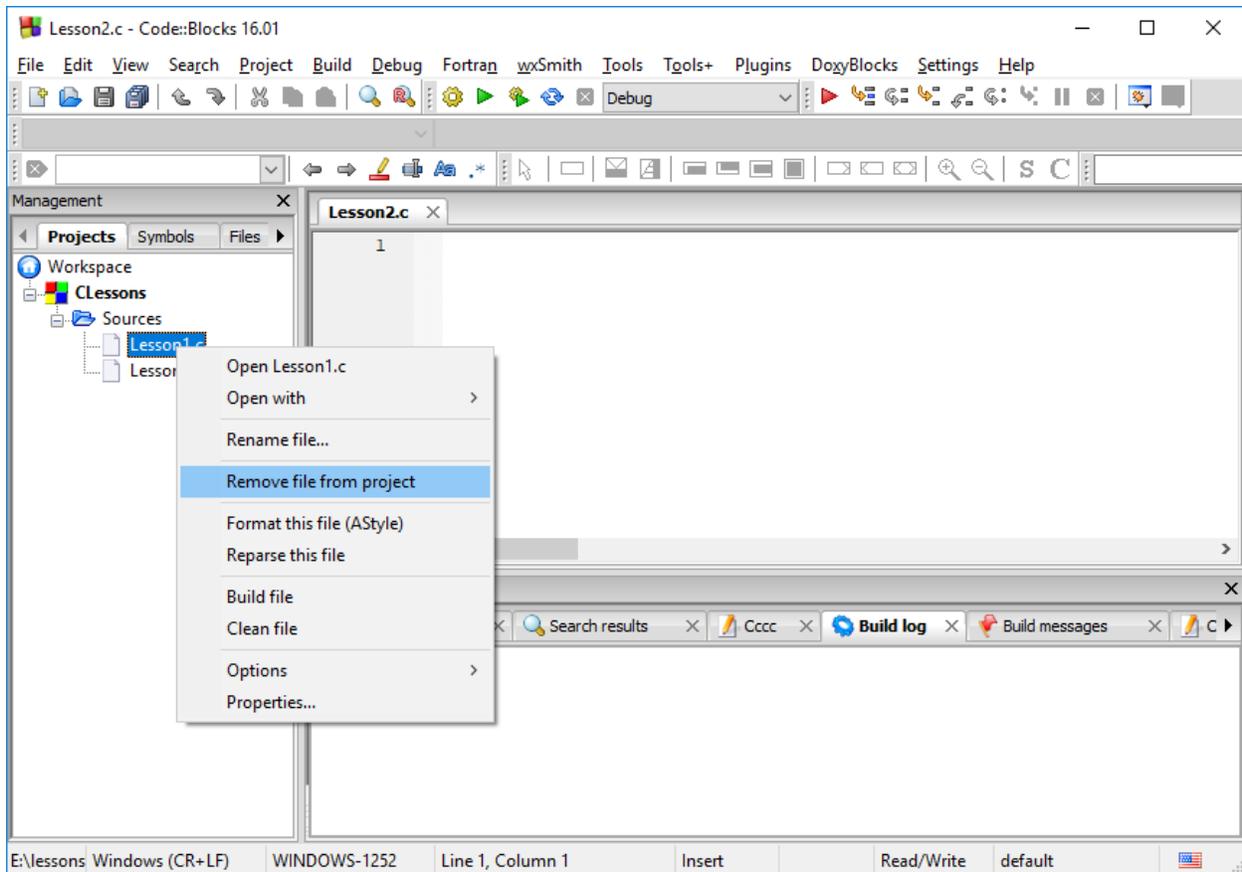
Check Debug and Release Checkbox's, then select file browse button [...]
And enter file name Lesson2.c



Press Save Button, you should get the following screen



Before pressing "Finish" button make sure The Debug and Release check boxes are checked. You now need to remove the Main.c file or Lesson1.c file from the project. A project can only have one c file with a main function.



In your Lesson2.c file type in the following code.

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

```
#define MAX_CHARS 81
```

```
void welcome();
```

```
void enterName(char name[MAX_CHARS]);
```

```
int enterAge();
```

```
void printPerson(char name[MAX_CHARS], int age);
```

```
void welcome()
```

```
{  
    printf("Hello World\n");  
}
```

```

void enterName(char name[MAX_CHARS])
{
    printf("Please type in your name: ");

    scanf("%s",name);
}

int enterAge()
{
    int age;

    printf("How old are you? ");
    scanf("%d",&age);
    return age;
}

void printPerson(char name[80], int age)
{
    printf("Nice to meet you %s\n",name);
    printf("%s You are %d age years old\n", name, age);
}

int main()
{
    char name[MAX_CHARS];
    int age;

    welcome();
    enterName(name);
    age = enterAge();
    printDetails(name, age);
    return 0;
}

```

You should now build and run the program. Enter Tom for name and 24 for age, you should get this:

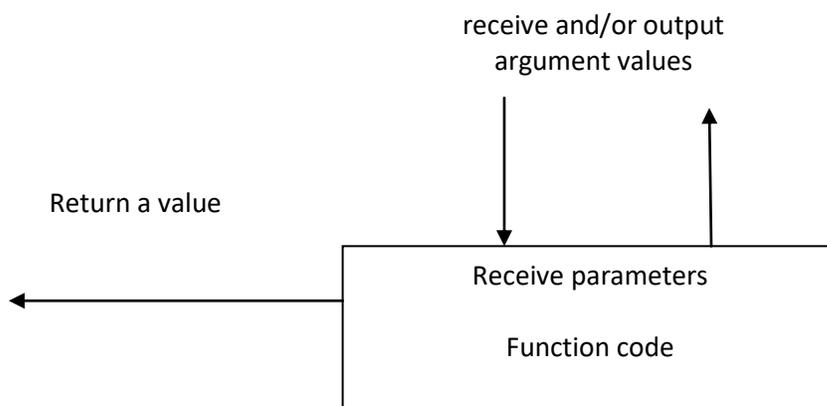
```
E:\lessons\c\CLessons\bin\Debug\CLessons.exe
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 age years old

Process returned 0 (0x0)   execution time : 7.548 s
Press any key to continue.
```

Functions make your program more organized and manageable to use. Functions have many different purposes. Function can receive values, return values, receive and return values or receive or return no value.

return_datatype function_name (parameter_list)
parameter list = data_type parameter_name [,data_type parameter_name]

Functions return values using the **return** statement and receive and/or output values through the **parameter list**. The return data type specifies what kind of data is returned or received. In Lesson 1 we were introduced to the int, float, double and char data types. Think that a function is like a factory that receives raw materials, manufacturers a product then ships it out when completed.



Before you can use a function, you need to declare it. A function declaration is just the function definition header ending in a semicolon. A function declaration is also known as a function prototype. Here are the function prototypes for our lesson2 program.

```
void welcome();  
void enterName(char name[MAX_CHARS]);  
int enterAge();  
void printPerson(char name[MAX_CHARS], int age);
```

After the function prototype declarations the function definitions are written. The **welcome** function just prints "Hello World" and receives no values or returns no value. The **void** data type specifies no value is returned or received.

```
void welcome()  
{  
    printf("Hello World\n");  
}
```

The **enterName** function outputs the name through the parameter list. This is known as a function outputting a value. The address of the name variable is passed to the **enterName** function. The **enterName** function obtains the value from the keyboard and fills the name variable with the value.

```
void enterName(char name[MAX_CHARS])  
{  
    printf("Please type in your name: ");  
    scanf("%s",name);  
}
```

The **enterAge** function obtains the age value from the keyboard and returns a value using the return statement.

```
int enterAge()
{
    int age;

    printf("How old are you? ");
    scanf("%d",&age);
    return age;
}
```

The **printPerson** function receives a name and age value to print out, but return's no value. The **printPerson** function receives the name and age values through the parameter list.

```
void printPerson(char name[MAX_CHARS], int age)
{
    printf("Nice to meet you %s\n",name);
    printf("%s You are %d age years old\n", name, age);
}
```

The **name** and **age** inside the round brackets of the **printPerson** function definition statement are known as **parameters** and contain values to be used by the function. The parameters just pass values from the calling function and are not the same variables that are in the calling function. Although the parameter names and values may be same as in the calling function variable names, but they are different memory locations. The main purpose of the parameters is to receive values for the functions. The **name** parameter is an input/output parameter so it can receive a value and outputs a value. It can do this is because it contains the address of the variable that belongs to the calling function. In this case the calling function in the main function contains the **name** variable.

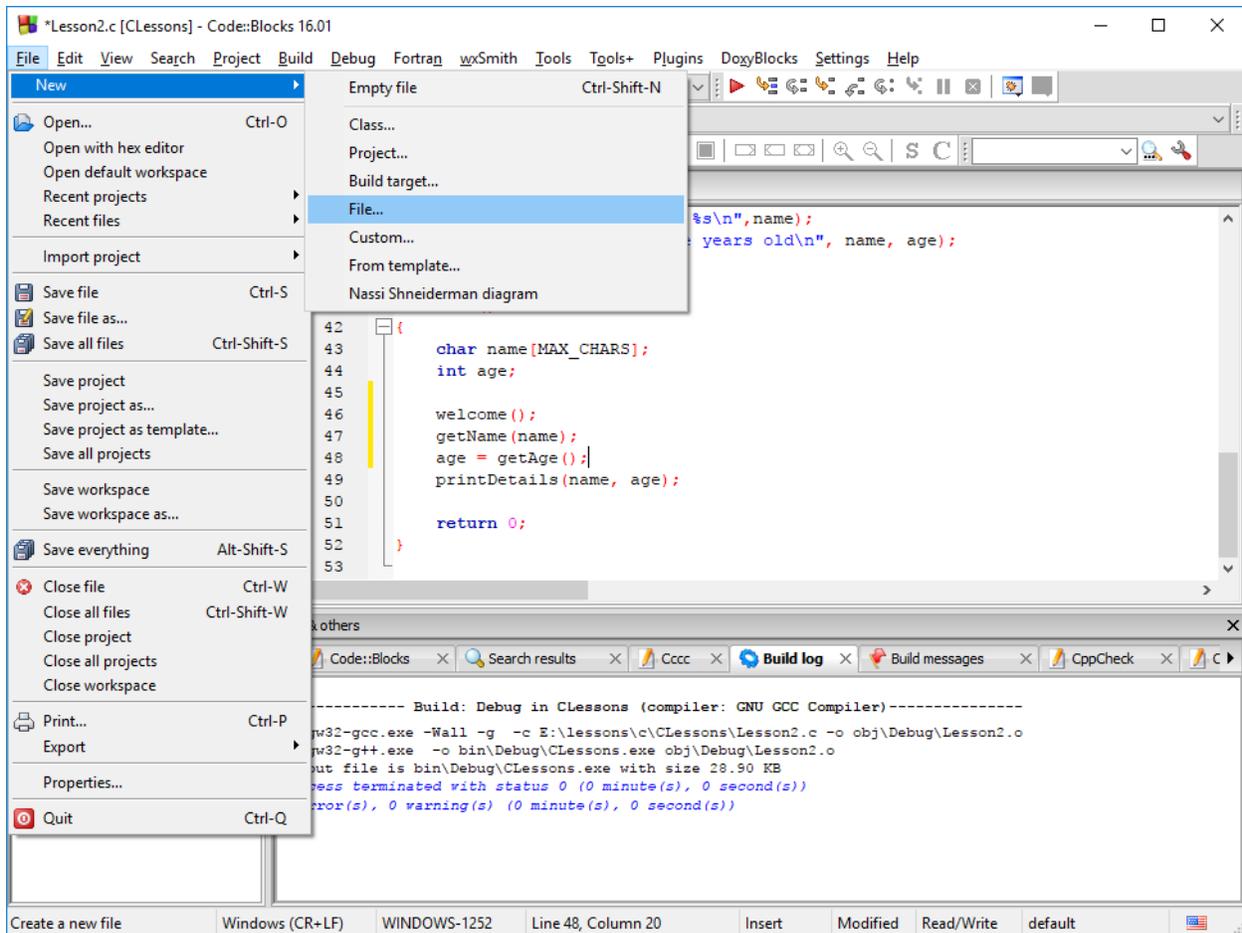
The main function call's the preceding functions to run them and store the values in variables and passes the stored variable values to the functions. Calling a function means to execute the function. The values that are passed to the called function from the calling function is known as **arguments**. The argument values are received by the function parameters. The function parameters store received values , the parameters can be used just like a variable in a function.

Variables inside a function are known as **local variables** and are known to that function only. Name and age are local variables in the main function but are also arguments sent to the printPerson function.

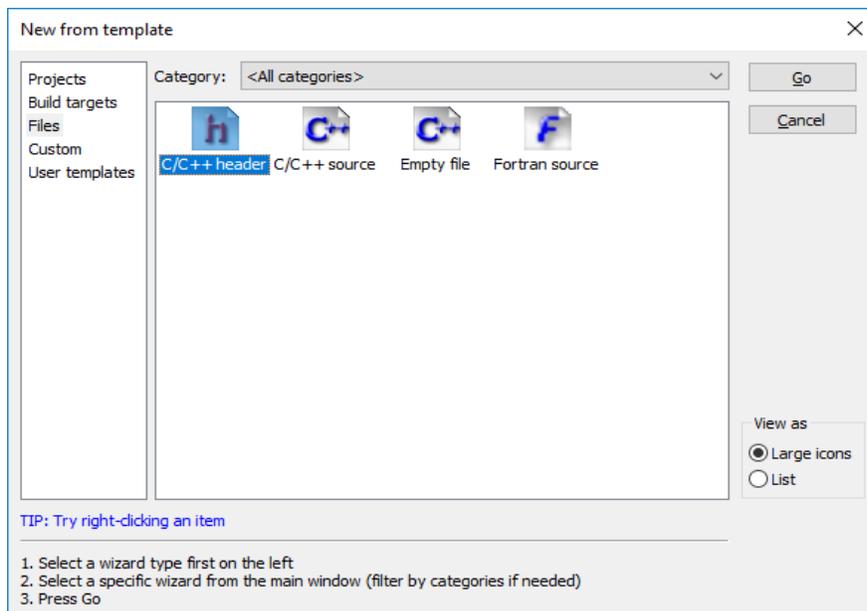
```
int main()
{
    char name[char name[MAX_CHARS]];
    int age;
    welcome();
    enterName(name);
    age = enterAge();
    printDetails(name, age);
    return 0;
}
```

Function prototypes are usually put into a header file. You should do the same. First make a C Header file called Lesson2.h

From the File menu select New then File



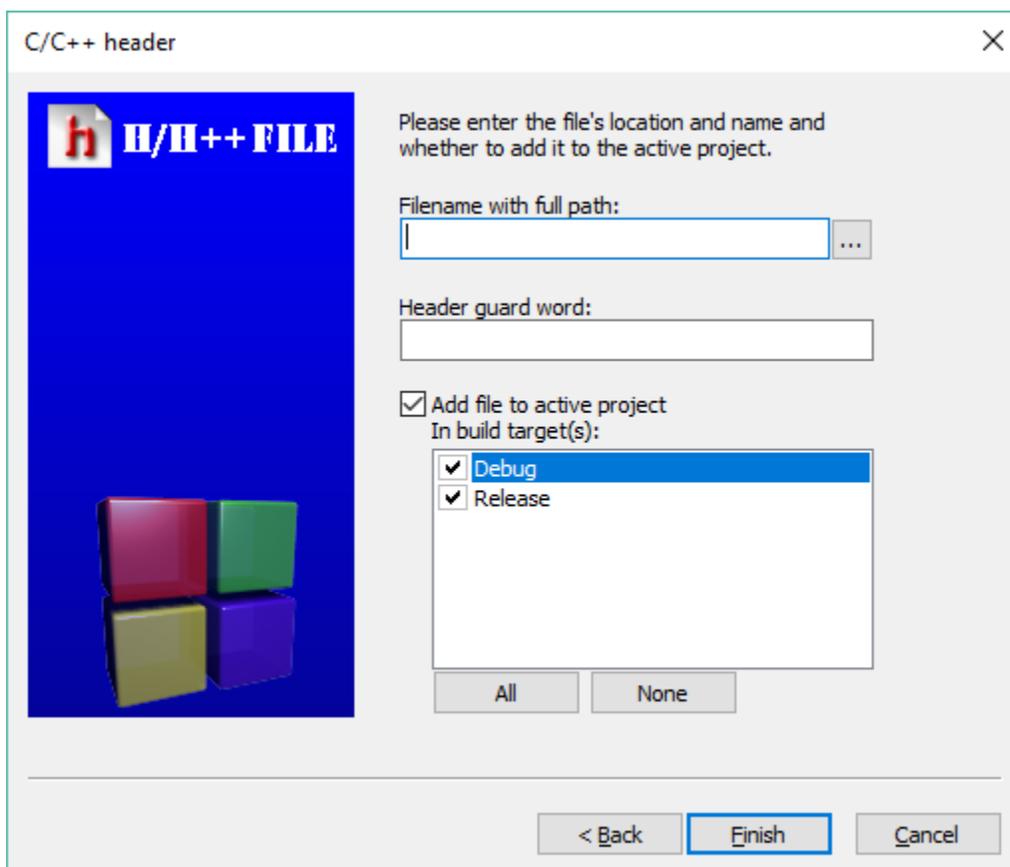
Select the C/C++ Header file type.



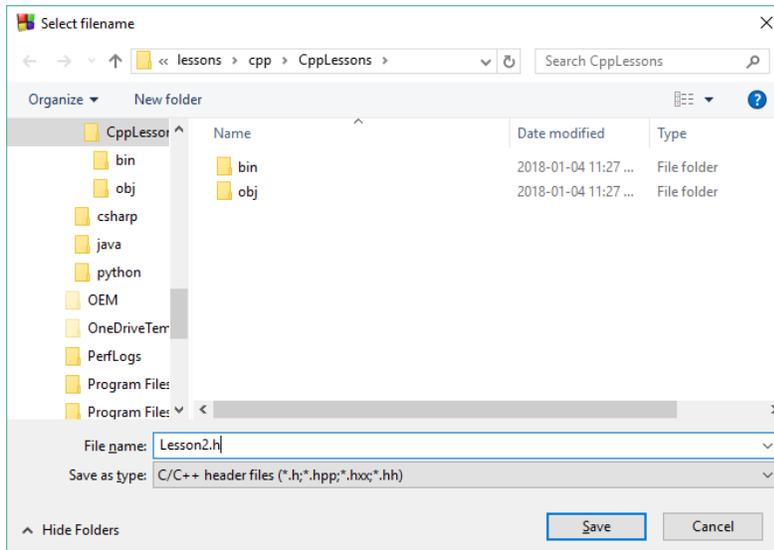
Press Go



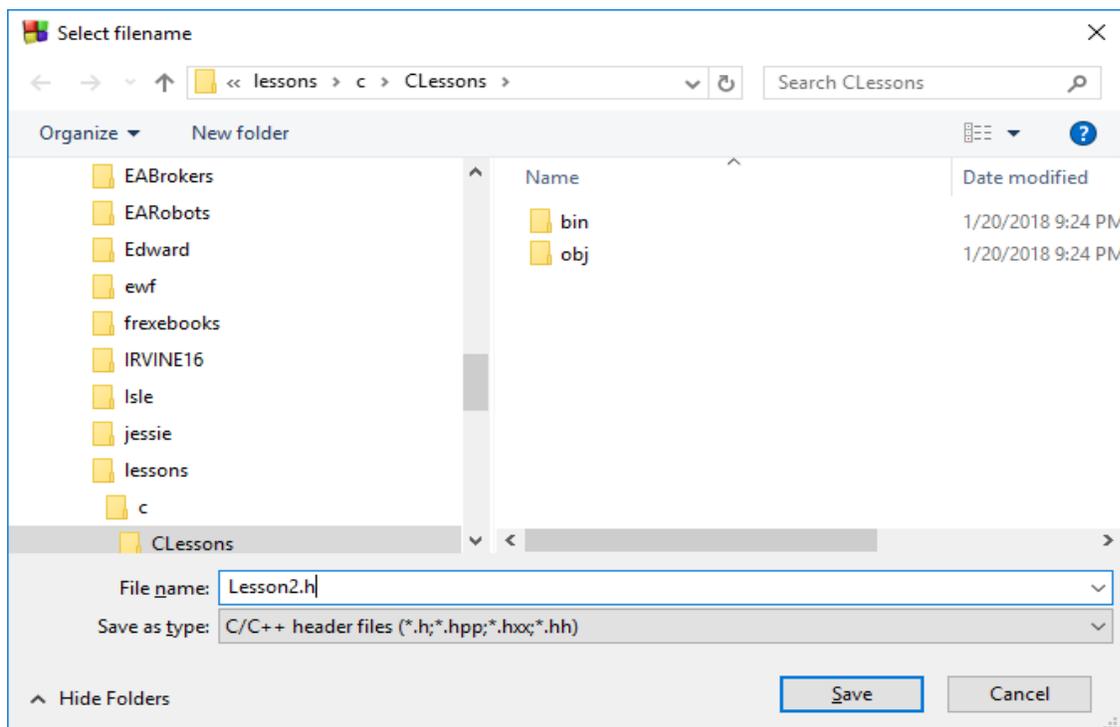
Press Next



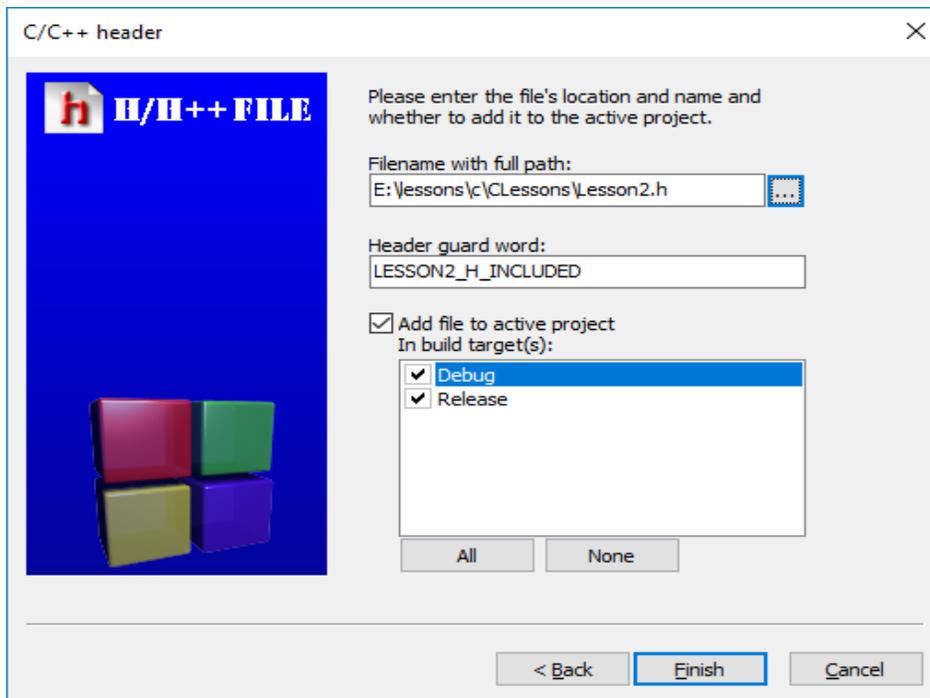
Make sure the Debug and Release check boxes are checked before proceeding. Select the filename browse button ... then type in Lesson2.h



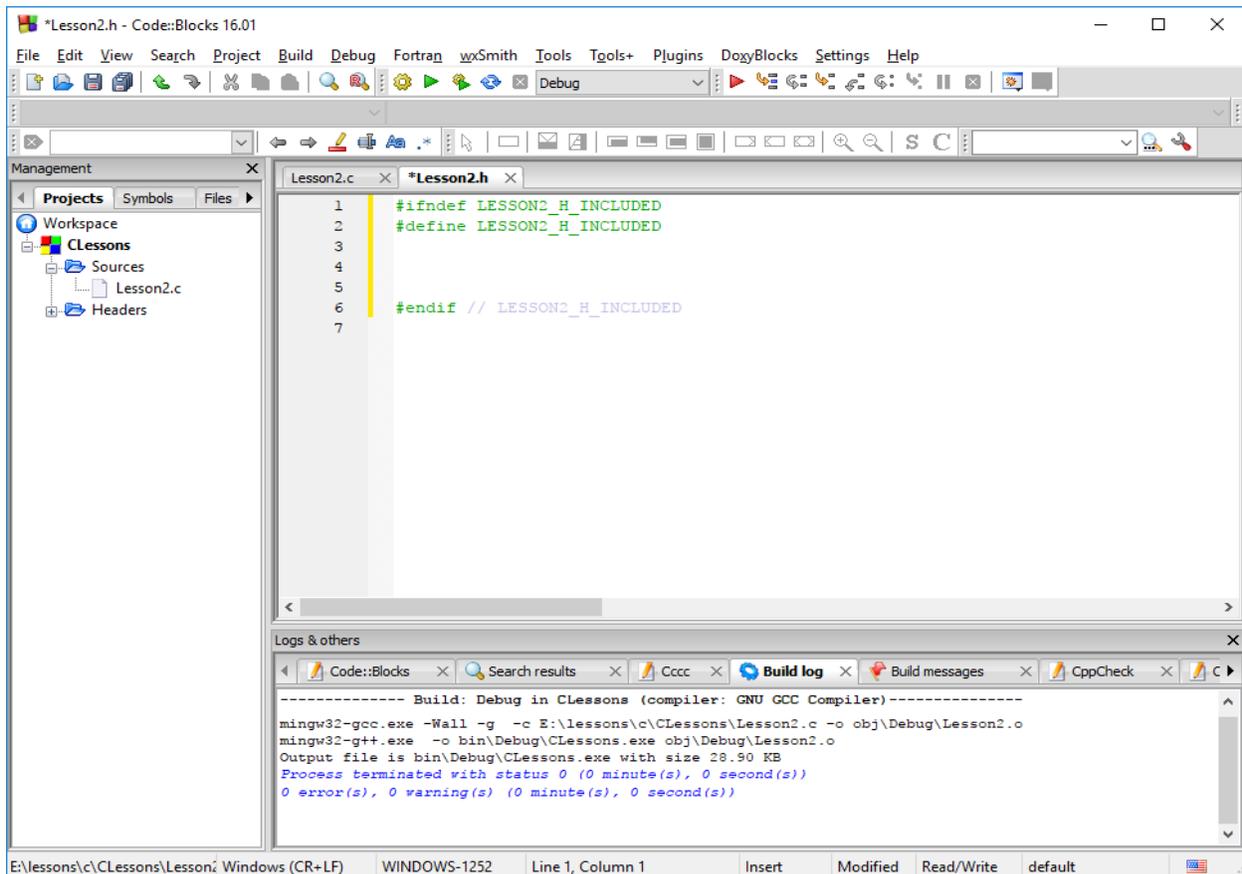
Press Next



Make sure Debug and Release are checked then Press Finish



You should now have something like this



```
#ifndef LESSON2_H_INCLUDED
#define LESSON2_H_INCLUDED
```

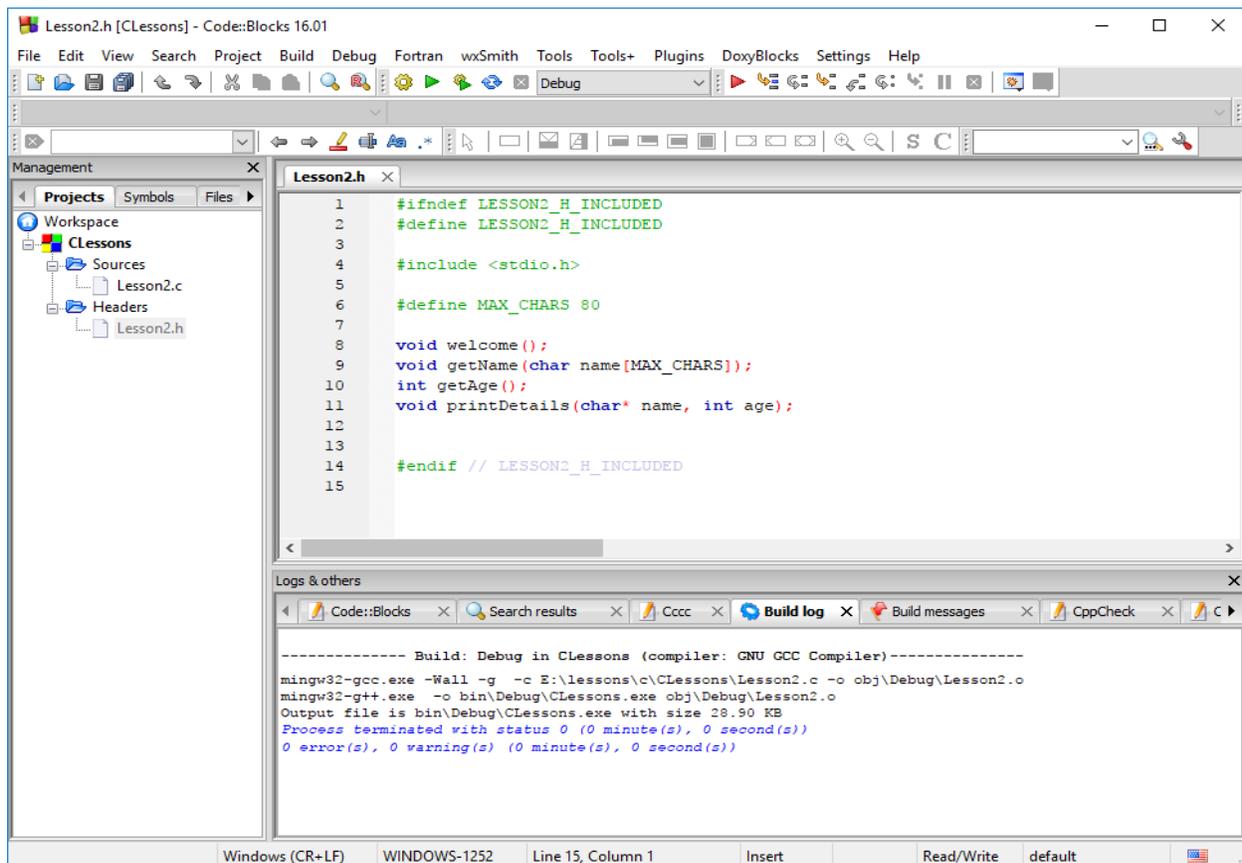
Are known as guards and allow the .h file only to be read once. Without the guards, the .h file may be read many times and resulting in duplicate function declaration error messages.

The guard ends with

```
#endif
```

Put the `#include<stdio.h>` preprocessor, and constant `#define MAX_CHARS 81` preprocessor and function prototypes from Lesson2.c into the Lesson2.h header file between the start and end guards.

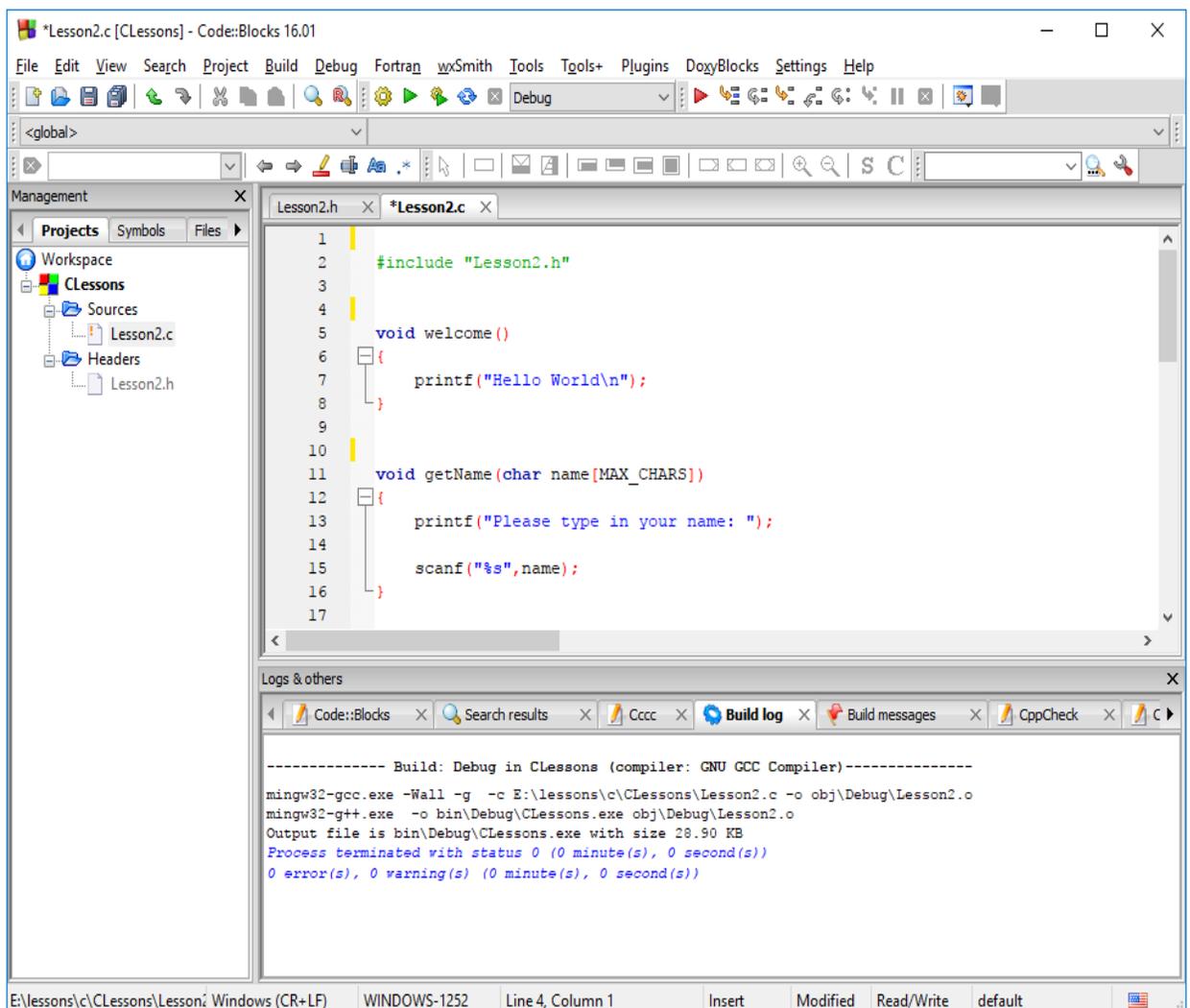
Your Lesson2.h should look like this after typing in the `#define` preprocessor constant and the function prototypes.



You now need to remove the include statements, the constant #define statement and function prototypes on the top of the Lesson2.c file since they are no longer needed. You need also to add an include statement on the top of the Lesson2.c file.

#include "Lesson2.h"

You need to do this so that the Lesson2.c file knows about the functions it will be using. The includes file statement allows the compiler to read the Lesson2.h file before compiling the rest of the Lesson2.c file. We use double quotes to specify the directory where our program resides. The triangle brackets <> specify to look for the include file in the compiler directory. You should have something like this



COMMENTS

All programs need to be commented so that the user knows what the program is about. Just by reading the comments in a program somebody will know exactly what the program is supposed to do. We have two types of comments in C. Header comments that are at the start of a program or a function. They start with `/*` and end with a `*/` and can span multiple lines like this.

```
/*  
Program to read a name and age from a user and  
print the details to the screen  
*/
```

Other comments are for one line only and explain what the current or proceeding program statement it is to do. The one-line comment starts with a `//` like this:

```
// function to read a name from the key board are return the value
```

Not all C compilers will recognize the one-line comments.

We now comment our program. Please add all these comments to your program.

```
/*  
Lesson2.c  
Program to read a name and age from a user and print  
the details on the screen  
*/  
  
#include "Lesson2.h"  
  
/* function to print welcome message */  
void welcome()  
{  
    printf("Hello World\n");  
}
```

```

/* function to obtain a name from the keyboard */
void enterName(char name[MAX_CHARS])
{
    printf("Please type in your name: ");
    scanf("%s",name);
}

/* function to obtain an age from keyboard */
int enterAge()
{
    int age;

    printf("How old are you? ");
    scanf("%d",&age);
    return age;
}

/* function to print name and age on screen */
void printPerson(char name[MAX_CHARS], int age)
{
    printf("Nice to meet you %s\n",name);
    printf("%s You are %d age years old\n", name, age);
}

int main()
{
    char name[MAX_CHARS];
    int age;

    welcome(); // welcome user
    enterName(name); // obtain a name
    age = enterAge(); // obtain an age
    printPerson(name, age); // print out name and age

    return 0;
}

```

Lesson2 Homework

Make a C program file called homework2.c that has a function to print a welcome message that describes what the program does. Has another a function called enterTitle() that asks someone what their profession title is, like doctor, lawyer etc. Has another function called enterSalary() that asks someone what their annual salary and returns a salary. Finally make a printProfession() function that prints out their title and salary..Print's out a message like this: "I am a Manager and I make \$100,000 dollars per year!". Call all the functions from the main function.

LESSON 3 STRUCTURES

Structures allow you to group different data type variables together under one common name. Structures start with the keyword **struct** followed by a name identifier, a curly bracket { the variable declarations and then closing curly bracket } ending with a semicolon. (Do not forget the semicolon.)

```
structure_name
{
declare_variables
};
```

We can use our **name** and **age** variables from previous lesson to make a Person structure. Our Person structure would look like this:

```
struct Person
{
char name[MAX_CHARS];
int age;
}; // do not forget the semicolon
```

Our Person structure actually becomes a new data type. We start our structure name with a capital letter to indicate it is a user data type.

Using a structure in your program

The structure is usually defined in the header .h file, but also can also be defined in the source .c file. In our case we define our structure in the header .h file so that other c files can use it.

Make a new header file called Lesson3.h and copy all the code from Lesson2.h into it. Also make a new C source file Lesson3.c and copy all the code from Lesson2.c into it. Make sure Debug and Release are checked before you press the Finish button or else they will not be compiled into the project. Remove Lesson2.h and Lesson2.c from the management window.

Put our Person structure definition in the Lesson3.h file just below the **#define MAX_CHARS 81** preprocessor statement. You also need to redefine the **printPerson** function to accept a Person structure rather than the name and age parameters

```
void printPerson(struct Person p);
```

You should then have something like this.

```
#ifndef LESSON3_H_INCLUDED  
#define LESSON3_H_INCLUDED  
  
#include <stdio.h>  
  
#define MAX_CHARS 80  
  
typedef struct person_type  
{  
char name[MAX_CHARS];  
int age;  
}Person;  
  
void welcome();  
void getName(char name[MAX_CHARS]);  
int getAge();  
void printPerson(Person p);  
  
#endif // LESSON3_H_INCLUDED
```

We now declare a Person structure in our Lesson3.c source file. Using a structure is a 3-step process, define a structure, declare a structure and then use the structure. You declare a structure just like a normal variable.

```
struct Person p;
```

Don't forget the **struct** keyword.

You can also initialize a structure when you declare them to a default value or some known values. To initialize to a default value we use {0} ending with a semi colon. All values in the structure would receive a the value 0.

```
struct Person p = {0}; // do not forget the semicolon
```

For older C compilers you may need to initialize each variable defined in the structure to default values separately.

```
struct Person p = {"",0};
```

To initialize the structure with known values we list the values enclosed in { } brackets.

```
struct Person p = {"Tom",24};
```

The structure variable p now has the values name "Tom" and age 24. It is important to distinguish between defining a structure and declare a structure. When you define a structure you list the variables that the structure will hold. When you declare a structure variable you are reserving memory for the structure to store values when the program runs. If you initialize the structure with values when you declare it, then the structure variable will already have these values when the program runs.

In Lesson3.c main function remove the name and age variables and replace them with the Person structure p.

```
struct Person p;
```

To access values in a structure variable you use the access **dot . operator**.

```
structure_variable_name . variable_name
```

To access name:

```
p.name
```

We would then call the getName function and pass the persons name from the person to it like this:

```
enterName(p.name)
```

To access age:

```
p.age
```

We would call the getAge function and assign the age to the person structure like this:

```
p.age = enterAge();
```

To pass a structure to a function you just pass the structure variable name to the function like this:

```
printPerson(p);
```

In our printPerson we will now have a struct Person parameter that will access the name and age values using the access dot . operator as follows:

```
void printPerson(struct Person p)  
{  
    printf("Nice to meet you %s\n",p.name);  
    printf("%s You are %d age years old\n", p.name, p.age);  
}
```

Update Lesson3.c main function to use our Person structure. You should have something like this:

```
/*  
Lesson3.c  
Program to read a name and age from a user using a structure  
and print the details on the screen  
*/
```

```

#include "Lesson3.h"

/* function to print welcome message */
void welcome()
{
    printf("I like C programming\n");
}

/* function to obtain a name from the keyboard */
void enterName(char name[MAX_CHARS])
{
    printf("Please type in your name: ");
    scanf("%s",name);
}

/* function to obtain an age from keyboard */
int enterAge()
{
    int age;
    printf("How old are you? ");
    scanf("%d",&age);
    return age;
}

/* function to print name and age on screen */
void printPerson(Person p)
{
    printf("Nice to meet you %s\n",p.name);
    printf("%s You are %d age years old\n", p.name,.p.age);
}

```

```

int main()
{

    /* welcome user */
    welcome();

    /* obtain a name */
    enterName(s.p.name);

    /* obtain an age */
    p.age = getAge();

    /* print out name and age */
    printPerson(p);

    return 0;
}

```

Build and run the program, type Tom for name and 24 for age. You will get the following output.

```

Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old

```

Do not proceed until you got your program working.

Using typedef

Typing **struct** all the time is a lot of work to do, to make programming life easy, **typedef** allows you to use your structure without the **struct** keyword. Typedef actually means type definition, that allows you to define your own data types from C data types. The syntax is:

```
typedef data_type user_data_type_identifier;
```

Our structure definition would now look like this:

```
typedef struct person_type  
{  
char name[MAX_CHARS];  
int age;  
}Person;
```

The structure still has a name but a different name **person_type**. We use lower case and an underscore for the structure name since it is the C convention. The type definition name is Person because this is the data type name we want to use in our program. In the Lesson3.h change the Person structure definition to use a typedef, then remove all the struct key words in the Lesson3.h and Lesson3.c files.

Build and run your program and see it is still working. Do not proceed until you got your program working.

HomeWork 3 Part 1

Convert your homework2 program to use a structure called **Profession**. The Profession structure would have a variable to store the profession **title** like doctor or manager and a variable to store their **salary**. Your Profession structure would look like this:

```
struct Profession  
{  
char title[MAX_CHARS];  
salary age;  
};
```

You may use **typedef** instead if you wish.

Make an enterDetails function that receives a pointer to a profession structure. The enterDetails function would ask for the profession type and profession salary and store in the profession structure passed to it. Make an enterDetails function that receives a profession structure that will be used to print out the profession type and salary. Include the welcome function from homework 2. Call your c program homework3.c

Structure inside a Structure

A Structure may include another structure. We can make a Student structure that contains the following Person structure for its name and age.

```
struct Person
{
  char name[MAX_CHARS];
  int age;
};
```

A Student structure will have an additional variable called **studentNumber** that will represent a char string student id number.

Our Student structure would look like this:

```
struct Student
{
  Person p;
  char studentNum[MAX_CHARS];
};
```

Notice our Student structure has a Person structure inside it, that will be used to store the students name and age.

We would declare a Student structure variable like this:

```
Student s;
```

We still use the access dot operator to access the student and person structure.

We would access the idnum id the Student structure like this:

```
s.studentNum
```

We would access the Person structure in the student structure like this:

```
s.p.age  
s.p.name
```

You can also initialize a structure inside a structure when you declare them to a default value or some known values. To initialize to a default value we use {0} ending with a semi colon. All values in the structure would receive a the value 0.

```
struct Student s = {0};
```

For older C compilers you may need to initialize each variable defined in the structure to default values separately.

```
struct Student s = {"",0,""};
```

To initialize the structure in a structure with known values we list the values enclosed in { } brackets for each structure,

```
struct Student s = {"Tom",24,"S1234"};
```

The structure s now has the values name "Tom" and age 24 for the Person structure and "S1234" for the rest of the Student structure. It is important to distinguish between defining a structure and declare a structure. When you define a structure you list the variables that the structure will hold. When you declare a structure variable you are allocating memory for the structure when the program runs. If you initialize the structure with values when you declare it then the structure variable will store the values.

The **printStudent** function would receive a Student structure print out details of a Student.

```
void printStudent(struct Student s)  
{  
  printPerson(s.p);  
  printf("Your Student number is %s\n",s.id);  
}
```

Notice we call the `printPerson` function inside the `printStudent` function. The `printPerson` function receives a person structure from the student structure.

to-do

Update the `Lesson3.h` header file to use a `Student` structure. You would also need to make a `printStudent` function prototype that receives a `Student` structure. You will also need an additional `enterStudentNum` function prototype to enter the student id number.

In the `Lesson3.c` code file make a `enterStudentNum` function to enter and return a student's id number. Make a `printStudent` function to accept a student structure to print the person details and student id number. Inside the `printStudent` function call the `printPerson` function to print out the person details and then print out the student id number. You should have something like this:

```
#ifndef LESSON3_H_INCLUDED
#define LESSON3_H_INCLUDED

#include <stdio.h>

#define MAX_CHARS 81

struct person_type{
char name[MAX_CHARS];
int age;
}Person;

struct student_type{
Person p;
char idnum[MAX_CHARS];
}Student;

void welcome();
void enterName(char name[MAX_CHARS]);
int enterAge();
void enterStudentNum(char studentNum[MAX_CHARS]);
void printPerson(Person p);
void printStudent(Student s);

#endif // LESSON3_H_INCLUDED
```

Inside the main function add a Student structure variable called **s**. Call functions enterName, enterAge , enterStudentNum to populate the student structure. Then call function printStudent() to print out the student details.

You should have something like this:

```
/*
Lesson3.c
Program to read a name, age and student number from a user using a structure
and to print the details on the screen
*/
#include "Lesson3.h"

/* function to print welcome message */
void welcome()
{
    printf("I like C Programming\n");
}

/* function to obtain a name from the keyboard */
void enterName(char name[MAX_CHARS])
{
    printf("Please type in your name: ");
    scanf("%s",name);
}

/* function to obtain an age from keyboard */
int enterAge()
{
    int age;
    printf("How old are you? ");
    scanf("%d",&age);
    return age;
}

// get student number
void enterStudentNum(char studentNum[MAX_CHARS])
{
    printf("Please type in your student number: ");
    scanf("%s",studentNum);
}
```

```

/* function to print name and age on screen */
void printPerson(Person p)
{
    printf("Nice to meet you %s\n",p.name);
    printf("%s You are %d age years old\n", p.name,.p.age);
}

/* function to print name and age on screen */
void printStudent(Student s)
{
    printf("Nice to meet you %s\n",s.p.name);
    printf("%s You are %d age years old\n", s.p.name, s.p.age);
    printf("Your student ID number is %s\n", s.idnum);
}

int main()
{
    /* make a student */
    Student s;

    /* welcome user */
    welcome();

    /* obtain a name */
    getName(s.p.name);

    /* obtain an age */
    s.p.age = getAge();

    /* obtain student idnum */
    getStudentNum(s.studentNum);

    /* print out name and age */
    printStudent(s);

    return 0;
}

```

Compile your program and correct any errors. Run you program with student id number with S1234. You should get something like this.

```
E:\lessons\c\CLessons\bin\Debug\CLessons.exe
Hello World
Please type in your name: Tom
How old are you? 24
Please type in your student ID number: S1234
Nice to meet you Tom
Tom You are 24 age years old
Your student ID number is S1234

Process returned 0 (0x0)   execution time : 10.892 s
Press any key to continue.
```

If you got this far then you should congratulate your self.

Homework 3 Part 2

Make a **JobDescription** structure to store the details of a Profession. An example would be a RealEstateAgent description would be “I sell houses”. Put the previous homework **Profession** structure inside your **JobDescription** structure. You should have something like this.

```
struct JobDescription {
    Profession p;
    char description[MAX_CHARS];
};
```

You may use **typedef** instead if you wish.

The **enterJobDescription** function would ask for the job description to get a job description from the key board. Make an **printJobDescription** function that receives a **JobDescription** structure that will be used to print out the profession type and salary and job description. Use the **enterTitle**, **enterSalary**, **printProfession** and **welcome** functions from the previous homework. Put all the updated code in the same homework3.c program file.

Lesson 4 Operators

Operators

Operators do operations on variables like addition + , subtraction – and comparisons > etc. We now present all the C operators with examples. Make a new C file called Lesson4.c. In your Lesson4.c in the main function type out the examples and use printf statements to print out the results. You can type in the operation right inside the printf statement just like this

```
printf("%d\n",(3+2));
```

or

```
printf("%d\n",(3>2));
```

Alternatively, you can use variables instead.

```
int x = 3;  
int y = 2;  
printf("%d + %d = %d \n", x, y, x+y);
```

unary operators

The - unary operators change the sign of a number, where as the + unary operator does not, it just confirms the present sign of the number.

```
int x = 5
```

Operator	Description	Example	Result
+	confirm positive number	+5	5
-	Negate positive number	-5	-5
+	confirm negative number	+ -5	-5
-	Negate negative number	--5	5

```
printf("-5 =%d\n", -5);
printf("+5 = %d\n", +5);
printf("+-5 =%d\n", +-5);
printf("--5 = %d\n", --5);
```

Arithmetic Operators

Arithmetic operators are used to do operations on numbers like addition and subtraction.

```
int x = 5;
```

```
int y = 2;
```

```
printf("%d + %d = %d \n",x ,y, x+y); // would print out 5 + 2 = 7
```

```
printf("%d %% %d = %d \n",x ,y, x+y); // would print out 5 % 2 = 7
```

Operator	Description	Example	Result
+	Add two operands	5 + 2	7
-	Subtract right operand from the left	5 - 2	-3
*	Multiply two operands	5 * 2	6
/	Divide left operand by the right one	5 / 2	2
%	Modulus - remainder of the division of left operand by the right	5 % 2	3

Comparison Operators (conditions)

Comparison operators are used to compare values. It either evaluates to a 1 meaning **true** or 0 meaning **false** according to the condition. A Comparison operator and values are known as a **condition**.

```
x = 5;
```

```
y = 3;
```

```
printf("%d > %d = %d \n",x,y,x>y); // would print out 5 > 2 = 1 (true)
```

Operator	Description	Example	Result
>	Greater than - true if left operand is greater than the right	5 > 3	true (1)
<	Less than - true if left operand is less than the right	3 < 5	true (1)
==	Equal to - true if both operands are equal	5 == 5	true (1)
!=	Not equal to - true if operands are not equal	5 != 5	true (1)
>=	Greater than or equal to - true if left operand is greater than or equal to the right	5 >= 3	true (1)
<=	Less than or equal to - true if left operand is less than or equal to the right	5 <= 3	true (1)

Logical Operators

Logical operators are the **and**, **or**, **not** boolean operators, where the result of the operation is true and/or false. Where:

1 = true

0 = false

x=1;

y=0;

printf("%d && %d = %d \n",x,y,x&&y);// // would print out 1 && 1 = 1 (true)

printf("%d || %d = %d \n",x,y,x||y);// // would print out 1 && 1 = 1 (true)

printf(!%d = %d \n",y,!y);// // would print out !0 = 1 (true)

Operator	Description	Example	Result
&&	true if both the operands are true	1 && 1	1
	true if either of the operands is true	1 0	1
!	true if operand is false (complements the operand)	!0	1

Compound conditions

Logical operators are combined with conditional operators to form compound conditions that are more powerful.

Condition logical operator condition
(5 > 3) && (3 < 5)

```
x = 5;  
y = 3;  
printf("%d > %d && %d < %d=%d\n", x,y,x > y && x < y);
```

```
5 > 3 && 3 < 5 = 1
```

result = 1 meaning true

todo

make another compound condition using the || comparison operator

Binary Numbers

All numbers in a computer are stored as binary numbers. Binary numbers (base 2) just has 2 digits 0 and 1 whereas decimal numbers have 10 digits 0 to 9. We also have hexadecimal (base 16) numbers 0 to F that represent decimal numbers 0 to 15. We use the letters A to F to represent decimal numbers 10 to 15. Here are the binary and hexadecimal numbers for decimal numbers 0 to 15.

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3

4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Bitwise Operators

Bitwise operators act on operands as if they were binary digits. It operates bit by bit. Binary numbers are base 2 and contain only 0 and 1's. Every decimal number has a binary equivalent. Every binary number has a decimal equivalent. For example, decimal 2 is 0010 in binary and decimal 7 is binary 0111.

In the table below: 10 = (0000 1010 in binary) 4 = (0000 0100 in binary)

`printf("%d | %d \n",x,y 10 | 4); // would print out 10 | 4 = 7`

The difference between comparison operators and bit wise operators, the bit wise operators change the value where as the comparison operators do not. They just compare.

Operator	Description	Example	Result
&	Bitwise AND	10 & 4	0 (0000 0000 in binary)
	Bitwise OR	10 4	7 (0000 1110 in binary)
^	Bitwise XOR	10 ^ 4	7 (0000 1110 in binary)
~	Bitwise complement	~10	-11 (1111 0101 in binary)

You may want to use variables values like 0 and 1 instead like this:

x = 0

y = 1

printf("%d & %d \n",x & y); // x & y = 0

using 0 and 1's rather than numbers make the bitwise operations easier to understand:

and	or	xor
0 & 0 = 0	0 0 = 0	0 ^ 0 = 0
0 & 1 = 0	0 1 = 1	0 ^ 1 = 1
1 & 0 = 0	1 0 = 1	1 ^ 0 = 1
1 & 1 = 1	1 1 = 1	1 ^ 1 = 0

The ~ operator reverse the bits. 0 becomes 1 and 1 becomes -

10 = 0000 1010
~ 1111 0101

Negative binary numbers have a 1 at the start known as the msb (most significant bit)

1111 0101 is actually =-11

You can use 2's complement to convert a positive binary number to a negative binary number or a negative binary number to a positive binary number.

	0000 1011	1111 0101
Step 1 complement binary number	1111 0100	0000 1010
Step 2 add	1	1
	-----	-----
	1111 0101 (-11)	1111 1011 (11)

Shift Operators

Shift operator allow you to multiply or divide a variable by multiple of 2. The << shift operator multiplies by powers of 2 by shifting bits left. The >> shift operator divides by number of powers of 2 by shifting bits right. We left and right shift the number 10 by powers of 2. Binary 10 is 00001010

```
x = 2
```

```
y = 3
```

```
x = x << y; // 2 << 3 = 8
```

```
printf("x << 3 = %d\n,x);
```

```
x << y = 8
```

```
x = x >> 3; // 8 >> 3 = 2
```

```
printf("x << 3 = %d\n,x);
```

```
x >> y = 2
```

Operator	Description	Example	Result
<<	Shift bits left by 3 bits (multiply by $2^3 = 8$)	$2 << 3$ ($2 * 8$)	8 ($2 * 2 * 2$)
>>	Shift bits right by 2 bits (divide by $2^2 = 4$)	$8 >> 2$ 2	2 ($8/2 /2 /2$)

Increment/Decrement Operators ++ --

Increment operators ++ increment a variable value by 1 and decrement operators -- decrement a value by 1.

They come in two versions , **prefix** increment/decrement value **before** or **postfix** increment/decrement value **after**.

Prefix Increment before `y = ++x`

x is incremented then value of y is assigned the value of x

	x	y
	-----	-----
x = 5	5	?
y = ++x	6	6

`x = 5`

`y = ++x;`

`printf("y=x ++ y=%d x = %d\n",y,x);`

y=++x y=6 x = 6

postfix increment after `y = x++`

The value of y is assigned the value of x and then x is incremented

	x	y
	-----	-----
x = 5	5	?
y = x++	6	5

`x = 5`

`y = ++x;`

`printf("y=++x y=%d x = %d\n",y,x);`

y=++x y=6 x = 6

y does not increment but x increments

prefix Decrement before $y = --x$

x is decremented then value of y is assigned the value of x

	x	y
	-----	-----
x = 5	5	?
y = ++x	4	4

x = 5

y = --x;

printf("y=--x y=%d x = %d\n",y,x);

y=--x y=4 x = 4

both y and x decrement

postfix decrement after $y = x--$

The value of y is assigned the value of x and then x is decremented

	x	y
	-----	-----
x = 5	5	?
y = x--	4	5

x = 5

y = x--;

printf("y=x -- y=%d x = %d\n",y,x);

y=x-- y=5 x = 4

y does not decrement but x decrements

Increment decrement operators are usually used stand alone to increment or decrement a variable value by 1.

```
x++
```

```
x--
```

Assignment Operators

Assignment operators are used to assign values to variables. `x = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `x` on the left. There are various compound operators in like `x += 5` that adds to the variable and later assigns the same. It is equivalent to `x = x + 5`.

```
x = 5;  
printf("x = %d\n",x);
```

```
x+= 5;  
printf("x += 5 = %d\n",x); // x += 5 = 10
```

Operator	Compound	Equivalent	Operation
=	<code>x = 5</code>	<code>x = 5</code>	Assign 5 to x
+=	<code>x += 5</code>	<code>x = x + 5</code>	Add 5 to x
-=	<code>x -= 5</code>	<code>x = x - 5</code>	Subtract 5 from x
*=	<code>x *= 5</code>	<code>x = x * 5</code>	Multiply x by 5
/=	<code>x /= 5</code>	<code>x = x / 5</code>	Divide x by 5
%=	<code>x %= 5</code>	<code>x = x % 5</code>	MOD x by 5
<<=	<code>x <<= 5</code>	<code>x = x << 5</code>	Shift x left 5 bits (multiply by 2 (32))
>>=	<code>x >>= 5</code>	<code>x = x >> 5</code>	Shift x right 5 bits (divide by 2^5 (32))
&=	<code>x &= 5</code>	<code>x = x & 5</code>	AND x by 5
=	<code>x = 5</code>	<code>x = x 5</code>	OR x by 5
^=	<code>x ^= 5</code>	<code>x = x ^ 5</code>	XOR x by 5

Lesson4 Homework Part 1

1. Print out if a number is even, using just a print statement and a arithmetic operator
2. Print out if a number is odd, using just a print statement and a arithmetic operator
3. Swap 2 number using a temporary variable
4. Multiply a number by 8 using a shift operator
5. Divide a number by 8 using a shift operator
6. In a print statement, add 2 numbers together and check if they are less than multiplying them together
7. In a print statement, add 2 numbers together and check if they are less than multiplying them together **and** greater then multiplying them together.
8. In a print statement, add 2 numbers together and check if they are less than multiplying them together **or** greater then multiplying them together

Put all your homework in a file called homework4.c

Character String Operations

In C operations on character strings are carried out by built in functions. To use these built in functions you must place

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

On the top of your Lesson4.c file.

Character strings in C are also known as CStrings. You declare CStrings with **char** data type and square [] brackets. The [] brackets means the char variable holds 1 one more character.

```
// declare and assign character string  
char s1[] = "hello";
```

```
// print out string  
printf("%s\n",s1); // hello
```

You do not need to specify the number of characters in the string if you initialize with a string of letters. If you do, you always need to specify the number of letters + 1, because you must allow 1 extra character to hold the end of string terminator that is a '\0' or just a 0.

```
char s1[6] = "hello";
```

```
// get a character from string  
char c = s1[0];  
printf("%c\n", c) // h
```

```
// change a character in a string  
s1[0] = 'j';
```

```
// print out string  
printf("%s\n",s1); // jello
```

making empty strings

If you make a empty string, you must specify the number of characters you want. You must reserve an extra character for the end of string character '\0'.

```
char s2[81] = {0};  
char s2[81] = ""; // alternate empty string
```

Once you got a character string you can always make it Empty by setting the first character to the end of string character '\0'.

```
s2[0] = '\0'; // the more professional way  
or  
s2[0] = 0; // the lazy way
```

It is probably better to do the more professional way using the end of string character '\0' rather than 0, but they are both the same value of 0.

string functions

There are many string functions. Here are just a few of them:

```
// get length of a string  
x = strlen(s1)  
printf("The length of the string is %d\n",x); // 5
```

```
// copy a string  
strcpy(s2,"goodbye");  
printf("%s\n", "goodbye); // goodbye
```

```
// join two strings together  
strcat(s2,s1);  
printf("%s\n", s2); // goodbyehello
```

```

// test if 2 strings are less greater or equal
// -1 = less 0 = equal 1 = greater
printf("%d\n",strcmp(s1,s2)); // -1
printf("%d",strcmp(s1,s1)); // 0
printf("%d",strcmp(s2,s1)); // 1

// get pointer to start of a sub string
char* pchr = strstr(s2,"hello");
printf("%s",pchr); // hello

```

Lesson4 Homework Part 2

9. Make a string of your favourite word and replace the first letter with another letter, hint use substring.

Example : change "hello" to "jello"

10. Make a string of your favourite word and replace the last letter with another letter, hint use strlen

Example : change "jello" to "jelly"

11. Make a string of your favourite word and change the middle letter, hint use substring.

Example : change "jelly" to "jexly"

12. Replace the last letter with the first letter in a word

Example : change "jely" to "yelj"

13. Compare if the two above strings are equal, greater or smaller to each other.

14. Use **strstr** to point to substrings in to 2 different strings. Copy the first one to an empty string ,then next concatenate a string of your choice and finally concatenate the second string to the end.

Put all your homework in a file called homework4.c

LESSON 5 PROGRAMMING STATEMENTS

Programming statements allow you to write complete C programs. We have already looked at simple input, print and assignment statements. We now present you with branch and loop programming statements. Continue with the C file Lesson5.c and try out all these branch loop statements one by one. Once you see the program execution you will understand how these branch and loop statements work. You may also want to add some extra statements of your own.

Branch Control Statements

Branch control statements allow certain program statements to execute and other not.

if statement

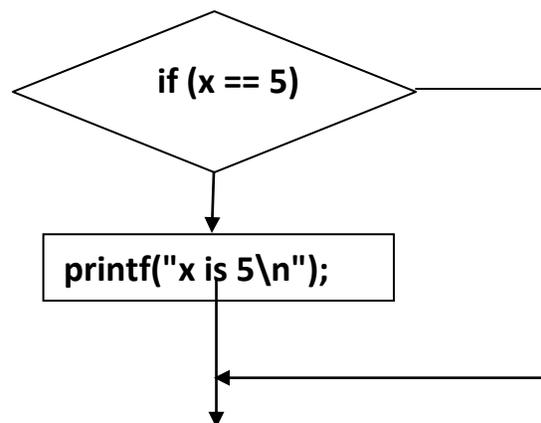
The **if** branch **control** statements contain a condition using conditional operators from the previous lesson to direct program flow.

```
If (condition)  
Statement(s)
```

When the condition is evaluated to be **true** the statements belonging to the if statement execute. An if statement is a one-way branch operation.

```
// if statement  
x = 5;  
if (x == 5)  
{  
    printf("x is 5\n");  
}
```

x is 5



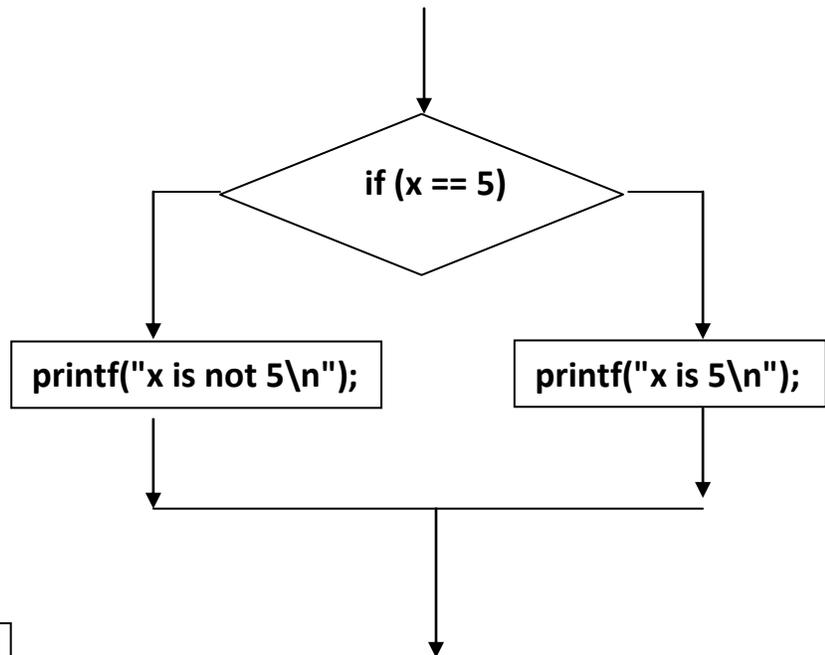
If – else statement

We now add an else statement. An if-else control construct is a two-way branch operation.

```
If (condition)  
statements  
else  
statements
```

```
// if – else statement  
x = 2;  
if (x == 5)  
    printf("%s\n", "x is 5");  
else  
    printf("%s\n", "x is not 5");
```

```
x is not 5
```

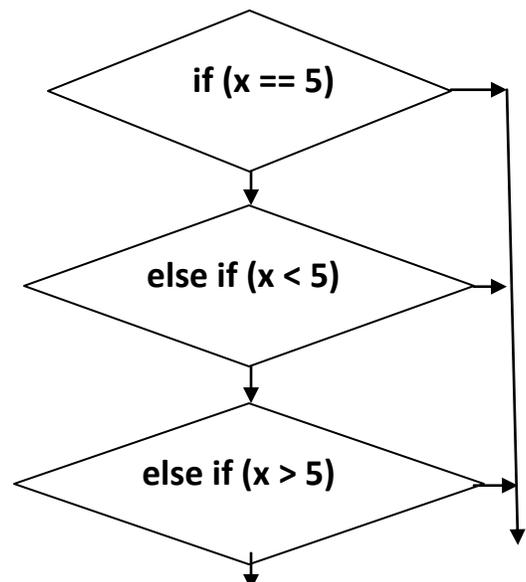


Multi if-else statement

We can also have additional else if statements to make a multi-branch.

```
// multi if else  
x = 10;  
if (x == 5)  
    printf("x is 5\n");  
else if (x < 5)  
    printf("x less than 5\n");  
else if (x > 5)  
    printf("x greater than 5\n");
```

```
x greater than 5
```



A multi branch if-else can also end with an else statement.

```
// multi if-else else
x = 5;
if (x < 5)
    printf("x less than 5\n");
else if (x > 5)
    printf("x greater than 5\n");
else
    printf("x is 5\n");
```

x is 5

switch statement

A switch statement is considered an organized if-else statement. It is a little limited since it can only handle numeric equals. When the case values matched the switch value the statements in the case execute. The **break** keyword exits the switch statement. The default statement is executed if there is no match.

// switch statement

```
x = 2;
switch(x)
{
    case 1:
        printf("%s\n", "x is 1");
        break;
    case 2:
        printf("%s\n", "x is 2");
        break;
    case 3:
        printf("%s\n", "x is 3");
        break;
    default:
        printf("%s %d\n", "x is ", x);
        break;
}
```

x is 2

nested if-else statement

if statements can also be nested to make complicated conditions simpler.

```
// nested if statement
x = 5;

if (x >= 0)
{
    if (x > 5)
        printf("%s\n", "x greater than 5");
    else
        printf("%s\n", "x less than equal 5");
}
```

x less than equal 5

Loop Control Statements

Loop control statements allow program statements to repeat themselves.

while loop

The while loop allows you to repeat programming statements repeatedly until some condition is satisfied.

The while loop requires an initialized counter, a condition, program statements and then increment or decrement a counter.

```
Initialize counter
while condition:
    statement(s)
    increment/decrement counter
```

When the condition is false the loop execution exits. While loops are used when you do not know how many items you have.

Here is a while loop that prints out the number 0 to 4

```
// while loop
x = 0;
while (x < 5)
{
printf("%d\n",x);
    x++;
}

printf("\n"); // new line
```

```
0
1
2
3
4
```

Todo

Change the above while loop to print out 1 to 5

Make a while loop that prints out the numbers 1 to 5 backwards.

do loop

The do loop also known as a **do-while** loop allows you to repeat programming statements repeatedly until some condition is satisfied. The condition is at the end of the loop, so the programming statements execute at least once.

The do loop requires an initialized counter, program statements, increment or decrement a counter and finally a condition.

```
Initialize counter
do{
    statement(s)
    increment/decrement counter
} while condition;
```

When the condition is false the loop execution exits. **do loops** are used when you do not know how many items you have.

Here is a **do loop** you can try out that prints out the number 0 to 4.

```
// do loop
x = 0;
do
{
    printf("%d\n",x);
    x++;
} while (x < 5);
```

```
0
1
2
3
4
```

```
printf("\n"); // new line
```

Todo

Change the above **do while** loop to print out 1 to 5

Make a do while loop that prints out the numbers 1 to 5 backwards.

for loop

Another loop is the **for** loop. It is much more automatic than the while loop but more difficult to use. All loops must have a counter mechanism. The for loop needs a start count value, condition, increment/decrement counter. When the condition is false the loop exits. For loops are used when you know how many items you have.

```
for (start_count_value,condition, increment/decrement_counter):
    Statement(s)
```

Here we have a for loop to print out values 0 to 4, to try out.

```
// for loop
int i;
for (i=0;i<5;i++)
{
```

```
0
1
2
3
4
```

```
    printf("%d\n",i);  
}
```

To do:

Change the above **for loop** to print out 1 to 5

Make a **for loop** that prints out the numbers 1 to 5 backwards.

Nested for loops

Nested for loops are used to print out 2 dimensional grids by row and column.

```
// nested for loop  
int r;  
int c;  
for (r=0;r<5;r++)  
{  
    printf("%d:");  
    for (c =0; c < 5; c++)  
    {  
        printf("%d",c);  
    }  
    printf("\n");  
}
```

```
1 : 1 2 3 4 5  
2 : 1 2 3 4 5  
3 : 1 2 3 4 5  
4 : 1 2 3 4 5  
5 : 1 2 3 4 5
```

Loops can also be used to print out characters in a string variable

```
// print out characters in a string  
char* s = "Hello";  
int i;  
for (i=0;i<strlen(s);i++)  
{  
    printf("%c\n",s[i]);  
}  
printf("\n");
```

```
H  
e  
l  
l  
o
```

LESSON 5 HOMEWORK TO DO:

Exam Grader

Ask someone to enter an exam mark between 0 and 100. If they enter 90 or above printout an "A", 80 or above print out a "B", 70 or above print out a "C", 60 or above print out a "D" and "F" if below 60. Hint: use if else statements.

You can visualize a grade chart like this:

Mark Range	Exam Grade
90 to 100	A
80 to 89	B
70 to 79	C
60 to 69	D
0 to 59	F

Mini Calculator

Make a mini calculator that takes two numbers and a operation like -, +, * and /. Prompt to enter two number's and a operation like this:

Enter first number: 3

Enter second number: 4

Enter (+, -, *, /) operation: +

Then print out the answer like this:

3 + 4 = 7

Hint: use a switch statement.

Use a while or do while loop so that they can repeatedly enter many calculations.
Terminate the program when they enter a letter like 'X' for the first number.

Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
      *
     * *
    * * *
   * * * *
  * * * * *
```

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Enhanced Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
      *
     * * *
    * * * * *
   * * * * * * *
  * * * * * * * *
 * * * * * * * * *
```

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Reverse a String

Reverse a String using a **while** loop or a **for** loop in place. Print the string before and after reversal also using a loop.

Test if a number is prime

Make a function called **isPrime(x)** that tests if a number is prime. In a loop divide the number between 2 to number-1 (or 2 to square root of number+1. For square root use:

```
x = (int)Math.sqrt(n);
```

If the number can be divided by any of the divisors then the number is not prime, else it is prime. Print out the first 100 prime numbers.

The first 10 prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29

Print out all factors of a number

Make a function call **factors(x)** that will print out all the factors of a number. The factors of a number is all the divisors divided by the number evenly.

Example:

The Factors of 50 are:

```
1  
2  
5  
10  
25  
50
```

Print out all prime factors of a number

Make a function call **prime_factors(x)** that will print out all the prime factors of a number. The prime factors of a number is all the prime number divisors divided by the number evenly.

Example: $12 = 2 \times 2 \times 3$

Following are the steps to find all prime factors.

- 0) Enter a number n
- 1) While n is divisible by 2, print 2 and integer divide n by 2
- 2) In a **for** loop from i = 3 to square root of n + 1 increment by 2
in a **while** loop while n is divisible by i
print i
integer divide n by integer i
- 3) print n if it is greater than 2.

For square root use:

```
x = Math.sqrt(n);
```

Make a Guessing game

Ask the user of your game to guess a number between 1 and 100. If they guess too high tell them "Too High". If they guess too low tell them they guess "Too Low". If they guess correct tell them "Congratulations you are Correct!". Play 10 games as a round. Keep track in an array how many tries each game took. At the end of 10 games in a table print out the tries for each game in the round. At the end of the table print out total score of all the game tries. For each round keep track of the lowest total score and inform the user if they beat the current lowest score or not. At the end of each round ask the user if they want to play another round of 10 games. You will need to first generate a random number to guess. You can use this code to generate a random number:

```
// seed random number generator  
srand((unsigned int)time(0));
```

```
// generate a random number  
int number = rand() % MAX_NUMBER + 1;
```

Where **MAX_NUMBER** is a constant placed at the top of your program.

```
const int MAX_NUMBER = 10;
```

Also make another constant **MAX_GAMES** for the number of games to play.

```
const int MAX_GAMES = 10;
```

You will need to include the following at the top of your program, for the compiler to recognize the **srand()**, **rand()** and **time()** functions.

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

You should have functions to print a welcome message explaining how to play the game, generate a random number, get a guess from the keyboard, check if a guess is correct and print out the game scores. The main function should just call your functions in a loop. Call your c file Homework5.c or GuessingGame.c

Guessing Game using a Structure

Make a Guess Game structure that will keep track of the guess number and tries per game. The main function would update the structure per round. After all games have been played print out the average game. Call your c file Homework5b.c or GuessingGame2.c

LESSON 6 ARRAYS

For this lesson make a new C source file called Lesson6.c and in the main function type in the following programming statements for the following Arrays

ARRAYS

Arrays are sequential values accessed under a common name. Arrays store many sequential values together. We have one dimensional arrays and multi dimensional arrays. One dimensional arrays are considered a single row of values having multiple columns. You can visualize a one-dimensional array as follows.

Value1	Value2	Value3	Value4	Value5
--------	--------	--------	--------	--------

We declare and initialize 1 dimensional int array of size 5 as follows. The size of the array is enclosed inside the square brackets.

```
int a[5] = {1,2,3,4,5};
```

1	2	3	4	5
---	---	---	---	---

When initializing all the values of an array the size is optional and can be written as follows:

```
int a[] = {1,2,3,4,5};
```

In this situation the size of the array is determined by the number of listed initialized values.

You can also declare a one-dimensional array of a specified size without initializing the values. Here we declare an array of size 5.

```
int a2[5];
```

↑
Number
of elements

You can initialize all array values to a single value like this:

```
int a2[5] = {0};
```

In this situation you must specify the size of the array you need also to assign array values separately as follows. Arrays locations are assigned by an index. All indexes start at 0.

```
a2[0] = 1;  
a2[1] = 2;  
a2[2] = 3;  
a2[3] = 4;  
a2[4] = 5;
```

0	1	2	3	4
1	2	3	4	5

The indexes are at the top and the array values are at the bottom

Arrays locations are also retrieved by an index

```
int x = a[0];  
printf("%d\n", x);
```

todo:

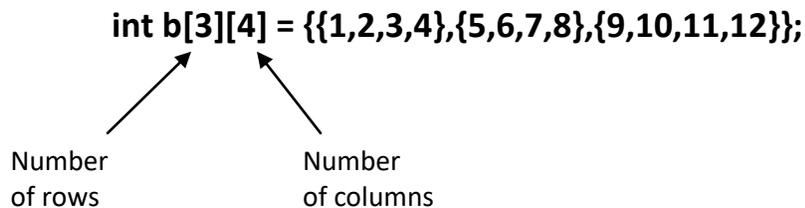
Print out all the elements in the above 1 dimensional array using a for loop

Two-dimensional arrays

Two-dimensional arrays have grid of rows and columns. A two-dimensional array having 3 rows by 4 columns is visualized as follows:

Row 1	column 1	column 2	column 3	column 4
Row 2	column 1	column 2	column 3	column 4
Row 3	column 1	column 2	column 3	column 4

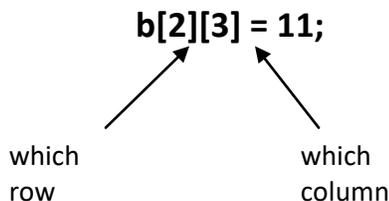
Here we declare and initialize a two-dimensional int array. We specify the number of rows and columns inside square brackets..



1	2	3	4
5	6	7	8
9	10	11	12

We assign values to the two-dimensional array by row and column index. The row index specified first and the column index specified second. The row and column index's both start at 0;

array name [row index] [column index] = value;



We retrieve values from the two-dimensional array also by row index and column index. The row index is first and the column index is second.

The row and column index's start at 0;

```
x = b[2][3];
printf("%d\n", b2[2][3]); // 11
```

The row index and column index of a two-dimensional array can be visualized as follows. The row index is first and the column index second. The row and column index's start at 0;

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

We use nested for loops to print out the values of a 2 dimensional array.

// print out values in a two-dimensional array

int r;

int c;

for (r=0;r < 3; r++)

{

for (c=0;c < 4; c++)

{

printf("%d\n", b[r][c]);

}

printf("\n");

}

```
1 2 3 4
5 6 7 8
9 10 11 12
```

To do

Use 2 nested for loops to assign new values to the two-dimensional array using some kind of formula like

$b[r][c] = r + 6 * c;$

Then print out the 2 dimensional array b;

LESSON 6 HOMEWORK Part1

Question 1

Make an array of 10 numbers 1 to 10, print out the numbers in the array, then add up all the numbers and print out the sum.

Question 2

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Ask the user of your program to enter a number in the array. Search for the number in the array and report if it is found or not found.

Question 3

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Ask the user of your program to enter a number in the array. Search for the number in the array and report the array index where the number was found otherwise print -1 meaning no index found.

Question 4

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Reverse all the numbers in the array in-place using a loop. Hint: use swap and 2 indexes i and j. Index i starts at the beginning of the array and index j starts at the end of the array. The i's increment and the j's decrement. Print out the reversed array.

Question 5

Make a 2 dimensional array of 3 rows and 3 columns. Fill the 2 dimensional array with numbers 1 to 9. Add up the sum of all rows, and print the sum at the end of each row. Add up the sums of all columns, and print the sums at the end of each column. Your output should look like this.

```
1 2 3 : 6
4 5 6 : 15
7 8 9 : 24
-----
11 15 18
```

Question 6

Make an array to hold 10 numbers 1 to 10.
Generate 1000 random numbers between 1 and 10.
Keep track of the random numbers generated in your array.
Print out all the numbers and their counts from the array.
Print out the numbers with the smallest and largest count.
Print out the number of even and odd number counts.
You can make a random number like this:

```
int x = rand() % 10
```

and you will also need to seed the random number generator to get different numbers when you run the program

```
srand((unsigned int)time(0))
```

You will need at the top of your c file:

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

So the C compiler recognizes the **srand()** , **rand()** and **time()** functions

Put all answers in a C file called Homework6.c

Arrays of Structures

Arrays of Structures allow you to group many structures together under a common name. We can make an array of person structures using the Person structure as follows.

```
struct Person
{
char name[MAX_CHARS];
int age;
};

struct Person persons[] = {
    {"Tom",24},
    {"Mary",20},
    {"Sue",28}
};
```

using typedef

```
typedef struct person_type
{
char name[MAX_CHARS];
int age;
} Person;

Person persons[] = {
    {"Tom",24},
    {"Mary",20},
    {"Sue",28}
};
```

todo

Make an array to hold 3 Persons and initialize with 3 persons.

In a loop print out the persons in the person array. Use the **printPerson** function from previous lesson to print out the person details

```
void printPerson(struct Person p)
{
    printf("Nice to meet you %s\n",p.name);
    printf("%s You are %d age years old\n", p.name, p.age);
}
```

Or the one using typedef

```
void printPerson(Person p)
{
    printf("Nice to meet you %s\n",p.name);
    printf("%s You are %d age years old\n", p.name, p.age);
}
```

LESSON 6 HOMEWORK part 2

Question 7

Make an Array of Structures using your **Profession** structure from Lesson3. In a loop print out the persons in the person array. Use your **printProfession** function to print out the Profession details. Put all your main function in a file called Homework6.c

Array of structures containing structures

We can also make an array of structures that contain other structure using the Student structure from previous lesson.

```

struct Student
{
Person p;
char idnum[MAX_CHARS];
};

```

You would initialize the array of structures like this:
 A structure inside a structure so each structure needs curly brackets.

```

struct Student students[] = {
    {{"Tom",24}, "S1234" },
    {{"Mary",20}, "S5678" },
    {{"Sue",28}, "S1111" }
};

```

Using typedef

```

typedef struct student_type
{
Person p;
char idnum[MAX_CHARS];
}Student;

```

You would initialize the array of structures like this:

```

Student students[] = {
    {{"Tom",24}, "S1234" },
    {{"Mary",20}, "S5678" },
    {{"Sue",28}, "S1111" }
};

```

You could visualize the 1 dimensional array of person structures like this

Person Structure	Person Structure	Person Structure
------------------	------------------	------------------

todo

Make an array to hold 3 Students and initialize with 3 students. In a loop print out the persons in the person array. Use the following **printStudent** function from previous lesson to print out the student details.

```
void printStudent(struct Student s)
+{
    printPerson(s.p);
    printf("Your Student id is %s\n",s.id);
}
```

Or using typedef

```
void printStudent(Student s)
{
    printPerson(s.p);
    printf("Your Student id is %s\n",s.id);
}
```

Most people use typedef.

You could visualize the 1 dimensional array of student structures like this

Student Structure	Student Structure	Student Structure
-------------------	-------------------	-------------------

LESSON 6 HOMEWORK part 3

Question 8

Make an Array of Structures of Structures using your **JobDescription Structure** from Lesson3. In a loop print out the job description in the job description array. Use your **printJobDescription** function to print out the JobDescription details Put all your main function in a file called Homework6.c

2 Dimensional Array of Structures.

We can also make a 2 dimensional array of persons structures like this:

```
Person persons2d[2][3] ={
    {
        {"Tom",24 },
        {"Mary",20},
        {"Sue"},
    },
    {
        {"Tom",24},
        {"Mary",20},
        {"Sue",28}
    }
};
```

We have 2 rows and 3 columns.

Notice: we also enclose the rows in curly brackets

You could visualize the 2 dimensional array of Person structures like this:

Person Structure	Person Structure	Person Structure
Person Structure	Person Structure	Person Structure
Person Structure	Person Structure	Person Structure

Todo

Print out the 2 dimensional array using nested for loops and use the printPerson function.

We can also make a 2 dimensional array of student structures like this:

```
Student students2d[2][3]={
    {
        {"Tom",24}, "S1234" },
        {"Mary",20}, "S5678" },
        {"Sue",28}, "S1111" },
    },
    {
        {"Tom",24}, "S1234" },
        {"Mary",20}, "S5678" },
        {"Sue",28}, "S1111" }
    }
};
```

You could visualize the 2 dimensional array of student structures like this:

Student Structure	Student Structure	Student Structure
Student Structure	Student Structure	Student Structure
Student Structure	Student Structure	Student Structure

Todo

Print out the 2 dimensional array using nested for loops and use the printStudent function.

LESSON 6 HOMEWORK part 4

Question 9

Make an 2D Array of Structures using your **Profession** structure from Lesson3. In a nested for loop print out the persons in the person array. Use your **printProfession** function to print out the Profession details. Put all your main function in a file called Homework6.c

Question 10

Make an 2D Array of Structures of Structures using your **JobDescription Structure** from Lesson3. In a nested for loop print out the job description in the job description array. Use your **printJobDescription** function to print out the JobDescription details
Put all your main function in a file called Homework6.c

LESSON 7 POINTERS and ALLOCATING MEMORY

When a program runs variables are stored in a computer memory location. Each variable is stored at a memory location. The memory location is known as an address. It is possible to get the address of the memory location using a pointer variable. Ordinary variables store **values**, pointer variables store **addresses**.

To declare a pointer variable we use a star* after the data type. A pointer is a variable that stores a memory location.

```
int* ptr = NULL;
```

We have declared an **int** pointer variable to store the address of an **int** variable. The pointer data type must match the data type of the variable that the pointer points to. We also have initialized the pointer variable to a default 0 address. called NULL. NULL is a constant representing a 0 address. A 0 address actually represents no address. Alternately you may see this as well

```
int* ptr = 0;
```

Using a 0 is considered poor programming practice, but you should always use NULL instead.

Declaring a pointer variable is just like declaring a ordinary value variable.

```
int x = 5;
```

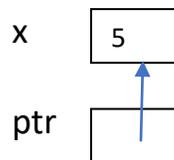
To get the address of a variable we use a & in front of the variable name.

& means “**address of**”

Our pointer can now point to the variable age using the & address of operator.

```
ptr = &x;
```

The pointer ptr variable now points to the variable x.

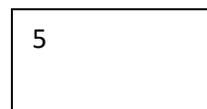


Once you have the address of the variable you can get the value that the pointer points to. You use a star * to get the value from the pointer. * means “**value of**”

```
int x = *ptr;
```

We can print out the value obtained from the pointer.

```
printf("%d\n",x);
```



Which would be the same as reading directly from the pointer *ptr.

```
printf("%d\n",*ptr);
```

5

You can also assign a new value to the age variable using the pointer

```
*ptr = 10;
```



Printing out the values that the pointer points to we now get a 10.

```
printf("%d\n",*ptr);
```

10

We can also print out the address that the pointer contains (the contents of the pointer). We printing out the address that the pointer contains using **%p** formatter:

```
printf("%p\n",ptr);
```

0x0005E000

%p is a format specifier used to print out a address.

We can also print out the address of the variable age using the **address of** operator **&** and the **%p** formatter.

```
printf("%p\n",&x);
```

0x0005E000

Note: Both addresses are the same.

todo

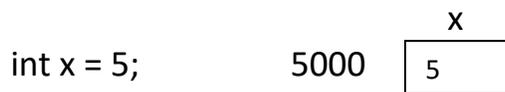
Try making some variables and pointer variables, assign values to the variable and assign the address of the variables to the pointer variable. Print out the values of the variables using the pointers.

Incrementing and decrementing a pointer

When you increment or decrement a pointer you are incrementing its contents by the memory size that the pointer points to. The pointer contents does not just increment by 1 but rather than the memory size of the variable it points to. The contents of a pointer is the memory address of some variable that the pointer points to.

Example:

X is at memory location 5000, Memory location 5000 contains the value 5.



ptr at memory location 6000 points to x at memory location 5000.



post increment pointer ptr++

pointer contents is now 5004, because the memory size of a int is 4 because sizeof(int) = 4. The pointer now points to an unknown value.



post decrement pointer ptr--

pointer contents is back to 5000, because the memory size of a int is 4 because sizeof(int) = 4. The pointer now points back to x.



Incrementing and decrementing the variable value that a pointer points to

Post increment the value that the pointer is pointing to:



The value of x has now incremented to 6

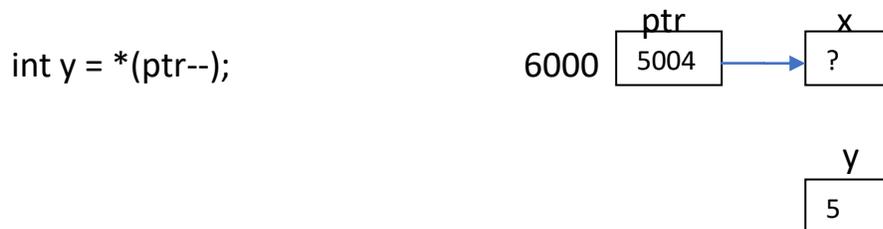
Pre increment the value that the pointer is pointing to:



The value of x has now decremented to 5. We are using post decrement.

Accessing values and Incrementing/decrementing pointers

Access the value of the pointer and post decrement the pointer



The value of x has now incremented to 6

y has the value 5

You may also use without the brackets.

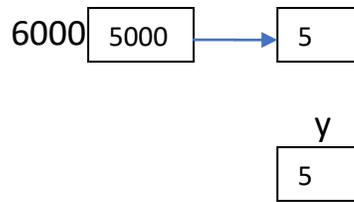
```
int y = *ptr--;
```

because ++ has precedence over *. The pointer decrements first, then the value is accessed.

Access the value of the pointer and pre-increment the pointer



```
int y = *(++ptr);
```



The value of x has now decremented to 5
y has the value 5 again because we have pre-decremented the pointer
You may also use without the brackets.

```
int y = *++ptr;
```

because ++ has precedence over *, the pointer increments first then the value is accessed.

todo

Try making some variables and pointer variables, assign values to the variable and assign the address of the variables to the pointer variable. Print out the values of the variables using the pointers. Then increment and decrement the pointers and print out the values they point to. Next increment the values they point to and print out their values. Lastly read the values from the pointers when you increment them or decrement them. Use pre increment and post increment.

Try to increment a pointer and a value all at the same time or vice versa.
Try to decrement a pointer and a value all at the same time or vice versa.

Pointers to Character Strings

```
char* s1 = "hello";  
char* s2 = "there";
```

Pointers to character strings are usually read only, because they are using code memory not data memory. You cannot change their values, but you can still copy them.

To do:

Make 2 two pointer to character strings and initialize them with your favourite words. Print(out the character strings using the %s formatter, don't forget a new line \n.

Printing out a character string using a loop and a pointer.

```
char* s = "tomorrow";
```

```
while (*s)
```

```
{
```

```
    printf("%c ",*s++;
```

```
}
```



tomorrow

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

loop to print out character string and increment a character before printing

In this situation we have to use reserved memory for the string not code memory. You cannot change code memory. We first reserve memory for the character string "tomorrow".

```
char s3[] = "tomorrow";
```

Then we assign the character pointer **s** to the character string **s3** start of memory.

```
s = s3;
```

```
while (*s)
```

```
{
```

```
    printf("%c ",++(*s++));
```

```
}
```

```
printf("\n")
```



u p n p s s p x

Pointers to Pointers

A pointer points to another pointer.

You declare a pointer to pointer with two stars ******.

```
int** pptr = NULL;
```

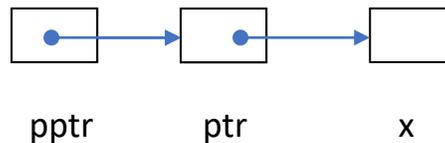
We use our pointer from previous section and assign to the address of the variable `x`

```
x = 5;  
ptr = &x;
```



We assign the address of a pointer to a pointer to pointer. A pointer to a pointer points to another pointer.

```
pptr = &ptr;
```



To access a value from a pointer to pointer is a 2 star process

```
x = **pptr;  
printf("%d\n",x); // 5
```

To assign the a value to a pointer to pointer is also a 2 star process

```
**pptr = 10;
```

We then print out the value from the pointer to pointer again

```
x = **pptr;  
printf("%d\n",x); // 5
```

Alternatively you access values using the pointer contained in the pointer to pointer

```
// Read value from pointer  
ptr = *pptr;  
x = *ptr;  
printf("%d\n",x);  
  
// Assign value using pointer  
*ptr = 10;  
x = **pptr;  
printf("%d\n",x);
```

Lesson7 Homework Part 1

Question 1

Make 2 int variables and assigned values to them.
Make 2 int pointers and assign the address of each variable to each pointer.
Use the pointers to swap the values. Print the values before and after the swapping.

Question 2

Make 2 int pointer to pointers. Assigned to each pointer to pointer the address of each pointer from question1.
Use the pointer to pointer to swap the values. Print the values before and after the swapping.
Next use the pointer obtained from the pointer to pointer swap the values. Print the values before and after the swapping.

Question 3

Make 2 character strings one initialized to your favorite word and the other one with reserved memory larger than the first character string. Copy from the first one into the second one backwards using a pointer. Print out the both strings after you copy them.

Allocating memory for Arrays

Alternatively you can allocate memory to a pointer using malloc. You need to put

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

at the top of your program so that the compiler knows what malloc is. You allocate memory for a int data type as follows:

```
ptr = (int *) malloc (sizeof(int))
```

(int *) is known as type casting which states the memory to be allocated is to represent an int address.

sizeof states the size of the data type to be allocated in bytes

sizeof(int) means the number of bytes for a int data type (usually 4 bytes)

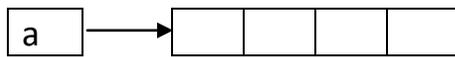
Allocating Memory for a 1 Dimensional Array

We can also allocate memory for arrays when the program is running;

```
int* a = (int*) malloc(sizeof (int) * 5);
```

a is known as a pointer because it holds the address of the allocated memory for the array. (points to the allocated memory)

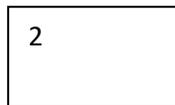
sizeof is used to indicate the size of a array column and the 5 means to have 5 columns.



A has the address of the 1 dimensional array

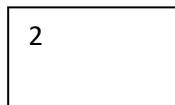
You can access the allocated memory by index

```
a[1] = 2;
x = a[1];
printf("%d\n", x);
```



or by the pointer **value of** operator *

```
*(a+1) = 2;
x = *(a+1)
printf("%d\n", x);
```



The value of * operator is also known as dereferencing. Dereference means to get the value from a pointer. A pointer is a reference to a memory address storing a value. So to get the value from a pointer it is called dereferencing.

Assigning values to a 1 dimensional array

When you allocate memory for a 1 dimensional array you need to give it some values using some formula.

```
// assign values to a 1 dimensional array
for (int i=0;i<5;i++)
{
        a[i] = (i+1);
}
```

Printing out values from a 1 dimensional array

We use for loops to print out values in array.

// print out values in a 1 dimensional array

```
for (int i=0;i<5;i++)  
    printf("%d ", a[i]);  
printf("\n");
```



1 2 3 4 5

Once you are finished using the allocated array you need to reclaim the memory, so other programs can use the memory.

```
free( a);
```

to do

Allocate 1 dimensional array of any size you want, fill it with values using a formula, then print out the array values and then free the array memory.

Allocating memory for a 2 Dimensional Array

We can also allocate memory for a 2 dimensional array . We first make a 1 dimensional array of integer row pointers (int**) known as **pointers to pointers**.

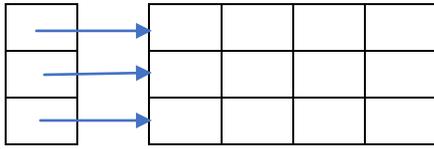
```
int** b = (int**) malloc(sizeof(int*) * 3); // declare number of row pointers
```

The **sizeof** operator indicated we need memory size for a int* and the 3 means an array of 3 int* pointers. * =multiply

Each row pointer will then point to a 1-dimensional array of int columns which will hold the values of the 2 dimensional array for each row.

```
b[0] = (int*) malloc(sizeof(int) * 4);  
b[1] = (int*) malloc(sizeof(int) * 4);  
b[2] = (int*) malloc(sizeof(int) * 4);
```

Picture a allocated 2 dimensional array like this. A 1 dimensional array of int pointers pointing to 3 a 1 dimensional array of int values.



Once you allocate memory for the 2 dimensional array you can assign new values to it like this:

```
b2[0][2] = 5;
```

You can retrieve values like this.

```
x = b2[0][5];
```

Where the first square bracket contents is the row index and the second square bracket contents is the column index. All indexes start at 0;

```
printf("%d\n",x);
```



Alternately you can use the value of * operator

```
*((*(b+1))+2) = 5;
```

Where `*(b+1)` get you a pointer to row 2 (index 1), because getting the value of a pointer to pointer is a pointer.

`((*(b+1))+2)` gets you a pointer to column 3 (index 2) the address of row 1 coliumn3. ALL memory values have their own address location.

The leading * lets you assign the value to the memory location the value 5.

```
*((*(b+1))+2) = 5;
```

We do the same for reading back the value, and then print it out.

```
x = (*(b+1)+2);  
printf("%d\n",x);
```

5

Most people use the square brackets [][] when accessing allocated 2 dimensional arrays .

Assigning values to a 2 dimensional array

We just assign values in a nested **for** loops.

```
int r = 0;  
int c = 0;  
for (r=0;r<3;r++)  
{  
    for (c=0;c<4;c++)  
    {  
        b[r][c] = (r+1)*c;  
    }  
}
```

Printing out values from a 2 dimensional array

We just print values in nested **for** loops

```
for (r=0;r<3;r++)  
{  
    for (c=0;c<4;c++)  
    {  
        printf("%d ",b[r][c]);  
    }  
  
    printf("\n");  
}
```

Deleting memory from an allocated 2-dimensional array.

We first delete memory for each row and then delete memory for the array that was storing the row pointers.

```
for (r=0;r < 3; r++)  
    {  
        free( b[r]);  
    }
```

```
free( b);
```

to do

Allocate 2 dimensional array of any rows and columns sizes you want, fill it with values using a formula, then print out the array values and then free the array memory.

Lesson7 Homework Part 2

Question 3

Allocate memory for a 1 dimensional array. Use a second pointer and a loop to assign values to each element of the array. You can increment the pointer contained address like this `p++` or use `*p++` to access a value and increment. Then print out the one dimensional array using the second pointer. Then free the memory for the 1 dimensional array.

Question 4

Allocate memory for a 2 dimensional array. Use a second pointer and a loop to assign values to each element of the array. You can use a pointer to pointer or just another pointer. You can increment the pointer contained address like this `p++` or use `*p++` to access a value and increment. Then print out the two dimensional array using the second pointer. Then free the memory for the 2 dimensional array

Allocating memory for a structure

We use the Person and Student Structure from Previous Lessons.

Where the **Person** structure is:

Not using typedef	using typedef
<pre>struct Person { char name[MAX_CHARS]; int age; };</pre>	<pre>typedef struct person_type { char name[MAX_CHARS]; int age; } Person;</pre>

And the **Student** structure is:

Not using typedef	using typedef
<pre>struct Student { struct Person p; char idnum[MAX_CHARS]; };</pre>	<pre>typedef struct student_type { Person p; char idnum[MAX_CHARS]; } Student;</pre>

You allocate memory to a structure using malloc. You need to put **#include <stdlib.h>** on the top of your program so that the compiler knows what malloc is.

You allocate memory for a Person structure data type as follows:

```
struct Person * ptr = (struct Person *) malloc (sizeof(struct Person));
```

If you use typedef then it is like this:

```
Person * ptr = (Person *) malloc (sizeof(Person));
```

(Person *) is known as type casting which states the memory to be allocated is to represent an structure Person address. We need to type cast because malloc returns a void* pointer. void* means a pointer to a no specified data type.

sizeof states the size of the data type to be allocated in bytes

sizeof(Person) means the number of bytes for a **Person** structure data type

When you access the variables in the structure using a pointer you use the arrow -> operator rather than the dot . operator

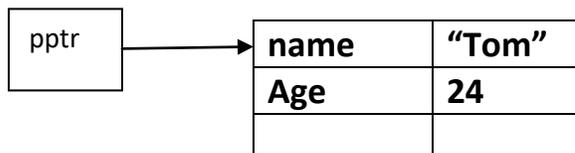
```
pptr->name
```

```
pptr->age
```

When you allocate memory for a structure you cannot initialize the elements in the structure when you create it, you must assign the values instead.

```
strcpy(pptr->name, "Tom");
```

```
pptr->age=24;
```



to do

Allocate memory for a person structure, assign values to it and print out the values, using a printf statement

You allocate memory for a Student structure data type as follows:

```
struct Student * sptr = (struct Student *) malloc (sizeof(struct Student));
```

If you use typedef then it is like this:

```
Student* sptr = (Student *) malloc (sizeof(Student));
```

(Student *) is known as type casting which states the memory to be allocated is to represent an Student structure address.

sizeof states the size of the data type to be allocated in bytes

sizeof(Student) means the number of bytes for a **Student** structure data type

When you access the variables in the structure using a pointer you use the arrow -> operator rather than the dot . operator

```
sptr->p.name
```

```
sptr->p.age
```

```
sptr->idnum
```

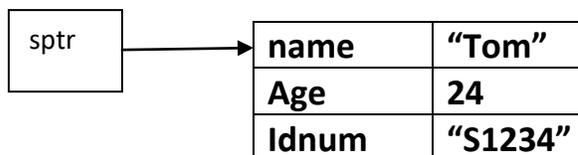
In the above situation we use the arrow -> operator on the p but use the dot operator to access the name from p, because p is not a person structure pointer but a just Person structure so we still use the dot . operator.

When you allocate memory for a structure you cannot initialize the elements in the structure when you create it, you must assign the values instead.

```
strcpy(sptr->p.name, "Tom");
```

```
sptr->p.age=24;
```

```
strcpy(sptr->idnum,"S1234");
```



To do

Allocate memory for a Student structure, assign values to it and print out the values using a printf statement

LESSON 7 HOMEWORK Part 3

Question 5

Allocate memory for your **Profession** structure from Lesson3 and fill in some values. Print out the profession using the printProfession function from also from Lesson3. Put all your main function in a file called Homework7.c

Question 6

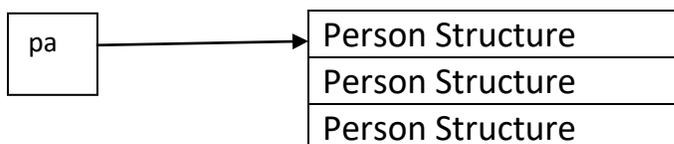
Allocate memory for your **JobDescription Structure** from Lesson3 and fill in some values. Print out the **JobDescription** using printJobDescription function also from Lesson3. Put all your main function in a file called Homework7.c

ALLOCATING MEMORY FOR AN ARRAY OF STRUCTURES

Note: we are using the **typedef** structure for the rest of the examples. **typedef** is much better to use.

Allocate an array to hold 3 persons and assign the array with 3 persons

```
Person* pa = (Person*) malloc(sizeof(Person)*3);
```



When you allocate memory for a structure you must assign values to each structure in the array.

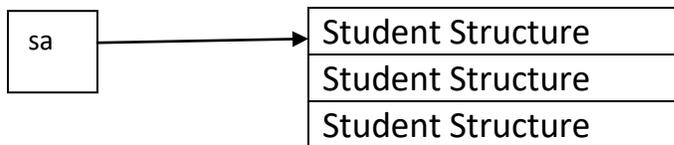
You use square brackets[] and a index to specify which structure you want to access.

To assign values to the first structure in the persons array you would do this:

```
strcpy(pa[0].name,"tom")  
pa[0].age = 24;
```

Allocate an array to hold 3 students and assign the array with 3 students.

```
Student* sa = (Student*) malloc(sizeof (Student)*3);
```



When you allocate memory for a structure you must assign values to each structure in the array.

You use square brackets[] and a index to specify which structure you want to access.

To assign values to the the first structure in the student array you would do this

```
strcpy(sa[0].p.name,"tom")  
sa[0].p.age = 24;  
strcpy(sa[0].idnum,"S1234");
```

todo

Allocate memory for array to hold 3 persons and initialize with 3 persons.
In a loop using a printf function print out the person details.

Allocate memory for array to hold 3 students and initialize with 3 students.
In a loop using a print function print out the student detail.

LESSON 7 HOMEWORK Part 4

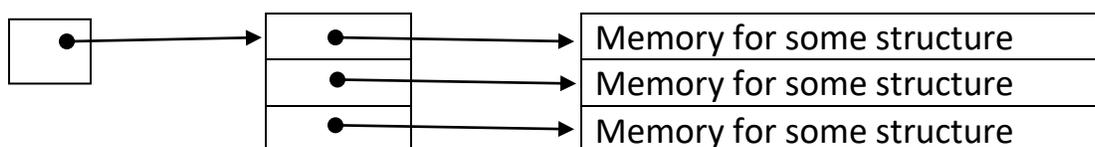
Question 7

Allocate an Array of Structures using your **Profession** structure from Lesson3 and fill in some values. In a loop print out the professions in the profession array using the printProfession function from also from Lesson3. Put all your main function in a file called Homework7.c

Question 8

Make an Array of Structures using your **JobDescription Structure** from Lesson3 and fill in some values. In a loop using a printJobDescription function print out the job description details. Put all your main function in a file called Homework7.c

ALLOCATING MEMORY FOR AN ARRAY OF STRUCTURE POINTERS

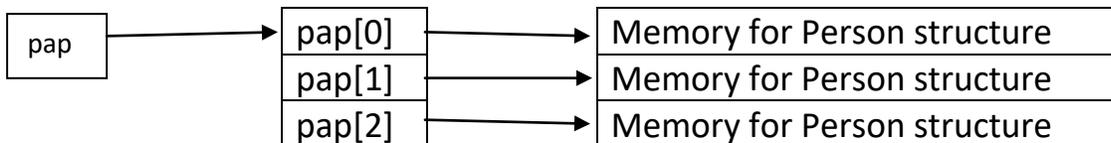


Allocate an array to hold 3 persons Structures pointers

```
Person** pap = (Person**) malloc(sizeof (Person*)*3);
```

You need to allocate memory for the array of Person structure pointers

```
pap[0] = (Person*) malloc(sizeof (Person));  
pap[1] = (Person*) malloc(sizeof (Person));  
pap[2] = (Person*) malloc(sizeof (Person));
```



When you allocate memory for a structure you must assign values to each structure in the array.

You use square brackets[] and a index to specify which structure you want to access,.

To assign values to the first structure in the persons array you would do this:

```
strcpy(pap[0]->name,"tom")  
pa[0]->age = 24;
```

to do

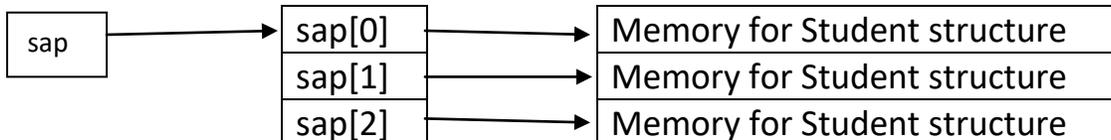
Allocate memory for array to hold 3 persons structure pointers and initialize with 3 persons. In a loop using a printf function print out the person details.

Allocate an array to hold 3 students structure pointers

```
Student** sap = (Student**) malloc(sizeof (Student*)*3);
```

You need to allocate memory for the array of Student structure pointers

```
sap[0] = (Student*) malloc(sizeof (Student));  
sap[1] = (Student *) malloc(sizeof (Student));  
sap[2] = (Student *) malloc(sizeof (Student));
```



When you allocate memory for a structure you must assign values to each structure in the array.

You use square brackets [] and a index to specify which structure you want to access,.

To assign values to the first structure in an array Student structures you would do this:

```
strcpy(sap[0]->p.name,"tom");  
sap[0]->p.age = 24;  
strcpy(sap[0]->idnum,"S1234");
```

We must use the arrow -> operator to access the student structure, since the array is storing structure pointers.

todo

Allocate memory for array to hold 3 students structure pointers and initialize with 3 students. In a loop using a print function print out the student detail.

LESSON 7 HOMEWORK Part 5

Question 9

Allocate an Array of Structures Pointers using your **Profession** structure from Lesson3 and fill in some values. In a loop print out the professions in the profession array using the printProfession function from also from Lesson3. Put all your main function in a file called Homework7.c

Question 10

Make an Array of Structures Pointers using your **JobDescription Structure** from Lesson3. In a loop using a printJobDescription function print out the job description details. Put all your main function in a file called Homework7.c

Converting memory from one data type to another using a Pointer

There are situations when you want to use a different data type from an existing data type. Data types are just bits of ones and zeros grouped together in bytes known as hexadecimal numbers. 1 byte is made up of 8 bits. It all depends in how you interpret the bits to arrive at the data type you want. Each data type interprets the data bits differently.

You may want to access individual bytes of the int data type separately. int usually is 32 bits made up of 4 bytes.

32 bits int			
Byte 0	Byte1	Byte2	Byte3

To access a int memory as a 4 byte array we first make a int variable called x and assigned hexadecimal 12345678 to it. (0x means hexadecimal)

```
int x = 0x12345678;  
printf("%04x\n", x);
```

We then assign a char pointer to it, we use the & to get the address of x and you need to type cast the int * data type to a char *

```
char* p=(char*)&x;
```

We now can read individual bytes from the in memory location

```
// reading bytes
```

```
int b1 = p[0];
```

```
int b2 = p[1];
```

```
int b3 = p[2];
```

```
int b4 = p[3];
```

12 34 56 78

```
printf("%0x %0x %0x %0x\n", b1,b2,b3,b4);
```

We can also change individual bytes to new values:

```
// writing bytes
```

```
p[0] = 0x11;
```

```
p[1] = 0x22;
```

```
p[2] = 0x33;
```

```
p[3] = 0x44;
```

11223344

```
printf("%0x %0x %0x %0x\n", p[0], p[1], p[2], p[3]);
```

```
printf("%04x\n", x);
```

Do not confuse type casting a pointer to type casting a non pointer variable,. When you type cast a pointer you are forcing the pointer type when you type casting a non variable you are type casting a value data type not an address data type.

void* is use to represent a memory address with no assigned data type.

You can also allocate a block of memory and then convert it to the data type you want later.

```
void* px = (void*)malloc(100);
```

Copying blocks of memory

You may have a void* block of memory of unknown data type, so you need to copy byte by byte.

We can copy using a pointer or using an index.
We first copy using a pointer.

```
// allocate memory block of 100 bytes  
void* m = malloc( 100);  
  
// type cast to a char*  
char* p = (char*)m;  
  
// initialize with sequential values  
int i = 0;  
for (i=0;i<100;i++)  
{  
    *(p++) = i;  
}
```

Note: we use ***(p++)** which assigns the value to the memory location pointed to by p the address of p is increment. The address increments in bytes to the size of the data type. For bytes it is 1 for ints it may be 4.

Which is quite different from **(*p)++**
(*p)++ means to increment the value pointed to by p by 1

```
// copy to another memory location
```

We first allocate memory using malloc.

```
// allocate memory block of 100 bytes  
void* m2 = malloc( 100);
```

```

// point to and type cast to first and second block of memory
char* p1 = (char*)m;
char* p2 = (char*)m2;

// copy block using pointers
for (int i=0;i<100;i++)
{
    *(p2++) = *(p1++);
}

// read back memory from second memory block
char* p2 = (char*)m2;
int i = 0;
for (i=0;i<100;i++)
{
    printf("%d ",*(p2++) );
}
printf("\n");

```

Copy block of memory not using a pointer and using index I instead

We first typecast m to a char*
(char*)m

We now use the square brackets and the i used to select each individual bytes
((char*)m)[i]

```

int i = 0;
for (int i=0;i<100;i++)
{
    ((char*)m2)[i] = ((char*)m)[i];
}

```

We can also use the * operator as well.
 We first typecast m to a char*

(char*)m

We now use and the i to select each individual byte as an offset to the start of the memory location

```
((char *)m)+i
```

We then use the * operator to get the byte value

```
*((char *)m)+i)
```

The round brackets are very important. The inner round brackets are used to type cast m to a char* so that the i index can act as a offset to the memory location m. The outer round braces are used for the star * to access the value at the selected byte.

```
int i = 0;  
for (i=0;i<100;i++)  
{  
    *(((char*)m2)+i) = *((char*)m)+i);  
}  
  
// read back memory from second memory block  
char* p2 = (char*)m2;  
int i = 0;  
for (i=0;i<100;i++)  
{  
    printf("%d ",*(p2++) );  
}  
printf("\n");
```

Lesson 7 HOMEWORK part 6

Question 11

Make a float variable and initialize with any value you like. Pass the float variable to the parameter of a void* function that will return an int. Type cast the input parameter as an int* and return the value of the int, back to the main into another int variable. In the main type cast the variable to a float* and print out the value. Put all your main function in a file called Homework7.c

Question 12

Make a unsigned int and assign 0x12345678 to it. Reverse the int to 0x87654321 using char pointers. Print the int before and after reverse. Put all your main function in a file called Homework7.c

LESSON 8 PASSING ARRAYS, STRUCTURES TO FUNCTIONS

Passing arrays to functions

Arrays are passed to a function by address. You can use the square operator [] or star * operator. They both indicate passing an array by address. When you are using the *operator them you can treat the array parameter as a pointer.

We can make a **printArray** function to demonstrate passing an array to a function by address and print out the values. We also include an additional parameter **n** to indicate the length of the array.

```
// pass array using [] operator
void printArray(int a[], int n)
{
    int i = 0;

    for(i =0;i < n;i++)
        {
            printf("%d ",a[i]);
        }
    printf("\n");
}

// pass array using * operator
void printArrayPtr(int* a, int n)
{
    int i = 0;

    for(i =0;i < n;i++)
        {
            printf("%d ",*a);
            a++;
        }
    printf("\n");
}
```

You would pass an array to the printArray functions as follows:

```
int a[] = {1,2,3,4,5};
```

```
printArray(a, 5);
```

```
1 2 3 4 5
```

```
printArrayPtr(a, 5);
```

```
1 2 3 4 5
```

to do

Passing a structure to a function by pointer

Passing structures to functions by pointer address is more efficient than passing a structure by value. When you pass a structure by value then all the memory or the structure is sent to the functions. When you pass a structure by pointer then the address of the structure is sent to the function, that is more efficient. When you pass a structure by value to a function, you cannot change the values of the outside values of the structure inside the function, since you are actually passing a copy of the structure values. When you pass a structure by pointer to a function then you can change the outside values of the structure inside the function, because you are passing the address of the structure to the function. You can change the values of the variables in the structure because you have the address of the structure variables.

Where the **Person** struct is:

```
typedef struct person_type  
{  
    char name[MAX_CHARS];  
    int age;  
} Person;
```

Note we are using **typedef** structure because it is more convenient.

We can change the **printPerson** function from previous lesson to accept a person structure pointer instead.

```
void printPerson(struct Person p)  
{  
    printf("Nice to meet you %s\n",p.name);  
    printf("%s You are %d age years old\n", p.name, p.age);  
}
```

Our person structure parameter now will have a star * to indicate pass by pointer.

To access values in a pointer structure variable you use the arrow operator ->

structure_variable_name -> variable_name

To access name by Person pointer p:

p->name

To access age by Person pointer p:

p->age

Here is the print person function using a person structure pointer parameter.

```
void printPersonPtr(Person* p)  
{  
    printf("Nice to meet you %s\n",p->name);  
    printf("%s You are %d age years old\n", p->name, p->age);  
}
```

To call the **printPerson** function we supply the address of the structure using the & operator.

```
Person p = {"Tom",24};  
printPersonPtr(&p);
```

to do:

Make a **printStudentPtr** function to accept Student pointer to print out details of a Student. Update the **printStudent** function to receive a Student structure by pointer. Use the **printStudentPtr function** to print out a student.

Here is the Student structure as a typedef.

```
typedef struct student_type  
{  
    Person p;  
    char idnum[MAX_CHARS];  
}Student;
```

Where the **Person** struct is:

```
typedef struct person_type  
{  
    char name[MAX_CHARS];  
    int age;  
} Person;
```

Passing Arrays of Structures to Functions

Arrays of structures are passed to a function by address. You can use the square operator [] or star * operator. They both indicate passing an array to structures by address. When you are using the *operator them you can treat the array parameter as a pointer. We can make a **printPersons** function to demonstrate passing an array of structures to a function by address and print out the values. We also include an additional parameter n to indicate the length of the array. Our **printPersons** function calls the **printPersonPtr** function rather than the

printPerson function to print out details of the person. The **printPersonPtr** function is more efficient.

```
// pass array of structure using [] operator
void printPersons(Person persons[], int n)
{
    int i = 0;

    for(i = 0; i < n; i++)
    {
        printPersonPtr(&persons[i]);
    }
    printf("\n");
}
```

```
// pass array of structures using * operator
void printPersonsPtr(Person* persons, int n)
{
    int i = 0;

    for(i = 0; i < n; i++)
    {
        printPersonPtr(persons);
        persons++;
    }
    printf("\n");
}
```

You would pass an array of structures to the `printPersons` functions as follows:

```
struct Person persons[] = {"Tom",24},{"Mary",20},{"Sue",28};
printPersons(persons, 3);
```

```
Nice to meet you Tom
Tom You are 24 age years old
```

```
Nice to meet you Mary
Mary You are 20 age years old
```

```
Nice to meet you Sue
Sue You are 28 age years old
```

Notice in the **printPersons** function and the **printPersonsPtr** function we have called the **printPersonPtr** function rather than the **printPerson** function

In this situation you need to pass the person structure as a pointer address like this, the **&** means **address of**

```
printPersonPtr(&persons[i]);
```

It is more efficient to pass by pointer.

todo

Make an array of 3 students.

Make an **printStudents** function that receives a student array and number of students in the array. And print out the students in the array using the **printStudentPtr** function.

Make an **printStudentsPtr** function that receives a student array as a pointer and number of students in the array. PRINT out the students in the array using the **printStudentPtr** function.

Homework part A

Question1

Make an Array of Structures using your **Profession** structure from Lesson3. Make a **printProfessions** function and a **printProfessionsPtr** to print out the persons in the person array. Make a **printProfessionPtr** function to print out the Profession details. Put all your main function in a file called Homework8.c

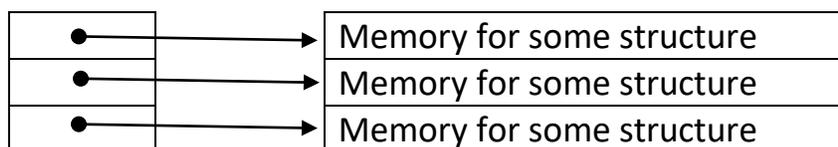
Question2

Make an Array of Structures using your **JobDescription Structure** from Lesson3. Make a **printJobDescriptions** function and a **printJobDescriptionsPtr** function to print out the job description in the job description array. Make a **printJobDescriptionPtr** function to print out the JobDescription details. Put all your main function in a file called Homework8.c

Passing an Array of Structure Pointers to a Function

Recapping:

An Array of Structure Pointers contain an array of pointers to a function, each element contains the memory address of some structure



Allocate an array to hold 3 persons Structures pointers

```
Person** pap = (Person**) malloc(sizeof (Person*)*3);
```

You would then initialize the arrays with Person structure memory addresses.

You can do the easy way like this:

```
Person p1 = {"tom",24};  
Person p2 = {"sue",22};  
Person p3 = {"bill",26};
```

```
pap[0] = &p1;  
pap[1] = &p2;  
pap[2] = &p3;
```

Or you can allocate memory for each structure and assign values to them the hard way like this:

```
pap[0] = (Person*) malloc(sizeof (Person*));  
pap[1] = (Person*) malloc(sizeof (Person*));  
pap[2] = (Person*) malloc(sizeof (Person*));  
strcpy(pap[0]->name,"tom")  
pap[0]->age = 24;
```

```
strcpy(pap[1]->name,"Sue")  
pap[1]->age = 20;
```

```
strcpy(pap[2]->name,"Mary")  
pap[2]->age = 26;
```

We now have to make a function to print out the array of structure pointers called `printPersonsPtrPtr`.

using * []

```
// pass array of structures using p*[] operator
void printPersonsPtrPtr(Person* persons[], int n)
{
    int i = 0;

    for(i = 0; i < n; i++)
    {
        printPersonPtr(persons[i]);
    }
    printf("\n");
}
```

Or using **

```
// pass array of structures using ** operator
void printPersonsPtrPtrPtr(Person** persons, int n)
{
    int i = 0;

    for(i = 0; i < n; i++)
    {
        printPersonPtr(*persons);
        persons++;
    }
    printf("\n");
}
```

Again we will use our printPerson function that receives a pointer

```
void printPersonPtr(struct Person* p)
{
    printf("Nice to meet you %s\n", p->name);
    printf("%s You are %d age years old\n", p->name, p->age);
}
```

When we call the printPersonPtr function we must only give them a persons[i]

```
printPersonPtr(persons[i]);
```

because persons[i] is already a pointer.

To do

Type in the above c code int your laesson8.c file and run it using both versions of print Persons pointer functions

```
printPersonsPtrPtr(pap,3);  
printPersonsPtrPtrPtr(pap,3);
```

You will get something like this:

```
Nice to meet you Tom  
Tom You are 24 age years old  
  
Nice to meet you Mary  
Mary You are 20 age years old  
  
Nice to meet you Sue  
Sue You are 28 age years old
```

We need now need to do the same for the Student structure

Here is the Student structure as a typedef.

```
typedef struct student_type  
{  
Person p;  
char idnum[MAX_CHARS];  
}Student;
```

Where the **Person** struct is:

```
typedef struct person_type
{
    char name[MAX_CHARS];
    int age;
} Person;
```

You should do this in steps

(1) Allocate an array to hold 3 students structure pointers

```
Student** sap = (Student**) malloc(sizeof (Student*)*3);
```

(2) Initialize the arrays with Student structure memory addresses.

```
Student s1 = {"Tom",24, "S1234" };
Student s2 = {"Mary",20, "S5678" };
Student s3 = {"Sue",28, "S1111"};
```

```
sap[0] = &s1;
sap[1] = &s2;
sap[2] = &s3;
```

(3) to do

make the print students array pointer functions

```
void printStudentsPtrPtr(Student* students[], int n)
void printStudentsPtrPtrPtrStudent** students, int n)
```

(4) run the program

You should get something like this:

```
Nice to meet you Tom
Tom You are 24 age years old
Your Student id is S1234
Nice to meet you Mary
Mary You are 20 age years old
Your Student id is S5678
Nice to meet you {Sue
{Sue You are 28 age years old
Your Student id is S1111
```

Homework part B

Question 3

Make an Array of Structures Pointers using your **Profession** structure from Lesson3. In a loop print out the persons in the person array. Make a **printProfessionPtrPtr** and **printProfessionPtrPtrPtr** function to print out the Profession details. Put all your main function in a file called Homework8.c

Question 4

Make an Array of Structures of Structures using your **JobDescription Structure** from Lesson3. In a loop print out the job description in the job description array. Make a **printJobDescriptionPtrPtr** and **printJobDescriptionPtrPtrPtr** functions to print out the JobDescription details. Put all your main function in a file called Homework8.c

Question5

In the Student structure change the Person structure to a punter. When you initialize the student structure you need to allocate memory for it and assign values to it, You will also need to change tithe printStudent functions just to accept a pointer for the person structure.

LESSON 9 FUNCTION POINTERS

A function pointer is a pointer to a function. Function pointers let you execute a function from calling it from the function pointer. Function pointers can be standalone, stored in an array or structure or passed to another function. They make your programming more convenient and optimal. Function pointers may be a little difficult to understand and use.

We first make a simple function that will print out the word "hello".

```
void printHello()
{
    printf("Hello\n");
}
```

We then declare a function pointer that will point to the hello function.

```
void (*f)();
```

void is the return type (*f) is the function pointer () is the parameter list of the called function. The parameter list receives argument values that are passed to the function when it executes.

The function pointer return type and parameter list must match the function you want to point to.

Next we assign the **printHello** function to the function pointer.

```
f = printHello;
```

The function pointer receives starting address where the code for the **printHello** function is stored.

We can now execute our function from the function pointer.

```
(*f)();
```

Hello

We are calling the **printHello** function from the function pointer using (*f) and the argument list(). We need to put the *f in round brackets (*f) to avoid confusion.

This is very similar to calling a regular function by its name: **f()**;
Except we use round brackets with a * around the function name: **(*f())**

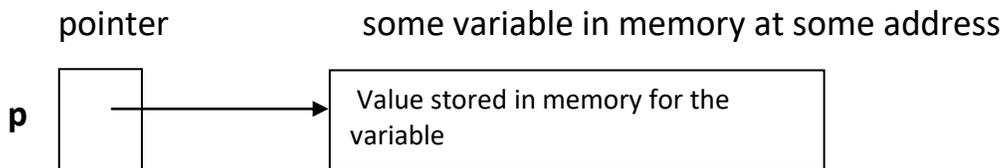
The only difference is that the function pointer requires a star * to extract the function code from the pointer just like a * in a regular pointer is used to extract the value from the pointer.

```
int x = 5;  
int* p = &x  
printf("%d\n",*p);
```

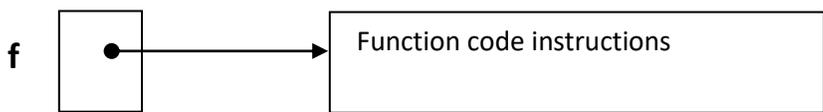


Without the * preceding the pointer it will print out the address of the variable not the value it contains.

```
int x = 5;  
int* p = &x  
printf("%d\n",p);
```



The same thing with the function pointer, the function pointer contains the address of the memory where the code for the function is stored. The * means to start executing the code instructions at that address.



function pointer some function in memory at some address

Here is the complete program:

```
#include <stdio.h>

// program to call a function pointer

// function to print out hello
void printHello()
{
    printf("hello\n");
}

int main()
{

// declare a function pointer
void (*f)();

// assign a function to function pointer
f = printHello;

// execute a function using a function pointer
printf(" execute a function using a function pointer\n");
(*f)();

return 0;
}
```

Todo

Type in or copy past the following program and run it, then change the message in the printHello function

passing arguments to a function called by a function pointer

We first make a function to receive a message, you will need to put the function at the top of your program.

```
void printMsg(char* msg)
{
    printf("%s\n",msg);
}
```

The next thing we need to do is make a function pointer with the same signature as the printMsg function.

```
void (*f2)(char* msg);
```

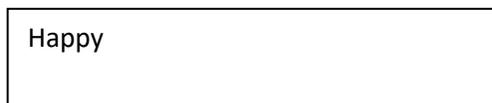
The function pointer return type and parameter list must match the function you are pointing to. In our case the parameter list receives a char* the message we want to print out and returns no value.

We now assign the **printMsg** function to the function pointer f2

```
f2 = printMsg;
```

We call the printMsg function using the f2 function pointer and pass it the char string "happy" to it, which it prints out on the computer screen.

```
(*f2)("Happy");
```



Happy

Here is the complete program:

```
#include <stdio.h>

// program to call a function pointer that receives a parameter

// function to print a message
void printMsg(char* msg)
    {
        printf("%s\n",msg);
    }

int main()
{

// declare a function pointer
void (*f)(char* msg);

// assign a function to function pointer
f = printMsg;

// execute a function using a function pointer
printf(" execute a function using a function pointer\n");
(*f)("Happy");

return 0;
}
```

Happy

Todo

Type in or copy past the following program and run it, then change the message in the passed to the printMsg function by way of the function pointer.

passing arguments to a function and receiving values from a function called by a function pointer

We first make a function that will receive 2 int values that will add them together and return the results. Remember you need to put this function at the top of your program to avoid compile errors.

```
int add(int a, int b)
{
    return a + b;
}
```

The next thing we make a function pointer f3 with the same signature as the add function.

```
int (*f3)(int a, int b);
```

We then assign the **add** function to the f3 function pointer.

```
f3 = add;
```

We now execute the function, passing 3 and 4 as arguments to it.

```
int x = (*f3)(3,4);
```

The function adds together the two values 3 and 4 and returns the result to the variable x that we then print out the value on the computer screen.

```
printf("%d\n",x);
```

7

Here is the complete program:

```
#include <stdio.h>

// program to call a function pointer that receives a parameter and returns a
value

// function add 2 values and return the result
int add(int a, int b)
{
    return a + b;
}

int main()
{

// declare a function pointer
int (*f)(int a, int b);

// assign a function to function pointer
f = add;

// execute a function using a function pointer
int x = (*f)(3,4);
print("\d\n",x);
return 0;
}
```

7

Todo

Type in or copy past the following program and run it, then change the values passed to the add function by way of the function pointer.

array of function pointers

The next thing we want to store many function pointers in an array so we can execute them all using a loop.

We make the following additional **sub**, **mul** and **divide** arithmetic functions that we put at the top of our program file.

```
int sub(int a, int b)
{
    return a - b;
}
```

```
int mul(int a, int b)
{
    return a * b;
}
```

```
int divide(int a, int b)
{
    return a / b;
}
```

We then make an array to hold the function pointers.

```
int (*f4[4])(int a, int b) = {add,sub,mul,divide};
```

It is same as making an array of **ints**

```
int a[4] = {1,2,3,4};
```

except we use a function pointer (***f4[4]**) instead

In a loop we execute all the arithmetic functions and print out the result values.

```
int i = 0;
for(i=0;i<4;i++)
{
    x = (*f4[i])(5,3);
    printf("%d\n",x);
}
```

8
2
15
1

The correct function is called and executed using the i index and the array function pointer (*f4 [i]) and argument values (5,3).

Here is the complete program

```
#include <stdio.h>
// program to call a function pointer that receives a parameter
// and returns a value
int add(int a, int b)
{
    return a + b;
}

int sub(int a, int b)
{
    return a - b;
}

int mul(int a, int b)
{
    return a * b;
}

int divide(int a, int b)
{
    return a / b;
}
```

```

int main()
{

// declare an array of function pointer's
int (*f4[4])(int a, int b) = {add,sub,mul,divide};

// execute each function in the array using the function pointer
int i = 0;
for(i=0;i<4;i++)
{
    x = (*f[i])(5,3);
    printf("%d\n",x);
}
return 0;
}

```

8
3
15
1

Todo

Type in or copy past the following program and run it, then change the values passed to the arithmetic function by way of the function pointer.

Allocating memory for an array of function pointers

We first make a variable to store an array **of function pointers to function pointers**.

```
int (**f5)(int a, int b) ;
```

We then allocated memory for the array of function pointers using malloc and assign to our function pointer array variable.

```
f5 = malloc(sizeof(**f5)*4);
```

This is the same thing as allocating memory for an array

```
int* pa = (int*)malloc(sizeof(int)*4)
```

except we use a function pointer (***f5**) we need the extra * because it is a pointer to a pointer (****f5**)

We then assigned the arithmetic functions to the allocated array.

```
f5[0] = add;  
f5[1] = sub;  
f5[2] = mul;  
f5[3] = divide;
```

We then execute the arithmetic functions from the array of function pointers

```
for(i=0;i<4;i++)  
{  
    x = (*f5[i])(5,3);  
    printf("%d\n",x);  
}
```

8
2
15
1

Here is the complete program

```
#include <stdio.h>  
  
// program to call a function pointer that receives a parameter and  
// returns a value  
int add(int a, int b)  
{  
    return a + b;  
}  
  
int sub(int a, int b)  
{  
    return a - b;  
}  
  
int mul(int a, int b)  
{  
    return a * b;  
}
```

```

int divide(int a, int b)
{
    return a / b;
}

int main()
{

// declare an array of function pointer's
int (**f)(int a, int b) ;

// allocate memory for the array of function pointers
f = malloc(sizeof(**f)*4);

// assigned the arithmetic functions to the allocated array.
f[0] = add;
f[1] = sub;
f[2] = mul;
f[3] = divide;

// execute each function in the array using the function pointer
int i = 0;
for(i=0;i<4;i++)
{
    x = (*f[i])(5,3);
    printf("%d\n",x);
}

return 0;
}

```

Using typedef to allocate memory for an array of function pointers (optional)

Typdef is a little easier to do but may be more difficult to understand. **Typedef** allows you to define your own data type from known data types like **int**.

```
typedef int(*funptr)(int, int);
```

Using typedef our data type is now **funptr** representing the function pointer signature of our arithmetic functions.

We now allocate an array of funptr's using malloc

```
funptr* f6 = malloc(sizeof(funptr)*4);
```

We now assign the arithmetic functions to our arrays of function pointers.

```
f6[0] = add;  
f6[1] = sub;  
f6[2] = mul;  
f6[3] = divide
```

we now execute all the arithmetic functions from the allocated array of function pointers.

```
for(i=0;i<4;i++)  
{  
x = (*f6[i])(5,3);  
printf("%d\n",x);  
}
```

```
8  
2  
15  
1
```

Passing a function pointer to another function

We first make a function called **calculate** that will receive a function pointer. We will pass one of our arithmetic functions to the **calculate function**

```
int add(int a, int b)
{
    return a + b;
}
```

The calculate function will call the received function pointer f with the received parameter values a and b.

Here is the **calculate** function, remember to put it at the top of your program to avoid compile errors.

```
void calculate(int(*f)(int a, int b),int a, int b)
{
    int x = (*f)(a,b);
    printf("%d\n",x);
}
```

The calculate function receives a function pointer

```
int(*f)(int a, int b)
```

having the same signature as one of the arithmetic functions like

```
int add(int a, int b).
```

The calculate function will call the add function using the function pointer and passes parameters a and b. The calculate function will then print out the calculated value it received from the arithmetic function.

Here is an example program:

```
#include <stdio.h>

// function add to add 2 numbers a and b
int add(int a, int b)
{
    return a + b;
}

// declare function calculate that has a function pointer f as a parameter
// function pointer f receives the arguments a and b
// which are the parameters a and b in the calculate method
void calculate(int(*f)(int a, int b),int a, int b)
{
    int x = (*f)(a,b);
    printf("%d\n",x);
}

int main()
{
    // declare and initialize variables a and b
    int a = 5;
    int b = 3;

    // call function calculate and pass the add function to it
    // calculate function receives the add function
    // and the variables a and b as parameters
    // the function calculate calls the add function to add the variables
    // the add function adds the variables a and b together
    // and returns the answer back to the calculate function 5 + 3 = 8
    // the calculate function receives the result 8 and prints to the screen

    calculate(add, a, b);

    return 0;
}
```

8

Todo

Type in or copy past the following program and run it, then change the values passed to the calculate function by way of the function pointer.

Passing an array of function pointers to another function

We can also make a function to receive an array of function pointers.

```
void do_calculations(int(*f[])(int a, int b), int a, int b)
{
    int i = 0;

    for(i=0;i<4;i++)
    {
        calculate(f[i],a,b);
    }
}
```

The **do_calculations** function receives an array of function pointers. Then in a loop it calls the calculate function passing a function point and values a and b to it. Using the pointer array and passed values a and b (5,3) it prints out this:

```
8
2
15
1
```

Here is an example program:

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}
```

```
int sub(int a, int b)
{
    return a - b;
}
```

```
int mul(int a, int b)
{
    return a * b;
}
```

```
int divide(int a, int b)
{
    return a / b;
}
```

```
// receive an array of function pointers
void do_calculations(int(*f[])(int a, int b), int a, int b)
{
    int i = 0;

    for(i=0;i<4;i++)
    {
        calculate(f[i],a,b);
    }
}
```

8
2
15
1

```
int main()
{
```

```
// declare an array of function pointer's
int (*f[4])(int a, int b) = {add,sub,mul,divide};
```

```
// declare and initialize variables a and b
int a = 5;
int b = 3;
```

```
do_calculations (f, a, b);
```

```
return 0;  
}
```

```
8  
2  
15  
1
```

To do

Try the above program, try different values of a and b.

Make a `do_calculations` function that get a pointer to a pointer like this:

```
void do_calculations_ptr(int(**f)(int a, int b), int a, int b)  
{  
    int i = 0;  
  
    for(i=0;i<4;i++)  
    {  
        calculate(f[i],a,b);  
    }  
}
```

Use the `do_calculation_ptr` function in the above program. Try with different values of a and b. The operation should be the same because `(**f)` and `(f[])` usually mean the same thing in most C compilers.

Next use make a allocated array of function pointer of arithmetic functions, and pass to the `do_calculation_ptr` function.

Storing a function pointer in a structure.

We can make a structure to hold a function pointer. We will use our person structure from before.

```
// structure to hold persons name, age and print function
typedef struct person_type
{
char name[81];
int age;
void (*printPerson)(char* name,int age);
}Person;
```

We can also use one of our print person function from previous lessons

```
// function to print out person info
void printPerson(char* name, int age)
{
    printf("your name is: %s, you are %d years old\n",name, age);
}
```

In our main function we make the person structure

```
// make the person structure
Person p = {"tom",24,printPerson};
```

Then we print out the person using the function pointer stored in the person structure like this:

```
// print person info
p.printPerson(p.name, p.age);
```

Here is the complete program:

```
#include <stdio.h>

// structure to hold person's name, age and print function
{
char name[81];
int age;
void (*printPerson)(char* name,int age);
}Person;
```

```

// function to print out person info
void printPerson(char* name, int age)
{
    printf("your name is: %s, you are %d years old\n",name, age);
}

int main()
{
//We make the person structure
Person p = {"tom",24,printPerson};

// print out the person using the function pointer
// stored in the person structure
p.printPerson(p.name, p.age);
return 0;
}

```

Todo

Type in or copy paste the above program and run it, and try different values. The allocate a person structure and rerun the program. What did you have to do to make it run?

Lesson 9 Homework

Make a message from an array of words. Make a print message function to receive the array of words message. Make functions lower, upper, reverse_letters and reverse_words.

Function	Description
Lower	Make all words lower case
Upper	Make all words upper case
Reverse_letters	Reverse all letters
Reverse_words	Reverse all letters in each word

Note: These functions change the message words in place, and does no printing.

Put functions lower, upper, reverse_letters and reverse words, in an array of function pointers. Call the print_message function with the array of function pointers and the array of words. Inside the print_message function call each function from the array of function pointers in a loop and print out the results. Call your homework file homework9.c

LESSON 10 FILE ACCESS

Files allow you to store data for later retrieval. Files are opened, read, written and closed using a file pointer. For this lesson make a new C source file called Lesson10.c and in the main function type in the following programming statements for the following File example programs.

FILE* fp;

Write character to a file

We use the **fputc** function to write characters one by one sequentially to a file. We first open the file "**test.txt**" for write with the **fopen** function using the **","w"** **write** specifier. Then we check if the file is open and the write characters to the file. You must close the file when you are finished writing characters. If you do not close the file, then the contents of the file will be lost.

```
// open file
FILE* fp = fopen("test.txt","w");

// check if file opened
if (fp != 0)
{
    // report cannot open file
    printf("cannot open file: test.txt for read\n");
    exit(1);
}

// make characters to write to file
char s[] = "Hello";

// write characters to file
int i=0;
char ch;
```

```

for (i = 0; i < strlen(s); i++)
{
    char ch = s[i]; // get char from string
    fputc(ch,fp); // write char to file
}

fputc('\n',fp); // write end of line char to file
fclose(fp); // close file

```

Read characters from a file

The **fgetc** function is used to read characters from a file. Each char from the file is read as an int so that the end of file EOF indicator -1 can be acknowledged.

We first open the file **"test.txt"** for read with the **fopen** function using the read **"r"** specifier. Then we check if the file is open and read character from the file. You must close the file when you are finished reading characters. If you do not close the file, then the file may not be able to be used by somebody else.

```

// open file
FILE* fp = fopen("test.txt","r");

// check if file opened
if (fp != 0)
{
    // report cannot open file
    printf("cannot open file: test.txt for write\n");
    exit(1);
}

// get first char in file
int ch = fgetc(fp)

```

```

// loop to end of file
while (ch != -1)
{
    putchar(ch); // print out char to screen
    ch = fgetc(fp); // get next char
}

fclose(fp); // close file
}

```

write lines to a file

The **fputs** function is also used to write lines one by one sequentially to a file. **fputs** automatically inserts the newline character `\n` at the end of the line for you. We first open the file `"test.txt"` for write with the **fopen** function using the write `"w"` specifier. Then we check if the file is open and the write characters to the file. You must close the file when you are finished writing characters. If you do not close the file, then the contents of the file will be lost.

```

// open file
FILE* fp = fopen("test.txt","w");

// check if file opened
if (fp != 0)
{
    // report cannot open file
    printf("cannot open file: test.txt");
    exit(1);
}

// write lines to file
fputs("Hello",fp);
fputs("there",fp);
fclose(fp);

```

Hello
there

Read line by line from a file

To read lines from a file line by line we use the **fgets** function that takes in a character string, length of character string and a file pointer. This function return 0 if the end of file is encountered.

We first open the file "**test.txt**" for read with the **fopen** function using the "**r**" specifier. Then we check if the file is open and read character from the file. You must close the file when you are finished reading characters. If you do not close the file, then the file may not be able to be used by somebody else.

We keep reading lines to the end of file is found. When the fgets function returns 0 the end of file has been reached. Some Unix compilers work different from Windows compilers. The **fgets** function may retain the end of line terminator **\n**. We can remove the **'\n'** character by using the **strstr** function that you were introduced to in the previous lessons.

```
// read lines from a file  
  
// open file  
FILE* fp = fopen("input.txt","r");  
  
// check if file opened  
if (fp != 0)  
{  
    // report cannot open file  
    printf("cannot open file: test.txt");  
    exit(1);  
}  
  
char line[256];  
char* ptr;
```

```

// loop to end of file
while (fgets(line,256,fp))
{
    // remove \n
    ptr = strstr(line, "\n");
    *ptr = 0;
    puts (line); // print out line
}

fclose(fp); // close file
}

```

<pre> Hello there </pre>

Append line's to end of file

The append "a" specifier is used to direct the **fputs** function to write lines to the end of the file one by one sequentially to a file. We first open the file "test.txt" for append write with the **fopen** function using the **append "a"** specifier. Then we check if the file is open and the write lines to the file starting to the end of the file. You must close the file when you are finished writing characters. If you do not close the file, then the contents of the file will be lost.

```

// append lines to end of file

// open file
FILE* fp = fopen("test.txt","a");

// check if file opened
if (fp != 0)
{
    // report cannot open file
    printf("cannot open file: test.txt");
    exit(1);
}

```

```
// append lines to file
fputs("see you later",fp);
fputs("goodbye",fp);
```

```
fclose(fp);
```

To do:

Printout test.txt file

```
Hello
there
see you later
goodbye
```

write to a csv file (comma separated values)

The **fprint** function is also used to write columns to a file separated by commas. We first open the file "**test.csv**" for write with the **fopen** function using the **write** "**w**" specifier. Then we check if the file is open and the write characters to the file. You must close the file when you are finished writing characters. If you do not close the file, then the contents of the file will be lost.

```
// write columns separated by commas to a file
```

```
// open file
FILE* fp = fopen("test.csv","w");

// check if file opened
if (fp != 0)
{
// report cannot open file
printf("cannot open file: test.txt");
exit(1);
}
```

```
// write lines to file
fprintf(fp,"one,two,three\n");
fprintf(fp,"five,six,seven\n");
```

```
fclose(fp);
```

```
one,two,three
four,five,six
```

Read a csv file.

A csv file is a file where data are stored row by row in columns separated by commas. The **strtok** function is used to separate the words between the commas.

File: test.csv

```
one,two,three
four,five,six
```

```
// read lines from a csv file
```

```
// open file
FILE* fp = fopen("input.csv","r");

// check if file opened
if (fp != 0)
{
// report cannot open file
printf("cannot open file: test.txt");
exit(1);
}
```

```
char line[256];
char* ptr;
```

```

// loop to end of file
while (fgets(line,256,fp))
{
    ptr = strtok(line, " ,\n\r");

    while(ptr != NULL)
    {
        printf("%s\n",ptr); // print out word
        ptr = strtok(NULL, " ,\n\r");
    }
}

fclose(fp); // close file
}

```

Output token words:

```

one
two
three
four
five
six

```

Writing and Reading Records to and from a file

Records are the data variable values defined in a structure written to a binary file. A binary file differs from a text file since it stores binary values where as a text file only contains printable values. The values in the binary and text files may be the same it is just the way they are interpreted. For example hex value 10 is interpreted as a new line in a text file but in a binary file it is just the value 10.

To write to a binary file you need some data record. A record can be a structure. The data variables declared in a structure must be fixed lengths, therefore the **string char*** pointer cannot be used. For our example we will use the Book structure as follows:

```

typedef struct
{
    char ISBN[20];
    char title[50];
    double price;

} Book;

// print a book
void printBook( Book& b)
{
    printf("%s %s $%.2f",b.ISBN,b.title,b.price );
}

```

The first thing we need to do is write some book records to a file. Each record is the data variable values defined in the Book class.

We first make a book structure.

```

Book book1={"123456789","Happy Days",23.56};

```

Then write the book record to the file using the **fwrite** method. The **sizeof** method calculates the total number of data bytes in the Book structure., the 1 just indicated we are writing just 1 data record.

```

fwrite((char*)&book1,sizeof(Book),1,fp);

```

When we open the file in binary write mode. "**wb**"

```

// write records to a file

// open file
FILE* fp = fopen("records.bin","wb");

```

```

// check if file opened
if (fp==NULL)
{
// report cannot open file
Printf("cannot open file: records.bin\n");
exit(1);
}
// write book to file
Book book1("123456789","Happy Days",23.56);
fwrite((char*)&book1,sizeof(Book),1,fp);

Book book2("876543245","Wizard of Oz",19.96);
fwrite((char*)&book2,sizeof(Book),fp);
fclose(fp);

```

Once we write some book records to the file we can read back the records and display them on the console screen. We use the **fread** function to read book records stored previously on the file. We open file with the mode "**rb**" read binary.

```

// read from binary file

// open file
fp = fopen("records.bin","rb");

// check if file opened
if (fp==NULL)
{
// report cannot open file
printf("cannot open file: records.bin\n");
exit(1);
}

// read records from a file
Book book;

```

```

while(fread((char*)&book,sizeof(Book),1,fp))
{
    printBook(book);
}

fclose();

```

<pre> 123456789 Happy Days \$23.56 876543245 Wizard of Oz \$19.96 </pre>

Append records to a binary file

We can also add new records to the end of the file using the "ab"

```

// append records to a binary file

// open file
fp =open("records.bin","ab");

// check if file opened
if (fp==NULL)
{
    // report cannot open file
    printf("cannot open file: records.bin\n" ;
    exit(1);
}

// write book to file
Book book3("87654542","Alice in Wonderland",18.88);
fwrite((char*)&book3,sizeof(Book),1,fp);
fclose(fp);

```

Again we read the book records from the binary file and display on the console screen.

```

// read from binary file

// open file
fopen("records.bin","rb");

// check if file opened
if (fp==NULL)
{
    // report cannot open file
    cout << "cannot open file: records.bin" << endl;
    exit(1)
}

// read records from a file
while(fread((char*)&book,sizeof(Book),1,fp))
{
    printBook( book);
}

fclose();

```

123456789 Happy Days \$23.56
876543245 Wizard of Oz \$19.96
87654542 Alice in Wonderland \$18.88

Open binary file for simultaneously Read and Write

Opening a file for reading and writing is very convenient, we open an existing file with the "**r+b**" mode and a new file with the "**w+b**" mode or append "**a+b**" to end of file. We will open with append to end of file "**a+b**"

```

// open file for read/write
File* fp = fopen("records.bin", "a+b");

```

```

// check if file opened
if (fp != NULL)
{
    // report cannot open file
    Printf( "cannot open file: records.bin for read and write\n");
    exit(1);
}

// write book to file
Book book4("765344532", "Open Skies", 12.78);
fwrite((char*)&book4, sizeof(Book), 1, fp);

```

Once we write a new record to the end of the file we can go to the start of the file and read each record one by one and display on the console screen.

We use the **fseek** function to set the file pointer to any position of the file.

```
int fseek ( FILE * stream, long int offset, int origin );
```

stream - Pointer to a FILE object that identifies the stream.

Offset - Number of bytes to offset from origin.

Origin - Position used as reference for the offset. It is specified by one of the following constants to be used as arguments for this function:

Constant	Reference position
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

To start at the end of the file:

```
fseek(fp, 0, SEEK_SET)
```

To go to the end of the file:

```
fseek(fp, 0, SEEK_END)
```

To go to current position in the file:

```
fseek(fp,0, SEEK_CUR)
```

To go to any position in the file:

```
fseek(fp,position, SEEK_SET);
```

We now read the file and print out the records

```
// read from start of binary file  
fseek(fp,0, SEEK_SET)  
  
while(fread((char*)&book,sizeof(Book),1,fp))  
{  
    printBook( book);  
}
```

123456789 Happy Days \$23.56 876543245 Wizard of Oz \$19.96 87654542 Alice in Wonderland \$18.88 765344532 Open Skies \$12.78
--

We can read/wrote from any position on the file using `fseek(fp,position, SEEK_SET)`
We now write a new book to record position 2.

The formula is:

file record position = record number * size of record

Record position and record numbers start at 0..

```
fio.seekp(2*sizeof(Book));
```

```
// write book to file  
Book book5("3443223475","Hello World",6.89);  
fwrite((char*)&book5,sizeof(Book),1,fp);
```

We read all records again:

```
fseek(fp,0, SEEK_SET)
```

```
while(fread((char*)&book,sizeof(Book),1)  
{  
    printBook( book);  
}
```

123456789 Happy Days \$23.56
876543245 Wizard of Oz \$19.96
3443223475 Hello World \$6.89
765344532 Open Skies \$12.78

We can specify which record to read using **fseek** Here we read record 2 from the file.

```
fseek(fp, 2 * sizeof(Book)), SEEK_SET);  
fread((char*)&book,sizeof(Book),1,fp);  
printBook( book);
```

87654542 Alice in Wonderland \$18.88

Always close the file when you are finished using it or you will lose all your data.

```
fclose(fp);
```

LESSON 11 RECURSION

When a function calls itself it is known as **recursion**. Recursion is analogous to a while loop. Most while loop statements can be converted to recursion, most recursion can also be converted back to a while loop.

The simplest recursion is a function calling itself printing out a message.

```
void print_message()  
{  
    printf("I like programming\n");  
    print_message();  
}
```

```
I like programming  
...
```

Unfortunately this program will run forever.

We can add a counter **n** to it so it can terminate at some point.

```
void print_message(int n)  
{  
    if(n > 0)  
    {  
        printf("I like programming\n");  
        print_message(n-1)  
    }  
}
```

Now the program will print the message **n** times

Every time the `print_message` function is called **n** decrements by 1. When **n** is 0 the recursion stops. You may place the statement `printf("I like programming\n")` before or after the recursive call. If you put it before than the message is printed first before each recursive call.

If you put after than the message is printed after all the recursive calls are made. There is quite a difference in program execution. The operation is very similar to the following while loop:

```
n = 5
while(n > 0)
{
    printf("I like programming\n");
    n--;
}
```

You should now run the recursion function

You would call the function like this:

```
print_message(5);
```

It will print I like programming 5 times.

```
I like programming
```

Recursion is quite powerful, a few lines of code can do so much.

Our next example will count all numbers between 1 and n. This example may be more difficult to understand, since recursion seems to work like magic, and operation runs in invisible to the programmer.

```
void countn(int n)
{
    if(n == 0)
    {
        return 0;
    }
    else
    {
        return countn(n-1) + 1
    }
}
```

count(5) would return 5

When (n == 0) this is known as the base case. When n == 0 the recursion stops and 0 is return to the last recursive call. Otherwise the **countn** function is called and n is decremented by 1.

It works like this:

```
main calls countn(5) with n = 5
countn(5) calls countn(4) with n=4
countn(4) calls countn(3) with n=3
countn(3) calls countn(2) with n = 2
countn(2) calls countn(1) with n = 1
countn(1) calls countn(0) with n = 0
```

```
countn(0) returns 0 to count(1) since n == 0
countn(1) add's 1 to the return value 0 and then returns 1 to count(2)
countn(2) add's 1 to the return value 1 and then returns 2 to count(3)
countn(3) add's 1 to the return value 2 and then returns 3 to count(4)
countn(4) add's 1 to the return value 3 and then returns 4 to count(5)
countn(5) add's 1 to the return value 4 and then returns 5 to main()
```

```
main() receives 5 from count(5) and prints out 5
```

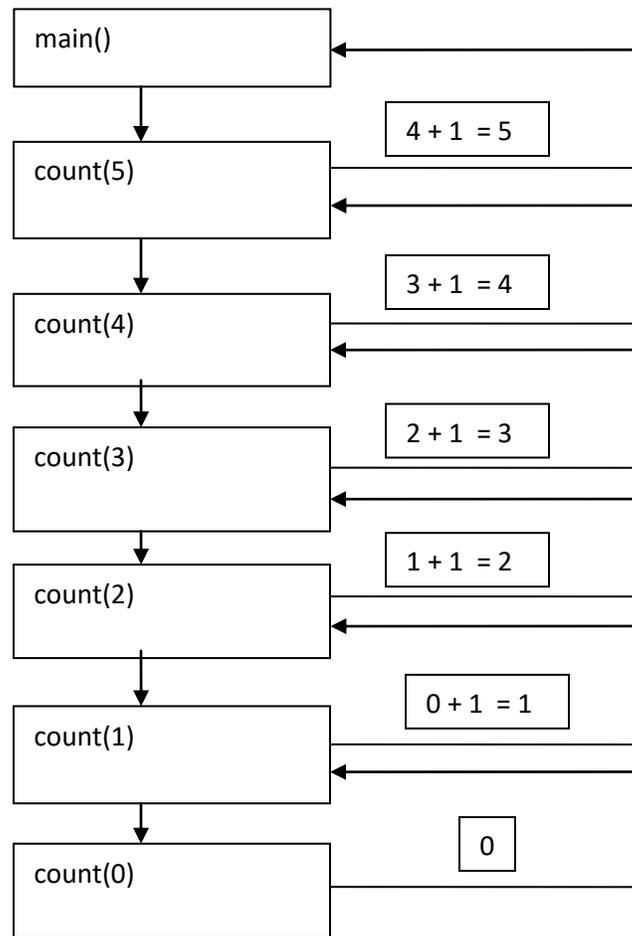
The statement **return countn(n-1) + 1** is used to call the function recursively and also acts as a place holder for the value returned by the called function.

We could rewrite the recursive part as follows:

```
int x = countn(n-1);  
return x + 1;
```

x will now receive the return value from the function call and 1 will be added to the return value and this new value will be returned to the caller.

If you can understand the above then you understand recursion. If you cannot then maybe the following diagram will help you understand.



You probably don't need to understand how recursion works right away. Sometime you just need to accept things for now then understand later. One day it will hit you when you are thinking about something else.

You could also think that main calls 5 functions sequentially where each function receives a number and returns the number and then 1 is added to it.

```

int count(int n)
{
    return n;
}
  
```

```
n = 0;
n = count (n)+1;
n = count (n)+1;
n = count (n)+1;
n = count (n)+1;
n = count(n)+1;
printf("%d\n",n);
```

Although not quite the same operation but it gives you an idea what the recursion is doing, it is just adding 1 to the number it has and then return the new value. Our **count** function just returns the value it receives, each statement adds 1 to it, then n receives the new value. Whereas in the recursion function **countn** returns 0 in the base case and then after each recursive call 1 is added to the value previously return to it and this new value is returned.

The thing to remember about recursion is it always return's back where it is called. Here are some more recursive function examples:

Sum numbers 1 to n

```
int sumn(int n)
{
    if(n ==0)
    {
        return 0;
    }

    else
    {
        return sumn(n-1) + n;
    }
}
```

sumn(5) would return 15

It works similar to countn instead of adding 1 its adds n.

$$0+1+2+3+4+5 = 15$$

Our counter n serves 2 purposes a recursive counter and a number to add.

Multiply numbers 1 to n (factorial n)

We can also make a **multn** function which multiples n rather than adding n. This is basically factorial n.

```
int multn(int n)
{
    if(n ==0)
    {
        return 1;
    }

    else
    {
        return multn(n-1) * n;
    }
}
```

multn(5) would return 55

since $1*1*2*3*4*5 = 55$

Our base case returns 1 rather than 0 or else our result would b 0;

Power x^n

Another example is to calculate the power of a number x^n

In this case we need a base parameter b and an exponent parameter n.

```

int pown(int b, int n)
{
    if(n ==0)
    {
        return 1;
    }

    else
    {
        return pown(b,n-1) * b;
    }
}

```

pawn(2,3) would return 9 because $2*2*2=8$ since $2^3=8$

Every time a recursive call is made the program stores the local variables in a call stack. Every time recursive call finishes executing, the save local variables disappear and the previous local variables are available. These are the ones present before the recursive function was called. These save variables may now be used in the present calculations.

For the above example $2^3=8$ the call stack would look like this.

			n=0			
			b=2	1		
			n=1	n=1		
			b=2	b=2	2	
	n=4	n=2	n=2	n=2		
	b=2	b=2	b=2	b=2	4	
n=5	n=5	n=3	n=3	n=3	n=3	
b=2	b=2	b=2	b=2	b=2	b=2	8

Every time the recursive function finished executing it returns a value. Each returning value is multiplied by the base b. In the above case the returning values are 1,2,4 and 8

The return value is the value from the previous function multiplied by b (2)

```
return pown(b,n-1) * b;
```

the function first returns 1 then $1 * b = 1 * 2 = 2$ then $2 * 2 = 4$ and finally $4 * 2 = 8$

efficient power x^n

A more efficient version of pown can be made relying on the fact that even n can return $b * b$ rather than just return $* b$ for odd n

```
int pown2(int b,int n)  
{  
  if (n == 0)  
  {  
    return 1;  
  }  
  
  if (n %2 == 0)  
  {  
    return pown2(b, n-2) * b * b;  
  }  
  
  else  
  {  
    return pown2(b, n-1) * b;  
  }  
}
```

Operation is now much more efficient $1 * 2 * 4 = 8$

Summing a sequence

Adding up all the numbers in a sequence

$n * (n + 1) / 2$

n	$n * (n + 2) / 2$
0	0
1	1
2	4
3	6
4	16
5	25

Total:	42

```

int seqn(int n)
{
    if(n == 0)
    {
        return 0;
    }

    else{

        return (n * (n + 2)) / 2 + seqn(n-1);
    }

}

```

Seqn(5) = 42

Fibonacci sequence

Recursion is ideal to directly execute recurrence relations like Fibonacci sequence

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

In mathematical terms, the sequence f_n of Fibonacci numbers is defined by the recurrence relation.

$$f_n = f_{n-1} + f_{n-2}$$

with seed values

$$f_0 = 0 \text{ and } f_1 = 1.$$

A **recurrence relation** is an **equation** that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

```
int fib(int n)
{
    if (n == 0)
    {
        return 0;
    }

    else if (n == 1)
    {
        return 1;
    }

    else
    {
        return fib(n-1) + fib(n-2);
    }
}
```

Notice The recursive statement is identical to the recurrence relation

fib(5) would return 5

Combinations

We can also calculate combinations using recursion.

Combinations are how many ways can you pick r items from a set of n distinct elements.

Call it nCr

Pick two letters from set $S = \{A, B, C, D, E\}$

Answer: $\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

There are 10 ways to choose 2 letters from a set of 5 letters. The combination formula is

$$nCr = \frac{n!}{r!(n-r)!}$$

The Recurrence relation for calculated combinations is:

base cases:

$$nC_n = 1$$

$$nC_0 = 1$$

recursive case:

$$nC_r = nC_{r-1} + nC_{r-1}$$
 for $n > r > 0$

Our recursive function for calculating combinations is:

```
int combinations(int n, int r)
{
    if (r == 0 || n == r)
    {
        return 1;
    }
    else
    {
        return combinations(n-1, r) + combinations(n-1, r-1);
    }
}
```

combinations(5,2) would return 10

Print a string out backwards

With recursion printing out a string backwards is easy, it all depends where you put the print statement. If you put before the recursive call then the function prints out the characters in reverse. Since n goes from n-1 to 0. If you put the print statement after the recursive call then the characters are printed not reverse since n goes from 0 to n.

```
// reverse a string
void print_reverse(char* s, int n)
{
    if(n == 0)
    {
        printf("\n");
    }

    else
    {
        printf("%c",s[n-1]);
        print_reverse(s, n-1);
    }
}
```

You would call the print_reverse function like this

```
char s[] = "hello";
print_reverse(s, strlen(s));
```

olleh

Check if a string is a palindrome

A palindrome is a word that is spelled the same forward as well as backwards: Like "radar" and "racecar"

```

int is_palindrome( char* s, int i, int j)
{
    if (i >= j)
        return 1;

    else
    {
        if (s[i] != s[j])
            return 0;
        else
            return is_palindrome(s,i+1, j-1);
    }
}

```

You would call the print_reverse function like this:

```

char s2[] = "radar";
printf("%d\n",is_palindrome(s2, 0,strlen(s2)-1));

```

1

```

char s3[] = "apple";
printf("%d\n",is_palindrome(s3, 0,strlen(s3)-1));

```

0

Permutations

Permutations are how many ways you can rearrange a group of numbers or letters. For example for the string "ABC" the letters can be rearranges as follows:

```

ABC
ACB
BAC
BCA
CBA
CAB

```

Basically we are swapping character and then print them out
We start with ABC if we swap B and C we end up with ACB

```
// print permutations of string s
void print_permutations(char* s, int i, int j)
{
    int k;
    char c;

    // print out permutation
    if (i == j)
    {
        printf("%s\n", s);
    }

    else
    {
        for (k = i; k <= j; k++) {

            // swap i and k
            c = s[i];
            s[i] = s[k];
            s[k] = c;

            // recursive call
            print_permutations(s, i + 1, j);

            // put back, swap i and k
            c = s[i];
            s[i] = s[k];
            s[k] = c;
        }
    }
}
```

You would call the **print_permutations** function like this:

```
char s4[] = "ABC";  
print_permutations(s4, 0, strlen(s4)-1);
```

```
ABC  
ACB  
BAC  
BCA  
CBA  
CAB
```

Combination sets

We have looked at combinations previously where we wrote a function to calculate how many ways you can choose r letters from a set of n letters.

nCr = n choose r

Combinations allow you to pick r letters from set $S = \{A, B, C, D, E\}$

$$n = 5 \quad r = 2 \quad nCr = 5C2$$

Answer: $\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

We are basically filling a seconded character array with all possible letters up to r .

Start with ABCDE we would choose AB then AC then AD then AE etc.

We use a loop to traverse the letters starting at $n = 0$, and fill the comb string.

When $n = r$ we then print out the letters stored in the comb string

```
void print_combinations(char s[], char combs[],  
    int start, int end, int n, int r)  
{  
    int i = 0;  
    int j = 0;
```

```

// Current combination is ready to be printed
if (n == r)
{
    for (j = 0; j < r; j++)
        printf("%c ",combs[j]);
    printf("\n");
    return;
}

// replace n with all possible elements.
for (i = start; i <= end && end - i + 1 >= r - n; i++)
{
    combs[n] = s[i];
    print_combinations(s, combs, i+1, end, n+1, r);
}
}

```

You would call the **print_combinations** function like this:

```

char s5[] = "ABCDE";
char combs[5+1] = {0};
print_combinations(s5, combs,0,strlen(s5)-1,0,2);

```

A B
A C
A D
A E
B C
B D
B E
C D

Determinant of a matrix using recursion.

In linear algebra, the determinant is a useful value that can be computed from the elements of a square matrix. The determinant of a matrix A is denoted $\det(A)$, $\det A$, or $|A|$

In the case of a 2×2 matrix, the formula for the determinant is:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For a 3×3 matrix A, and we want the formula for its determinant $|A|$ is

$$\begin{aligned} |A| &= \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= aei + bgf - ceg - bdi - afh \end{aligned}$$

Each determinant of a 2×2 matrix in this equation is called a "minor" of the matrix A. The same sort of procedure can be used to find the determinant of a 4×4 matrix, the determinant of a 5×5 matrix, and so forth.

Our code actually follows the above formula, calculating and summing the minors.

```
// calculate determinant of a matrix
float determinant(float matrix[3][3], int size)
{
    int c;
    float det=0;
    int sign=1;
    float b[3][3];
    int i,j;
    int m,n
    // base case
    if(size == 1)
    {
        return (matrix[0][0]);
    }
    else
    {
        det=0;
        for(c=0; c<size; c++)
        {
            m=0;
            n=0;
```

```

for(i=0; i<size; i++)
{
    for(j=0; j<size; j++)
    {
        b[i][j] = 0;
        if(i!=0 && j!=c)
        {
            b[m][n] = matrix[i][j];
            if(n<(size-2))
            {
                n++;
            }
            else
            {
                n=0;
                m++;
            }
        }
    }
    det = det + sign*(matrix[0][c]*determinant(b,size-1));
    sign = -1*sign; // toggle sign
}
}
return (det);
}

```

You call and run the determinant function like this:

```

float m[3][3] = {{6,1,1},{4,-2,5},{2,8,7}};

printf("det = %f\n",determinant(m,3));

```

-306

There are many more recursive examples, too numerous to present.
If you do all the following to do questions you will be a recursive expert.

Todo

Write a recursive function called **void reverse_string(char*s, int n)** that reverses a string in place. The recursive string receives the string and outputs the string in reverse. No printing is allowed.

Write a recursive function **int search_number(int a[], int n)** that searched for a number in an array and return the index of the number if found otherwise returns -1 if not found.

Write a recursive function **int search_digit(int d, int x)** that searched for a number in an array and return 1 if the number is found otherwise returns 0 if not found.

Write a recursive function called **int sum_digits (int d)** that adds up all the digits in a number of any lengths. The recursive function receives an int number and returns the sum of all the digits.

Write a recursive function called **void format_number(char* s, int n)** that can insert commas in a number. For example 1234567890 becomes 1,234,567,890

Write a recursive function **int is_even(int n)** that return true if a number has even count of digits or 0 if the number of digits is odd.

Write a recursive function **void print_binary(int d)** that would print a decimal number as a binary number. A binary number just has digits 0 to 1.

Where a decimal number has digits 0 to 9. The decimal number 5 would be 0101 in binary, since $1*1 + 0*2 + 1*4 + 0*8$ is 5. We are going right to left.

To convert a decimal number to binary You just need to take mod 2 of a digit and then divide the number by 2

$5 \% 2 = 1 \leftarrow 1$
 $5 / 2 = 2$
 $2 \% 2 = 0 \leftarrow 0$
 $2 / 2 = 1$
 $1 \% 2 = 1 \leftarrow 1$
 $1 / 2 = 0$
 $0 \% 2 = 0 \leftarrow 0$

We are done so going backwards
5 in binary is 0 1 0 1

Write a recursive function **int is_prime(int n)** that returns true (1) if a number is prime otherwise false (0).

A prime number can only be divided evenly by itself. 2,3,5,7, are prime numbers. You can use the mod operator % to test if a number can be divided evenly by itself. $4 \% 2 = 0$ 4 can be divided evenly by 2 so therefore 4 is not a prime number.

Put all your functions in a c file called Lesson11.c Include a main function that tests all the recursive functions.

LESSON 12 PROJECTS

Project 1 Spelling Corrector

Read in a text file with spelling mistakes, find the incorrect spelled words and offer corrections. The user should be able to choose the correct choice from a menu. Look for missing or extra letters or adjacent letters on the keyboard. Download a word dictionary from the internet as to check for correct spelled words. Use a **array** to store the words. Store the correct spelled file.

Project 2 MathBee

Make a Math bee for intermixed addition, subtraction, multiplication and division single digit questions. Use random numbers 1 to 9 and use division numbers that will divide even results. Have 10 questions and store results in a file. Keep track of the users score. You make random numbers like this:

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

```
// seed random number generator
srand((unsigned int)time(0))
```

```
// get random number 1 to 10
int x = (rand() % 10) + 1;
```

Project 3 Quiz App

Make a quiz app with intermixed multiple choice, true and false questions. You should have two structures MultipleChoice and TrueAndFalse. Store all questions in one file. Store the results in another file indicating the quiz results.

Project 4 Phone Book App

Make a phone book app that uses array of structures to store Phone numbers and names. You need an Contact structure to store name and phone number. You should be able to view, add, delete, scroll up and down contacts as menu operations. Contacts need to be displayed in alphabetically orders. Offer to lookup by name or by phone number. Contacts should be stored in a file, read when app runs, and saved with app finished running. Bonus, add email and address lookups as well.

Project 5 Appointment App

Make an Appointment book app that uses a array of structures to store Appointments. You need an Appointment structure to store name, description and time. You should be able to view, add, delete, scroll up and down appointments as menu operations. Appointments need to be displayed in chronological orders. Appointments should be stored in a file, read when app runs, and saved with app finished running.

END