

Basic Structure

Wednesday, 25 October, 2017 4:47 PM

-Header File: A header file acts as a library file. If a function that you need is inside of a header file, you must "include" that header file into the program

-Main Function: In C++ there always has to be one and only one "main function". This function is the first function that is run. It acts as a doorway into the program.

-Preprocessor: A program that processes input to create output that can be used as input in another program

Main program
Comments
Extra comments

```
/* .....
 *
 * THIS IS A BLOCK COMMENT SO THE LINES 1 TO 6 HAVE BEEN COMMENTED USING /STAR AS START AND STAR/ AS FINISH
 * THIS BLOCK COMMENT COULD BE USED AT THE BEGINING OF THE PROGRAM TO GIVE INFORMATION SUCH AS THE TITLE AND BUILD VERSION
 * .....
 */

//Introduction is where all the header files are which include functions or a library file
//The # is a pre-processor that using the "include" adds the function iostream which is short for Input/Output Stream
#include <iostream>

//It's using the standard (std) namespace
using namespace std;

//Setup

//Use has the main function. the gateway to other functions and parts of the program

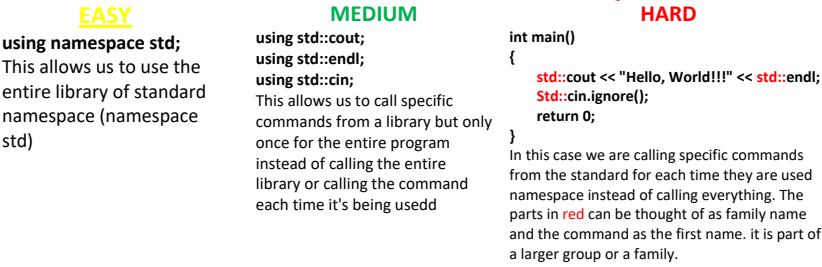
//one of multiple ways to create a main function
int main()

//everything in between the brackets is the main function
{
    //cout is the basic output command. endl ends the line and goes to a new line

    cout << "Hello, World!!!" << endl;
    cout << "String 2";

    //cin.ignore stops the program to close after finishing running the code
    cin.ignore();
    return 0;
}
```

using namespace std;
If we comment this part out then all of the commands that we used such as `cout` and `endl` and `cin` would be undefined which means the program won't know where they are from so standard namespace (namespace std) is a basic **library of commands**



Comments

Wednesday, 25 October, 2017 4:54 PM

-Line Comments: A comment that only covers a single line.

-Block Comment: A comment that engulfs multiple lines of code.

-Beauty Comment: Simply a block comment that has been formatted to stand out and be more eye appealing for example for very important parts

Bad Comment: Stating the obvious

Good Comment: Explaining the reasoning for the code or explaining the reasoning behind the design of the code

Input/Output

Wednesday, 25 October, 2017 6:55 PM

| <u>CODE</u> | <u>OUTPUT</u> | <u>Description</u> |
|---|---|---|
| Cout << "Hello world"; | Hello world | Outputs Hello world with nothing else |
| Cout << "Hello world" << endl; Cout << "This is my first program"; | Hello world This is my first program | Here we are trying to output 2 strings and they are on 2 separate lines, this is done by adding the endl at the end of line 1 and writing the second output in line 2 |
| Cout << "Hello world" << "This is my first program"; | Hello worldThis is my first program | Here we have two strings being outputted on the same command so they are on the same line |
| Cout << "Hello world" << endl << "This is my first program"; | Hello world This is my first program | Here we added an endl to separate the two strings and move the second one down by one line. This isn't the best way to do this since as the line is very long |
| Cout << "Hello world \nThis is my first program"; | Hello world This is my first program | Here we have used the \n inside a single string to separate two parts and put it in a new line |
| Cout << "Hello world \tThis is my first program"; | Hello world This is my first program | Here we have used \t to indent or "tab" the rest of the string |
| Cout << "Hello world \n\tThis is my first program"; | Hello world This is my first program | Here we have used \t in addition to \n to indent or "tab" the rest of the string in a new line |

A string requires a new header: **#include <string>** because string isn't part of the original C++ database. To call it and use it as a data type you just use **String** so it's not shortened

BIG PROBLEM

Using this you can only enter words. So if you try to type something like "Hello World" which has spaces, it won't work. To work around this we use **getline(cin, Name)**. This inputs a whole sentence with spaces and not just a word. This also doesn't require the **cin.ignore()** to be written after. Generally use **getline(cin, Name)** for strings and **cin >>** for other data types.

EXAMPLE CODE

```
#include <iostream>
#include <string> // This new module needs to be added to be able to use string

using namespace std;

int main()
{
    string Name, family, Name2; // string isn't shortened like int it's just string
    cout << "Please enter your first name: ";
    cin >> Name; // inputs a single word, no spaces
    cin.ignore();

    cout << "Please enter your last name: ";
    cin >> family; // single word no spaces
    cin.ignore();

    cout << "Enter a nice message: ";
    getline(cin, Name2); // a sentence with spaces. generally use this for strings and cin >> for other data types
    //No need for cin.ignore()

    cout << "Your name is: " << Name << " " << family << endl; // works just like python
    cout << "Your message is: " << Name2;
    cin.get();
    return 0;
}
```

RESULTS

Please enter your first name: Mahan
Please enter your last name: Noorbahr
Enter a nice message: I love C++
Your name is: Mahan Noorbahr
Your message is: I love C++

Variables and Basic Input

Wednesday, 25 October, 2017 8:12 PM

Format of variables:

| Data-type | Name | Value | Description |
|-----------|------------|----------|--|
| Int | My_age | (16) | Simple whole number |
| Float | My_Decimal | (3.1415) | Number with decimals |
| Char | My_char | (65) | One single character and the value is written in ascii so 65 is A in ascii |
| String | My_Name | "Mahan" | A series of characters. String requires the string header file. |

Integer Types

Tuesday, March 20, 2018 8:57 AM

Limits to values of an integer:

Since there are only 4 bytes of memory allocated to each integer variable, the maximum number of an integer value can be given using:

```
#include <iostream>
#include <limits.h>
```

Using namespace std;

```
Int main()
{
    Cout << INT_MAX;
    Cout << INT_MIN;
}
```

Results:

2147483647
-2147483648

Integer Types:

| Type | Memory allocated | Maximum | Minimum |
|------------------------|-------------------|----------------------------|----------------------------|
| Int | 4 bytes (32 bits) | 2,147,483,647 | -2,147,483,648 |
| Unsigned Int | 4 bytes (32 bits) | 4,294,967,295 | 0 |
| Short Int | 2 bytes (16 bits) | 32,767 | -32,768 |
| Unsigned Short Int | 2 bytes (16 bits) | 65,535 | 0 |
| Long Long Int | 8 bytes (64 bits) | 9,223,372,036,854,775,807 | -9,223,372,036,854,775,808 |
| Unsigned Long Long Int | 8 bytes (64 bits) | 18,446,744,073,709,551,615 | 0 |

Floating Point Types

Tuesday, March 20, 2018 11:03 AM

When dealing with large numbers, cout can decide to switch from **fixed** (760000) to **scientific** ($7.6 * 10^5$). We can manually do this using the fixed or scientific operators. Eg:

```
#include <math.h>
```

Results:

```
float num = 7.6 * pow(10,5); 760000
```

```
Cout << fixed << num;
```

$7.6 * 10^5$

```
float num2 = 760000;
```

```
Cout << scientific << num2;
```

3 Types:

| Type | Memory allocated | Min | Max |
|-------------|--------------------|--------------|---------------|
| Float | 4 bytes (32 bits) | 1.17549e-038 | 3.40282e+038 |
| Double | 8 bytes (64 bits) | 2.22507e-308 | 1.79769e+308 |
| Long Double | 12 bytes (96 bits) | 3.3621e-4932 | 1.18973e+4932 |

EXAMPLE CODE

| <u>Name</u> | <u>Operator</u> |
|-------------|-----------------|
| Add | + |
| Subtract | - |
| Multiply | * |
| Divide | / |
| Remainder | % |

Easier arithmetic:

| Arithmetic | Can be used instead of: |
|-------------------|--------------------------------|
| Num += 2 | Num = Num + 2 |
| Num -= 2 | Num = Num - 2 |
| Num *= 2 | Num = Num * 2 |
| Num /= 2 | Num = Num / 2 |
| Num %= 2 | Num = Num % 2 |

```
int main()
{
    float Num1, Num2, Num3; // This line creates 3 variables and gives them the data type of float
    cout << "Enter your first number: ";
    cin >> Num1; // Input number 1
    cin.ignore();

    cout << "Enter your second number: ";
    cin >> Num2; // input number 2
    cin.ignore();

    cout << Num1 << " + " << Num2 << " = " << Num1 + Num2; // adds the two numbers and outputs them
    cout << Num1 << " - " << Num2 << " = " << Num1 - Num2; // subtracts the two numbers and outputs them
    cout << Num1 << " X " << Num2 << " = " << Num1 * Num2; // multiplies the two numbers and outputs them
    cout << Num1 << " / " << Num2 << " = " << Num1 / Num2; // divides the two numbers and outputs them
    cin.get();
    return 0;
}
```

RESULTS
Enter your first number: 24
Enter your second number: 6
24 + 6 = 30
24 - 6 = 18
24 X 6 = 144
24 / 6 = 4

Member Function:

A function that is built into a data type that can be used on every variable of that data type. The functions are accessed by using a dot.

Member Access Operator:

The member access operator is the dot^

MEMBER FUNCTIONS

| | |
|---------|--|
| .at | Used to call the position of a string or an array like [] in python |
| .length | Used to give or store the length of a string or an array like len in python |
| .find | Used to find the position of a string in another string. In .at you find the value using the position but with .find you find the position using the value. |
| .substr | This function uses 2 parameters. When you have a string full of characters and you use .find to find a specific string it gives you the position of the first letter but then you can use substr to go to that position and take out |



```
int main()
{
    string Message, Name;
    int Pos, Len;
    cout << "Please enter a message including your name: ";
    getline(cin, Message);

    cout << "Please enter the position of your name: ";
    cin >> Pos;
    cin.ignore();

    cout << "Please enter the length of your name: ";
    cin >> Len;
    cin.ignore();

    Name = Message.substr(Pos-1, Len); //using the substring function we use the position of the value and the length of it to extract that value from a string
    cout << "\nYour name is " << Name;

    cin.get();
    return 0;
}
```

RESULTS

Please enter a message including your name: asdjashdasjkhdaMahanasdasdasdasd
Please enter the position of your name: 16
Please enter the length of your name: 5

Your name is Mahan

```
int main()
{
    string Name,Message;
    int Number;
    cout << "Please enter your name: "; //Input the user's name
    getline(cin, Name);

    cout << "Please enter a random number between 1 and " << Name.length() << ": "; // using .length to get the length of a string
    cin >> Number;
    cin.ignore();

    cout << "The letter at the position " << Number << " is " << Name.at(Number + 1) << endl; // Using the .at function to find the letter in the position that the user entered

    cout << "Please enter a message with your name in the word: ";
    getline(cin, Message);

    cout << "Your name is located at the position " << Message.find(Name) + 1; // using the .find function to find the position of a string in another string

    cin.get();
    return 0;
}
```

RESULT

Please enter your name: Mahan
Please enter a random number between 1 and 5: 3
The letter at the position 3 is h
Please enter a message with your name in the word: Hi my name is Mahan
Your name is located at the position 15

If Statement

Saturday, 11 November, 2017 10:32 PM

If statements are type of control structure which controls what your computer does and how it does it based on conditions. It also uses the same operators as python such as == or !=.

CONSTRUCTING AN IF STATEMENT:

```
If (condition)
{
    Command to be executed if condition is met
}
```

NOTE:

In python we use elif for the subsequent conditions after the first if statement but in C++ you have to use "else if". It does the same thing as elif. We can put each statement as if but using else if saves processing power because if the first statement is met then it won't do the others. The others are only tested if the one before is failed

EXAMPLE:

```
#include <iostream>

using namespace std;
```

```
int main()
{
    int Num = 10, User;
    cout << "Please enter a number";
    cin >> Guess;
    cin.ignore();

    if (User == Num)
    {
        cout << "Congrats you got it right!!";
    }

    cin.get()
    return 0;
}
```

OUTPUT:

Please enter a number: 10
Congrats you got it right

Conditions

Monday, March 26, 2018 12:24 PM

| Condition | Description |
|-------------------------|--|
| <code>==</code> | Equals. One value must equal another for the condition to be true. |
| <code>!=</code> | Not Equals. One value must be different from another for the condition to be true. |
| <code>></code> | More than. One value must be more than another for the condition to be true. |
| <code><</code> | Less than. One value must be less than another for the condition to be true. |
| <code>>=</code> | More than or equals to |
| <code><=</code> | Less than or equals to |
| <code>&&</code> | And. Two separate conditions must be true for the overall condition to be true |
| <code> </code> | Or. Either one of the conditions must be true for the overall condition to be true |

EXAMPLE:

```
#include <iostream>
using namespace std;
```

```
int main() {
```

```
    /*
     * == equal
     * < less than
     * > greater than
     * <= less than or equal
     * >= greater than or equal
     * != not equal
     *
     * && and
     * || or
     */
```

```
    int val1 = 7;
```

```
    int val2 = 10;
```

```
    if (val1 < 8 && val2 > 9) {
        cout << "And True!" << endl;
    } else {
        cout << "And False." << endl;
    }
```

```
    if (val1 == 7 || val2 > 11) {
        cout << "Or True!" << endl;
    } else {
        cout << "Or False." << endl;
    }
```

```
    if (val1 >= 7) {
        cout << ">= True!" << endl;
    } else {
        cout << ">= False." << endl;
    }
```

```
    bool cond1 = val1 < 8 && val2 > 9;
    bool cond2 = val2 != 88;
```

```
    if (((val1 < 8) && (val2 > 9)) || (val2 != 88)) {
        cout << "Condition 1 is true" << endl;
    }
    else {
        cout << "Condition 1 is false" << endl;
    }
```

```
    return 0;
```

```
}
```

Result:

And True!

Or True!

>= True!

Condition 1 is true

There are two ways of doing complex conditions.

1) Using Boolean variables to do comparison and use those in if statements

2) Doing all the comparison in the statement but using brackets. Very complicated.

Comparing Floats

Sunday, March 25, 2018 10:32 PM

When we try to compare floats in an if statement or in other places, we come across an issue:

```
int main()
```

```
{
    float Num = 3.14;

    if (Num == 3.14)
    {
        cout << "Good" << endl;
    }
    else{
        cout << "Bad" << endl;
    }
    cout << setprecision(20)<< Num;
}
```

Result:

Bad
3.1400001049041748047

This doesn't work because after a certain decimal places (7 for float and 15 for double) there is random numbers generated which doesn't match with the intended value therefore it doesn't work.

Solution:

Instead of trying to see if the number exactly matches the float we can check whether it is in a range or not:

```
int main()
```

```
{
    float Num = 3.14;

    if (Num < 3.15 and Num > 3.13)
    {
        cout << "Good" << endl;
    }
    else{
        cout << "Bad" << endl;
    }
    cout << setprecision(20)<< Num;
}
```

Result:

Good
3.1400001049041748047

This works because the value including the random junk is between 3.13 and 3.15

Loops

Saturday, 11 November, 2017 10:27 PM

WHILE LOOPS:

While Loops carry out a set of instructions as long a condition is met.

In C++ this is done by using the command

```
while ( condition )
```

```
{  
    Set of instructions  
}
```

FOR LOOPS:

For loops carry out a set of instructions in a set number of repeats. In python a for loop is used to do the instruction a set number of times but in C++ for loops act similar to While Loops.

```
for (int Loop = 0; Loop != 4; Loop++)  
{  
    cout << "Hi " << Loop + 1 << endl;  
}
```

Result:

Hi 1
Hi 2
Hi 3
Hi 4

DO-WHILE LOOPS:

Do while loops are just like while loops but in a different format. They are great for using them as the loop where your entire program is placed in. The difference between a while and a do while loop is that a do while will execute the code at least while. It executes the code then checks for the condition.

```
Do{  
    Code  
} while(condition);
```

Arrays

Tuesday, March 27, 2018 2:11 PM

Arrays can be used to store multiple value or "Elements" in them.

```
Int Numbers[5] = {2,5,6,3,2}
```

The array above is called "Numbers" and has 5 elements. These elements are all integers and have the values 2, 5, 6, 3, 2. If you don't declare any elements, you must declare the size of the array. If you declare elements, you don't need to declare array size

Just like python **all strings are arrays**.

This means that if we have a string we can output each position or "element" of the string.

Result:

h

```
#include <string>
```

```
String name = "Mahan";
```

```
Cout << name[2];
```

2D Arrays

Tuesday, March 27, 2018 3:14 PM

Sometimes you don't have a list of elements, you have a table of elements. We can use 2D arrays. An example of a 2D array:

```
string animals[2][3] = {  
    {"fox", "dog", "cat"},  
    {"mouse", "squirrel", "parrot"}  
};
```

This 2D array is a **string type** and is named **animals**. There are **2 rows** and **3 columns** (two arrays each with 3 elements).

| | Column 1 | Column 2 | Column 3 | Column 4 |
|-------|----------|----------|----------|----------|
| Row 1 | x[0][0] | x[0][1] | x[0][2] | x[0][3] |
| Row 2 | x[1][0] | x[1][1] | x[1][2] | x[1][3] |
| Row 3 | x[2][0] | x[2][1] | x[2][2] | x[2][3] |

NOTES:

You can leave the number of rows empty if you have declared number of arrays but you can never leave the number of columns empty. The number of elements in each array also must be equal for every array in the 2D array.

Find the size of an array without hard coding it:

Let's say we have an array of integers:

```
Int Array[] = {5,6,3,8}
```

We have 4 integers in the array and we know that each integer is 4 bytes in size (using sizeof(int)). We can use sizeof(Array) to get the size of the array.

We can use it the other way around so to find the number of elements we can divide the sizeof(array) by the sizeof(int). If we do this to the example we have $16/4 = 4$ which is the number of elements.

Switch Statement

Wednesday, March 28, 2018 11:39 PM

Switch is just like having multiple if statements.

Let's say we have a main menu for our console program and the user can have options 1 - 4. instead of using 5 if statements (4 for options and 1 for else) we can use switches:

```
switch (Choice) {  
  case 1:  
    Execute block of code for option 1  
    break  
  case 2:  
    Execute block of code for option 2  
    break  
  case 3:  
    Execute block of code for option 3  
    break  
  case 4:  
    Execute block of code for option 4  
    break  
  default:  
    What happens if they don't enter one of the above.  
}
```

NOTE:

You don't need the default statement but it is there to act like the else statement.

YOU SHOULD PUT BREAK AT THE END OF EACH CASE. Except the default statement or the last statement because. As soon as the switch function finds a case that matches the value, after executing the code it will ignore other cases and executes everything in them so you need to break out of it.

YOU CANT USE VARIABLES AS CASE LABELS (after case). You can't do `int num = 4`
Case num:
We can't use variables as case labels.
However we can use constant variables:
`Const int num = 4.`

Switches are only to be used with integers. Floats are too inaccurate to use and strings are not supported.

IN CASES THE SCOPE OF VARIABLES ARE NOT THE SAME AS IF STATEMENTS.

In if statements, any variable declared will only be in that statement so you can use them in other options too regardless of the changes made in previous statement. In switches any variables declared in any case will stay in the program afterwards so you can't redeclare them in another case.

FUNCTIONS

Thursday, March 29, 2018 2:50 PM

Creating and calling functions is very similar to Python. The structure of a function consists of a **return value type**, **Name of the function**, **arguments** (inputs from the actual program to be used in the function) and **code to be executed**:

```
int MainMenu(){  
    Int choice;  
    Cout << "1) option 1" << endl;  
    Cout << "2) option 2" << endl;  
    Cin >> choice;  
    Return choice;  
}
```

Here we have created a simple function called **MainMenu**. This function has the return value type of **int** which basically means it returns an integer. It has **no arguments** and all it does is that it outputs a few lines.

```
Int main(){  
    Choice = MainMenu();  
    Return 0;  
}
```

Return Value types:

| Type | Description(example) |
|---------------|---|
| Int | Returns an integer (addition) |
| String | Returns a string (input 20 names) |
| Float | Returns a float (calculating area of circle) |
| Double | Returns a double (calculating speed of light) |
| Void | Returns NOTHING (main menu program) |
| Shot/Long int | Returns short or long integers (calculate your birthday) |
| Long Double | Returns a long double (no clue why you'd use a long double) |

So far we could only define functions above our main function. Let's take this example:

```
int main()
{
    DoSomething();
    return 0;
}

void DoSomething()
{
    cout << "Hello";
}
```

This program wouldn't run because the compiler, compiles from **top to bottom** and when it comes across "DoSomething" in main it **doesn't yet know what that means**. This can be resolved by using **prototypes**. Prototypes tell the compiler that this function exists somewhere in the program so don't be spooked when you see it called. It requires the return type, name and arguments of the function:

```
void DoSomething();           <<<<----- prototype

int main()
{
    DoSomething();
    return 0;
}

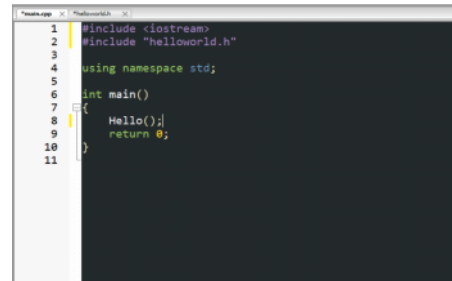
void DoSomething()
{
    cout << "Hello";
}
```

HEADER FILES:

When we have multiple functions that are generally in the same category (Validation functions) we put them on a different file called a header file. We can make header files in our project in CodeBlocks by going to Files tab, New, File, Header file. This new header file can be called the same way we call the <iostream> header file.

Example:

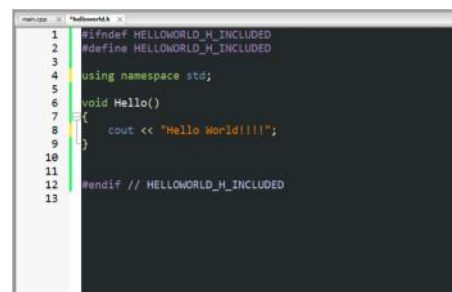
- Main program:



```
1 #include <iostream>
2 #include "helloworld.h"
3
4 using namespace std;
5
6 int main()
7 {
8     Hello();
9     return 0;
10 }
```

This program is a simple hello world program where **output of hello world has been replaced with a function** and that **function is situated in the header file called "helloworld.h"**. When the program is being compiled, the compiler comes across the keyword include and the **preprocessor** goes to the header file (helloworld.h in this case) and copies the contents of that file straight into where the statement was. **WE USE DOUBLE QUOTES BECAUSE ANGLE BRACKETS REFER TO STANDARD DIRECTORY THAT THE COMPILER KNOWS.**

- Header file:



```
1 #ifndef HELLOWORLD_H_INCLUDED
2 #define HELLOWORLD_H_INCLUDED
3
4 using namespace std;
5
6 void Hello()
7 {
8     cout << "Hello World!!!!";
9 }
10
11 #endif // HELLOWORLD_H_INCLUDED
```

This is the header file. A plain text file which can hold functions or classes. Because we are using cout in our function which is under the std namespace **we need to declare that we are using the namespace std**. We don't need to include any header files however the **required header files need to be either in the program or the header file the function is being written in.**

So far everything we've been learning is pretty much the same things we learned in python. This is known as **procedural programs** where the entire program is just one big chunk of code. Now we can do **object oriented programming in c++**.

What is an Object Oriented Language (OOL/P)?

In procedural languages like Python and C, programs are written in long chunks of code. Functions may be used but the program will be one giant piece of code. However in object oriented languages like C++ and Java, the code is **split up into self-oriented objects** like having a lot of mini programs.

Each object has its own data and logic and they communicate between themselves.

6 aspects of OOP:

1)

Classes:

A class is an idea, a blueprint or a **specification of how something should be made but it isn't the thing itself**. All classes define two things:
(In case of a person)

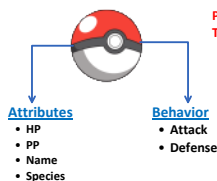
| (data_variables) | (functions) |
|--|-------------------------------------|
| <u>Attributes</u> (Properties in C++) | <u>Behavior</u> (Methods in C++) |
| Name Height Weight Age | Walk Run Jump Sleep |

2)

Object:

An object is **created from a class**. You can make multiple objects from a class like how you can make multiple houses from a blueprint. Objects represent things that exist in real world, animals, houses, cars and anything.

GREAT EXAMPLE:



Pokémon --> Class
The creatures --> Objects

BLUEPRINTS FOR CREATING A POKEMON

Abstraction:

Let's take the example of a student management software. We need to create a class for "student". We know we need attributes and behaviors:

Attributes:

- Student ID
- Name
- Age
- Marks
- Address
- Height
- Weight

Now we have all these attributes possible but do we need all these for a school student management software? No we only need a certain few so **we take the necessary details and ignore the rest**.

"Private" and "Public" are called interfaces.

Eg:
The remote control of an AC. You aren't supposed to open the AC and manually change the temperature, instead you have the remote which gives u limited access to what you can change and what you cant. Imagine the AC is the data that we want to encapsulate.

4)

Encapsulations:

Encapsulation is protecting the data in the object from code or functions that do not need those data. If we look back at our student management class we can write it in 3 ways:

- Not Encapsulated
- Over encapsulated (useless)
- Correctly Encapsulated

Class Student

```
{  
Private:  
    Number;  
    Name;  
    Marks;  
Public:  
    Private  
    ShowMarks();  
    ChangeMarks();  
}
```

If we make both properties and methods private, the class is not accessible by anything outside the {}. So it is basically useless.

So we use a way to encapsulate the data without making it useless, the data itself is private but the functions can access the data, this makes the data viewable from a controlled manner. If a function modifies the data, it can ask for verification before allowing changes to the data.

If we make both properties and methods public, data can be accessed by any code or function which is not protecting the data.

6)

Polymorphism:

Polymorphism is when you apply the same command to multiple object but each object results in a different outcome. This is achieved by forming relationships between objects.

5)

Inheritance:

The point of object oriented programming is to simulate the real world. Now in real world we have objects but one thing that all those objects have in common is a relationship. That relationship is simulated using inheritance. Some objects can inherit properties from other objects like a child class inheriting properties from a parent class but building upon those properties.

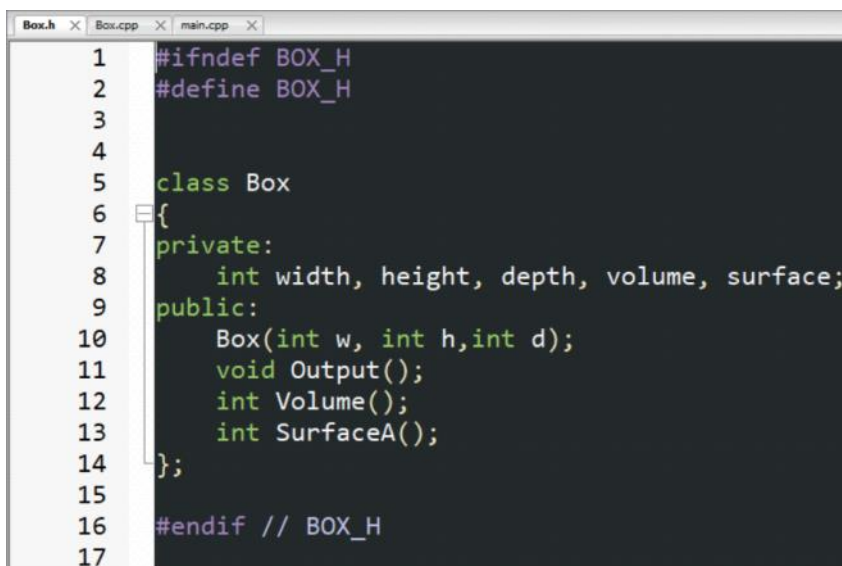
Classes in C++

Sunday, April 1, 2018 12:57 PM

Creating classes in C++ is very easy.

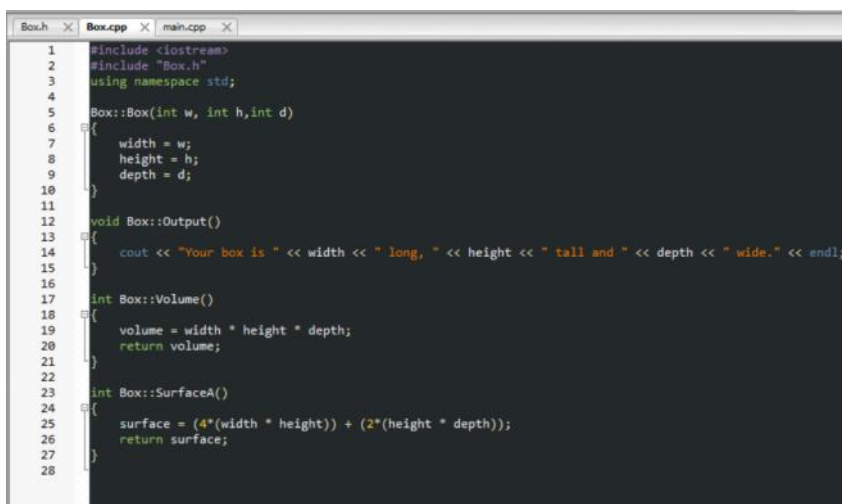
In C++ you can have classes in your main .cpp file or you can have classes in separate files (normal way of doing it). A class consists of a header file and a separate .cpp file. **The main.cpp file is where your objects are made and your class members are called. The header file is where your class is defined and so are your private and public properties. Your functions and members are declared in the header file but defined in the .cpp file.**

Example of Class in C++:

A screenshot of a code editor showing the Box.h header file. The code defines a class Box with private attributes width, height, depth, volume, and surface. It also declares public methods: a constructor Box(int w, int h, int d), and three other methods: Output(), Volume(), and SurfaceA(). The code is enclosed in a preprocessor guard #ifndef BOX_H to #endif.

```
1 #ifndef BOX_H
2 #define BOX_H
3
4
5 class Box
6 {
7 private:
8     int width, height, depth, volume, surface;
9 public:
10     Box(int w, int h, int d);
11     void Output();
12     int Volume();
13     int SurfaceA();
14 };
15
16 #endif // BOX_H
17
```

Here is the header file containing the declaration of the class. It creates and says that these functions exist. This file is then pasted onto the main.cpp and Box.cpp.

A screenshot of a code editor showing the Box.cpp implementation file. It includes the necessary headers <iostream> and "Box.h", and uses the std namespace. The implementation defines the constructor, Output() method (which prints box details), Volume() method (which calculates and returns volume), and SurfaceA() method (which calculates and returns surface area).

```
1 #include <iostream>
2 #include "Box.h"
3 using namespace std;
4
5 Box::Box(int w, int h, int d)
6 {
7     width = w;
8     height = h;
9     depth = d;
10 }
11
12 void Box::Output()
13 {
14     cout << "Your box is " << width << " long, " << height << " tall and " << depth << " wide." << endl;
15 }
16
17 int Box::Volume()
18 {
19     volume = width * height * depth;
20     return volume;
21 }
22
23 int Box::SurfaceA()
24 {
25     surface = (4*(width * height)) + (2*(height * depth));
26     return surface;
27 }
28
```

Here is the Box.cpp. This is where the functions or behaviors of our object is made. We use Box::function because it needs to know that this function belongs to the class Box. It's like doing the same thing if you don't declare using namespace std; you have to manually put std::cout whenever you want to use those functions.

The "Box" function is used whenever you declare a new object. It's used to fill in the blanks

```
4 using namespace std;
5
6 int main()
7 {
8     Box b1(10,10,10);
9
10    b1.Output();
11    cout << b1.SurfaceA();
12    return 0;
13 }
14
```

This is the class being used to create a new object called b1. it's like the same as using int num to make a number variable. We then use the output function in our class on b1 and output the surface area.

Constructors and Destructors

Sunday, April 1, 2018 5:59 PM

Instantiation: when an object is made from a class.

Constructors are ran when the class is instantiated. Let's take the box class as the example. We need a constructor to declare what the values of each dimension is (width, height, depth). Since we want our data to be encapsulated and private, we have to change the values using a function in the class. This is an example of a constructor of a class called Box.

```
Box::Box(int w, int h, int d)
{
    width = w;
    height = h;
    depth = d;
}
```

Here we have declared that when an object of type Box is made, take in three parameters for width, height and depth and declare those values as their respective variables. Constructors do not have any return type as they are constructors.

YOU CAN HAVE MORE THAN ONE CONSTRUCTOR.

We know that constructors have the same name as the class therefore **multiple constructors will have the same name but the difference would be in the parameters.** One constructor could have no parameters and can be used to allocate default values to an object while another constructor can take in values from the code. The compiler will choose the correct constructor depending on the parameters given during the instantiation of the object.

Destructors are ran at the end of the scope of the object.

Taking the same box class as an example, the box object is created by the constructor when instantiated. The object could be instantiated in the main function, in a loop or anywhere between a pair of {}. At the end of its scope (}), the object is destroyed to save space. This is done by the **destructor**. You can set what the destructor does when it runs by creating a function with ~ in front of your class name. like ~Box() for our example.

GENERAL RULE:

You can have multiple methods with the same name as long as the parameters are different. You can't however have the exact same method but with different return type.

String Streams

Sunday, April 1, 2018 11:22 PM

Let's say we want to make a string variable describing the status of someone (name and age). This would involve combining multiple different data types with a string (int, float). **A normal string variable can't** do this so we need to use StringStreams.

This variable type needs the **<sstream> header file**.

After declaring a stringstream variable you can **increment data to it using variablename << data**.

When you want to output this data or convert it back to a string variable you can use variablename.str(); which will **convert it back to string**.

Example:

```
#include <iostream>
#include <sstream>

using namespace std;

int main()
{
    double pi = 3.14159265;
    stringstream ss;
    ss << "The value of pi is: " << pi;

    cout << ss.str();
    return 0;
}
```

Result:

The value of pi is: 3.14159265

We've successfully incremented a double onto a string.

"This" Keyword

Monday, April 2, 2018 12:08 PM

Let's say you've made a constructor that takes in the name and age of a person. You could do it like this:

```
Person(string name, int age)
```

That would be a massive problem if your attributes that you are replacing are named the same thing:

```
Person(string name, int age)
    name = name;
    age = age;
```

The Blue variables only exist in the {} of the method. The green variables exist in the class.

The blue variables are in a different location on the memory compared to the green variables so we can use that to differentiate them.

Using the **"this"** operator we can refer to the variable at a this particular memory location so we get the green variable instead of the blue.

To use this keyword all we have to do is:

```
Person(string name, int age)
    this->name = name;
    this->age = age;
```

Initialization List

Monday, April 2, 2018 7:59 PM

We know we can use the "this" keyword if we want to use the same name for our parameters as our attributes. There is an easier way of doing it and that is using **Initialization Lists**. These basically allow you to easily set attribute values in the constructor.

Here is an example:

```
#include "Person.h"

Person::Person(string name, int age): name(name), age(age){}
```

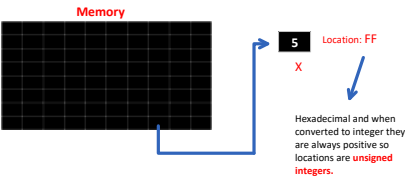
2 things:

- You need a **colon after the closing brackets** of the arguments. The variable outside the bracket is the **private variable of the class** whereas the variable inside the bracket is the **parameter taken in**.
- You might not have any code after that so you want to keep everything on one line but **you have to add a {} at the end**.

Let's take a simple line of code:

Int X = 5;
This declares the variable type This is the name of the variable This is the value that is stored in the variable

What does this mean?
It means that a new variable is created in the memory of the computer. We know that integers take up 4 bytes of memory so 4 bytes is allocated to the variable X and the value 5 is stored in it.



If we want to output the value of variable X we just do: `cout << X;`
If we want to output the address of X we can use the address operator: `&X`

Pointer Variables:

Pointer variables hold the memory address of another variable that contains data. They could hold the memory address of another pointer variable which holds the address of another variable that contains data

Example of a pointer variable:

Int* pX = &X;
The star means that it is a pointer variable and that it is storing the address of an integer variable This is the name of the pointer variable This means the address of variable X

This doesn't have to work with ints only. It can work with char and float and others.

If we want to create a pointer variable that holds the address to another pointer variable we must use:

Int**

The blue star means that it is a pointer variable and the red int* means that it is storing the address of a pointer variable.

Indirection operator:

Now that we can store the address of variables in pointers, we can use those pointers to get the data in the said address.

Let's take the previous example:

Int X = 5;
Int* pX = &X;
Cout << *pX;
Result: 5

We have declared a variable at a memory location with the value 5 in it.

We have stored the value of X in a pointer variable.

Using the **indirection operator** we can get the value at the address inside pX.

*

This is the value at address operator which goes to the given address of a pointer and gets the value

NOTE:

Int* ≠ *pX

The * and indirection operator (*) are not the same. One indicates that this variable is a pointer while the other gets data from a pointer's address.

EXAMPLE OF POINTERS BEING USED TO CHANGE THE VALUE OF A VARIABLE INSIDE A FINCTION:

```
#include <iostream>
using namespace std;

void CORRECT_PI(float* address)
{
    *address = 3.1415;
}

int main()
{
    float pi = 123.4;
    CORRECT_PI(&pi);
    cout << "Real value of pi is: " << pi;
    return 0;
}
```

- 3) This function returns no value but it takes the address of a variable and **manipulates the data in that address** so it affects the data outside the **scope** of the function
- 1) A fake value for pi is set
- 2) We input the address of pi into the function
- 3) The function accesses the value at that address and changes it.
- 4) Now the value of pi has been changed **without returning a variable.**

If you try to output an array by itself (cout << array;) all you will get is hex address. This is because arrays by themselves are kind of like pointers. They hold the address to the first element in the array. So if we create a pointer for an array we don't need to use the & operator because:

Int* pArray = &Array; = Int* pArray = Array

Now this pointer has the address of the first element of the array.
If we output the value at the pointer we just get the first element.
We can do arithmetic on pointers just like any other number variables.
If we increment the pointer, it is now pointing to the next element in the array.

Int* pArray = Array;

| | | | |
|----------|--|-----|----------|
| pArray | | --> | Element1 |
| pArray++ | | --> | Element2 |
| pArray++ | | --> | Element3 |

WARNING: trying to output the value of pointer after it's been incremented above the limit of the array can cause your program to crash

This can be used in for loops and while loops too:

```
#include <iostream>

using namespace std;

int main()
{
    string Array[] = {"hi", "one", "Two", "5"};
    string* pArray = Array;
    string* pnArray = &Array[3];

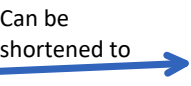
    for (int i = 0; pArray <= pnArray; pArray++)
    {
        cout << *pArray << endl;
    }

    return 0;
}
```

- 1) Here we have declared an array with 4 elements.
- 2) Here we have declared a pointer which points to the first element in the array
- 3) Here we have declared a pointer which points to the last element in the array. This is our limit. **3 because first is 0.**
- 4) Now we can use these two in our for loop. **pArray is the counter** while **pnArray is the limit**. After each round **address of pArray is incremented**. This loop outputs the value at the current address.

We know that we can **use strings to store words and sentences**. We know that **chars are one byte variables that only store a single character**. We also know that **strings are basically arrays and that we can use array commands to call for a single character from the string**.

An array of char can also be very similar to a string. It is an array of characters which is identical to a string:

Char Array[] = {"M","a","h","a","n"}  Char Array[] = "Mahan"

If we use a for loop to output each character as an ascii value we get a surprising result:

```
for (int i=0; i < sizeof(Array)/sizeof(char); i++)  
{  
    cout << i << ": " << Array[i] << endl;  
}
```

Result:

```
0: 77  
1: 97  
2: 104  
3: 97  
4: 110  
5: 0
```

We have done this because we are assuming that we do not know the number of elements in the array. Doing this returns a value of 6 although there are only 5 elements in the array...

This is because an array of characters has a **null terminator**. This is because a normal string does not require an ending point. It knows when the array is finished. The char array however (also known as a **null terminated string**) needs an ascii value of 0 (null) to know that the array has ended.



A reference is another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

Differences between a pointer and a reference:

- You cannot have an empty reference, a reference must always point to a piece of storage while a pointer can initially be empty
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

Because
it can't be
empty

References in C++:

A variable name is a label given to a specific memory location. A reference is a second label given to the same location.

If we declare a variable like this:

`Int Num = 5;` ← First label given to a memory location

We can create a reference like this:

`Int& rNum = Num;` ← Second label given to a memory location

Example use of a reference:

```
#include <iostream>

using namespace std;

void FixPi(double& Ref)
{
    Ref = 3.1415;
}

int main ()
{
    double Pi = 123.4;
    double& rPi = Pi;
    cout << Pi << endl;
    FixPi(rPi);
    cout << Pi;
    |
    return 0;
}
```

In this function we are changing the value of a variable **outside the function**.

This is because we are having a reference as a parameter therefore when we use this reference it is calling to a specific memory location which is where our variable outside the function is.

A reference is made using the & operator **(not the confused with the pointer & operator which gives the address)**.

We know that we can use variables to store data. **These variables can be varied** (hence vary-able) but sometimes you have data that you do not want to change, we have to use constant variables.

Constant variables **cannot be changed** and you **get errors if you try to**.

To create a constant variable you use the same syntax as before but you add a **"const"** in front of it:

```
const double Pi = 3.1415;
```

Let's say we have a class called "Animal", it has one attribute called name and two methods called "setName" and "speak" which will say its name.

```
class Animal {
private:
    string name;
public:
    void setName(string name) { this->name = name; };
    void speak() const
    {
        cout << "My name is: " << name << endl;
    }
};
```

We have made the speak method constant because **it does not modify any attributes** therefore it won't change anything. Const can not only be used on variables but **methods as well**.

Const and pointers:

We can make a pointer constant or the value at that pointer constant.

Here we have two ways of making a pointer constant.

If we read from the equal sign backwards we can understand what is constant.

```
int value = 8;

// const int* pValue = &value; Here the pointer's value is constant as const is at the front of the line
// int* const pValue = &value; Here the pointer's address is constant as const is after the int*
// const int* const pValue = &value Here both the pointer's address and the value at that address is constant so both errors will occur

cout << *pValue << endl;

int number = 11;
pValue = &number; // Error with this: int* const pValue = &value;
*pValue = 12; // Error with this: const int* pValue = &value;

cout << *pValue << endl;

return 0;
```

Copy constructor

Sunday, April 8, 2018 12:56 PM

When you **instantiate** an object the **constructor** is used. You can also instantiate an object from another object. Like having copies:

```
Animal cat1;  
Animal cat2 = cat1;
```

When you **instantiate by copying** another object, the **copy constructor is called** instead of the default constructor.

We can write this copy constructor ourselves. In the header file of our class we must declare this constructor as a function first. It does not have any return types but it does have parameters which make it a copy constructor.

```
class Cat  
{  
    public:  
        Cat();  
        ~Cat();  
        Cat(const Cat& |other);  
}
```

It has the same name as the constructor but the parameters make it a copy constructor.

For the parameters you just need "const", **class reference(&)** and an **object reference** which can be anything.

```
Cat(const Cat& oldcat)  
{  
    Name = oldcat.Name;  
    Age = oldcat.Age;  
}
```

This copy constructor just copies the name and age attributes from the copy object (oldcat) onto the new object.

This is why we put the const at the beginning because we are just "copying" attributes and not changing them.

The "new" operator

Thursday, April 12, 2018 4:21 PM

To make an object from a class we just use the same syntax as we would when making variables:

```
int main ()
{
    Cat cat1("Mahan", 5);
    cat1.Speak();

    return 0;
}
```

Here an object of class "Cat" is instantiated using two parameters

FOR REFERENCE:

```
class Cat
{
public:
    Cat(string name, int age);
    ~Cat();
    void Speak();

protected:

private:
    string name;
    int age;
};
```

There is another way we can use to create using the **new** keyword and pointers:

```
int main ()
{
    Cat* pCat1 = new Cat("Mahan", 5);
    pCat1.Speak();

    return 0;
}
```

pCat1 is a pointer that holds a "new" object of class Cat.

This means that we can't just do pCat1.Speak() because pCat1 is just a memory address.

```
Cat::Cat(string name, int age)
{
    cout << "cat created" << endl;
    this->name = name;
    this->age = age;
}

Cat::~~Cat()
{
    cout << "cat destroyed" << endl;
}

void Cat::Speak()
{
    cout << "My name is " << name << " and i am " << age << endl;
}
```

Trying to dereferencing the pointer also won't work:

```
int main ()
{
    Cat* pCat1 = new Cat("Mahan", 5);
    *pCat1.Speak();

    return 0;
}
```

This also doesn't work because **.Speak is done before the pointer is dereferenced.** Putting *pCat1 in brackets does work however there is a better way to doing it

We can use this method to access methods of a class using a pointer:

```
int main ()
{
    Cat* pCat1 = new Cat("Mahan", 5);
    pCat1->Speak();

    return 0;
}
```

Using the -> method instead of a . Means that **we don't need to dereference the pointer** to do access the methods.

As you can see in the result, the constructor is called when the object is made but the **destructor is not called** at the end of the program.

When objects are made using points you **manually need to destroy** them or else they will remain on your RAM even after the program is closed:

```
int main ()
{
    Cat* pCat1 = new Cat("Mahan", 5);
    pCat1->Speak();
    delete pCat1;

    return 0;
}
```

Here we can see after we **"delete"** the pointer the destructor is called and the memory is deallocated.

Returning objects from a method is quite common in C++.
To create a function that returns a method we use the same

Return Type Name of the Function (Parameters)

This means that if we want to make a function that returns an object with class type of "Animal" we must do this:

Animal createAnimal (Parameters)

```
Eg:
Animal createAnimal()
{
    Animal a;
    a.setName("Hi")
    return a;
}

int main()
{
    Animal Ani = createAnimal();
    Ani.speak()
}
```

<-- Our function that makes the object

<-- Our main function where the object is returned to

Why is this inefficient?

Let's see what happens when the code is ran:

- Constructor creates the object "a"
- Sets the name as "hi"
- **Uses the copy constructor to put the object on a temporary object and returns it**
- **Because we are using the equal sign the copy constructor is used again to copy everything from the temporary object onto "Ani"**
- Destructor is called when object goes out of scope

<-- This is why it is inefficient however some new compilers have learned to optimize this step so it isn't inefficient but it is good to know how to get around this

How to fix this?

Using **POINTERS**
If we change some stuff to make this we make it a lot more efficient:

```
Animal* createAnimal()
{
    Animal* pAnimal = new Animal();
    pAnimal->setName("Bertie");
    return pAnimal;
}

int main()
{
    Animal* pFrog = createAnimal();

    pFrog->speak();

    delete pFrog;

    return 0;
}
```

- How is this more efficient?
1. The function is now returning a pointer instead of an object
 2. A new object of type Animal is made at the memory location "pAnimal"
 3. The memory location (A hex value) is returned instead of an object
 4. The pointer variable "pFrog" is set to the returned memory location
 5. The pointer is used to access methods of the object

Allocating Memory

Monday, July 16, 2018 4:37 PM

NOTE: when using the new operator you must make sure to use the delete operator to deallocate the memory used otherwise this will result in memory leaks

We have use the "new" operator before which makes a new object at a specified memory location.

We can use the new operator on any data type so we can allocate memory for a data type but leave the value empty:

```
int main()
{
    int* pInt = new int;
    *pInt = 50;
    cout << *pInt;
}
```

Int* declares a pointer named pInt which points to a memory location allocated to an integer. This memory is left aside for an integer to occupy.

We can allocate entire arrays so for example if we want to allocate memory for 5 objects of the same class we can do:

```
int main()
{
    Animal* pAni = new Animal[5];

    delete [] pAni;
}
```

```
Animal created.
Animal created.
Animal created.
Animal created.
Animal created.
Destructor called
Destructor called
Destructor called
Destructor called
Destructor called
```

Using the square brackets we can indicate how many of the object or data type we want to allocate memory to therefore the constructor of the class is called that many times .

We also have to delete all that data which can be done by a set of empty square brackets which deletes an entire block of data

Passing arrays into functions:

Syntax:

ReturnType **FunctionName**(ArrayType **ArrayName**[])

One thing to note is that when an array is passed through a function, **it is passed as a pointer** and not a variable. This is why we cannot use sizeof to determine the array length

If we want to use a function that uses the array's length we need to pass that as one of our parameters:

```
void PrintArray(string Array[], int n)
{
    for (int i=0; i<n; i++)
    {
        cout << Array[i] << endl;
    }
}
```

Int n is the length of the array which is passed through as one of the parameters

Returning arrays from a function

We cannot return arrays as a variable therefore we must return a pointer.

We can set our return type of our function to be a pointer like so:

```
string* MakeArray()
{
    string* pArray;
    string Array[] = {"1", "2", "3"};
    pArray = &Array;
    return pArray;
}
```

In the function, an array is made, a pointer of that variable is made and is returned.

BUT THERE IS A BIG PROBLEM

We can't forget about variable scopes. After the function is done the variable goes out of scope and is deleted. What we need to do is use the **NEW** operator which does not get deleted after the curly brackets and is only deleted using the **DELETE** operator.

Here we have a function that creates an array and returns it.

```
char* MakeArray()
{
    char* pArray = new char[3];
    int j = 0;
    for (int i = 97; i<100; i++)
    {
        pArray[j] = i;
        j++;
    }
    return pArray;
}

int main()
{
    char* pNew = MakeArray();
    PrintArray(pNew, 3);
    delete pNew;
    return 0;
}
```

Using a pointer, a new char array is created using the **new** operator

The char array is filled with values
The pointer is returned.

The pointer is usable since the variable is not deleted when out of scope.
The array is printed on the screen and then **IS DELETED**

Namespaces

Thursday, July 19, 2018 12:44 PM

Namespaces are a way of avoiding conflict between classes or variables with the same name. We have already used a namespace, the standard (std) namespace which allows us to use cout.

What if we want to have another class with the same name as another class?

An example of a class in a namespace:

```
#include <iostream>
#include "Animal.h"

using namespace std;

namespace Mahan
{
    Animal::Animal()
    {
        cout << "constructor called" << endl;
        Age = 0;
    }

    Animal::Animal(int a)
    {
        cout << "constructor called" << endl;
        Age = a;
    }

    void Animal::SetAge(int b)
    {
        this->Age = b;
    }

    void Animal::Speak()
    {
        cout << "Age: " << Age << endl;
    }

    Animal::~~Animal()
    {
        cout << "destructor called" << endl;
    }
}
```

```

#include <iostream>
#include "Animal.h"

using namespace std;
using namespace Mahan;

int main()
{
    Animal Mahan(17);
    Mahan.Speak();
    return 0;
}

```

```

#ifndef ANIMAL_H
#define ANIMAL_H

namespace Mahan
{
    class Animal
    {
    public:
        Animal();
        Animal(int a);
        void SetAge(int b);
        void Speak();
        ~Animal();

    protected:

    private:
        int Age;
    };
};
#endif // ANIMAL_H

```

We don't always use the code that we wrote in our programs. Most of the things we do like cout are functions written by someone else. These are in libraries that we can implement into our programs to use. There are two types of libraries:

Static Libraries:

Static libraries are code that the compiler puts into the program before compiling. The code is bundled into an executable and is ready to run. It is actually compiled into the program. They have the suffix of .lib

Dynamic Libraries:

Dynamic libraries are code that the executable can look for and use when it is running. The program finds the code at runtime. They have the suffix of .dll

Signals and Slots are a way of getting input from the user (button, text box), processing it and then outputting it (Label)

A signal is an event caused by an object like the event of a button being clicked.

A receiver is an object whose slot is affected by the signal

A slot is an attribute of the receiver that is altered when a signal is triggered.

There are 3 ways of doing this:

The Graphical way:

By clicking on "Edit Signals/Slots" icon on the top of the screen we can begin to edit and make IO.

We can click and drag from our input to our output and then we are prompted to choose a signal and a slot.

This will only allow you to change the attribute of one object depending on what happens to the signal object, no other objects are involved so you can't take attributes from other objects.

The "connect" statement:

In our mainwindow.cpp file we can add a connect statement in the constructor of the class. The basic structure of the connect statement is:

```
connect(signal_object,SIGNAL(object_event(type)),receiver_object  
        ,SLOT(receiver_attribute(type));  
        (functions)(Statement)(Keyword)(Object)
```

This method also has the same problem as the graphical method.

The Event method way:

This is the best way to make signals and slots because you have all the freedom to use other objects, functions and processing. This basically creates a method in the mainwindow class. To make this function right click on your signal object on the UI editor, click on go to slot and choose the desired signal. QT will make a new event function in the header and cpp file of mainwindow class


```
void MainWindow::on_lineEdit_textChanged(const QString &arg1)
{
    ui->label->setText(ui->lineEdit->text());
}
```

IF YOU WANT TO REMOVE THE METHOD YOU MUST REMOVE IT FROM BOTH HEADER AND CPP FILE

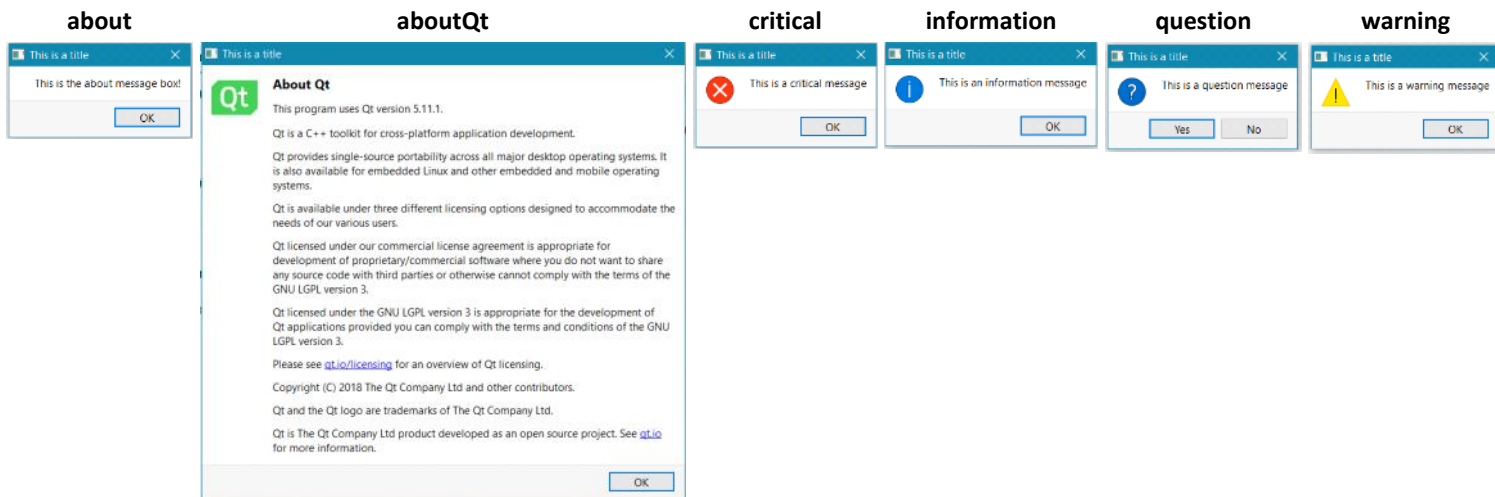
QT Message Boxes are a class in which a message box of different types can be displayed.
To use this class we need to include it in our header file
`#include <QMessageBox>`

Then we can use message boxes in our event method.
The general structure of a message box is like this:

```
QMessageBox::box_type(parent_widget, "Title", "Message");
```

Usually the parent widget is the "this" keyword.

Different Types of message boxes:



The question box:

The question box can be used to ask the user to make a decision and according to their decision, make an action.

So far we have only dealt with 3 arguments when calling a message box. There is an optional 4th one:

```
QMessageBox::question(this, "This is a title", "Do you want to quit?", QMessageBox::Yes | QMessageBox::No);
```

This returns a "StandardButton" object which can be either yes or no.

We can use this in an if statement to do actions based on the user's choice but to do this we need to store that choice in an object.

```
QMessageBox::StandardButton Choice = QMessageBox::question(this,
    "This is a title",
    "Do you want to quit?",
    QMessageBox::Yes | QMessageBox::No);
```

```
if (Choice == QMessageBox::Yes)
{
    QApplication::quit();
}
```

The return type of the message box is a "StandardButton" so we need to make an object of that type to store the choice. Then to check that choice in an if statement we need to use QMessageBox to see if it's a yes or a no.

QString is somewhat similar to `std::string`.
You can pick specific places in the string and it works as an array.

Using a certain character from QString:

```
QString name = "MAHAN"      Result:  
QChar letter = name.at(2)   Letter = "H"
```

This puts the 3rd character into char variable "letter"

Getting length of a QString:

```
QString name = "MAHAN";  
int len = name.length();
```

This puts the number 5 into variable "len"

Converting from int to QChar:

```
int ascii = 72;  
QChar chr((short) ascii)
```

This stores the character H in variable "chr"

Appending into a QString object:

```
QString emptystr;  
emptystr.append("A")
```

Getting ascii number character from QChar:

```
QChar chr = "H";  
int ascii = chr.toLatin1();
```

This puts the number 72 (ascii code for H) in int variable "ascii"

Multiple Windows

Thursday, August 16, 2018 12:46 AM

To use multiple forms we can show a second form when something happens.

After including the header file for the ui form, make a new object from the form class:

```
NewWindow* winD = new NewWindow;  
winD->show();  
hide();|
```

Its best to make a pointer and then make a new instance of the form class. Then use the method show to open the window. Use hide to close the current window.

Deploying QT applications

Monday, August 6, 2018 11:25 PM

- Build the program for release
- Delete all files in release folder except the EXE file
- Open QT cmd from windows menu under compiler folder
- type "windeployqt.exe"
- type cd <path to release folder>
- type "windeployqt.exe ."

RSA

Sunday, December 16, 2018 8:24 PM

RSA is a well known method of encryption. It is an algorithm that mainly uses prime numbers, public and private keys.

RSA uses two keys, the Public Key (En_Key, N) and the Private Key (De_Key, N) N is the modulus in both keys and is the product of two prime numbers used in the algorithm.

Steps to calculate the keys:

1. Choose any two prime numbers, p and q;
2. Where N, is the product of p and q, $N = pq$;
3. Calculate ΦN using, $\Phi N = (p-1)(q-1)$;
4. To pick En_Key or e we have to follow a set of rules:
 - a. $1 < e < \Phi N$;
 - b. Must be co-prime with ΦN and N;
5. To pick De_Key or d we have to follow another rule:
 - a. $d * e \pmod{\Phi N} = 1$

Why not RSA

Sunday, December 16, 2018

8:52 PM

I initially calculated my own keys using the prime numbers $P = 131$ and $q = 151$. This led me to calculating the values of N and ΦN to be 19781 and 19500 respectively.

Using my own python program, I calculated the values of e and d using the conditions.

```
from fractions import gcd
Counter = 0
for Loop in range (2,12):
    if gcd(Loop,12) == 1:
        print (Loop)

for Loop in range (1,1000):
    if ((Loop*7)%12) == 1:
        print (Loop)

from fractions import gcd

print(gcd(3587,19781))
a = 126**25
b = a%161
c = b**37
d = c%161
print (b)
print (d)
```

This gave me the value of e to be 3451 and d to be 59551

These values were too big for C++. No data type was able to calculate an ascii value to the power of e or d .

The results of these calculations resulted in overflow which made this method of encryption unsuitable.

I tried to choose much smaller prime numbers to get smaller values of e , d and N .

Although I was able to get much smaller values of e and N , the value of d got bigger as value of e got smaller which meant I could not decrypt the text file due to overflow of calculation results.

Finding how much memory is allocated to each variable type:

```
#include <iostream>
Using namespace std;
Int main()
{
    Cout << sizeof(int);      4
    Cout << sizeof(float);    4
    Cout << sizeof(double);   8
    Cout << sizeof(string);   24
}
```

Limits to values of an integer:
Since there are only 4 bytes of memory allocated to each integer variable, the maximum number of an integer value can be given using:

```
#include <iostream>
#include <limits.h>

Using namespace std;

Int main()
{
    Cout << INT_MAX;      2147483647
    Cout << INT_MIN;      -2147483648
}
```

Changing the way very large numbers are represented:

Normal (760000):

Use the normal std namespace;

Cout << fixed << variable;

Scientific (7.6 * 10^5):

Use normal std namespace;

Cout << scientific << variable;

Reversing an array using a function:
Using pointers and recursion you can reverse an array.

```
##REPLACE char* WITH ANY OTHER DATA TYPE##

void Reverse(char* Pointer)
{
    if (*Pointer != 0)
    {
        Pointer++;
        Reverse(Pointer);
        cout << *(Pointer - 1) ;
    }
}
```

Calculating a number to the power of an exponent:

This function needs the math.h header file.

Use:

```
#include <math.h>

Cout << pow(base,exponent);
```

Eg:

```
Cout << pow(2,4);
Result: 16
```

Set the decimal precision of a float or double:

This function uses the <iomanip> header file.

Use:

```
Setprecision(no of decimal places)
```

Eg:

```
#include <iomanip>

double num = 76.41231231111;
cout << setprecision(1) << fixed << num;
```

You need to use the fixed function for this to work

Getting the length of an array

This function uses the <string> header file which should already be in your program if you have a string.

Use:

```
NameOfString.length()
```

Eg:

```
#include <string>

String MyName = "Mahan";
Cout<<MyName.length();
```

Result:

5

EPQ REFERENCE

Wednesday, 25 October, 2017 7:15 PM

Websites used to expand knowledge on C++:

[C++ Tutorials](#)

[Very in-depth information about C++](#)

Learning about Object Oriented Programming:

[Computer programming: What is object-oriented language? | lynda.com overview](#)

[Object Oriented Programming Concepts](#)

C++ tutorials on YouTube:

[Let's Learn C++](#)

C++ online course on Udemy by John Purcell:

[C++ Tutorial for Complete Beginners](#)

[Learn Advanced C++ Programming](#)

Learning the QT framework:

[Qt Tutorials For Beginners \(YouTube Tutorial\)](#)

[Beginner's Qt GUI Programming \(Udemy Course\)](#)

[QT Documentation](#)

Books:

[Jumping Into C++ by Alex Allain](#)

[Accelerated C++ Practical Programming By Example](#)

RSA Encryption Algorithm (Eddie Woo):

[The RSA Encryption Algorithm](#)

[The RSA Generating Keys](#)