C++ Tutorials by The New Boston

Contents

1.1 Running C++ file in google Collaboratory. 6

1.2. How to run a C^{++} programme in windows powershell? 7

2.What is compiler?..8

3. Understanding a simple C++ code 9

4. More about printing text..10

5. Variables...11

5.1 Creating a basic calculator using variables...........................12

5.2 Overwriting the value of a variable. 13

6. Basic Arithmetic..14

7. if statement 14

8. Functions 15

8.1 Void functions 15

8.2 Function Prototyping...16

8.3 Creating functions that uses parameter / argument...............17

8.4 Creating functions that uses multiple parameters / arguments. 17

9. Classes and Objects 18

10. Using variables in classes...19

10.1 Wrong way (As per YouTuber) 19

10.2 Correct Way. 20

11. Constructors...21

11.1 Example 1...21

11.2. Example 2..22

12. Placing classes in a separate file 23

13. If statement 26

14. If-else statement 27

14.1 Nested ‘if’ statement. 27

15. While loop 28
15.1 Make an infinite while loop to finite....................28
15.2 Take 5 numbers entered by the user and get the sum using While loop. 29
16. Sentinel Controlled program 30
17. Assignment and increment operators 32
17.1 C++ works without ‘return 0’! 32
17.2. Example with increment operator. 33
17.3 Difference between x++ and ++x 33
18. For Loop 35
18.1 Making a stock market simulator using ‘for’ loop (Compound Interest Problem) 35
18.2 What is the use of ‘#include <cmath>’? 36
19. do while loop 37
20. Switch statement....................38
21. Logical Operators 40
21.1. Example....................40
21.2 Boolean Variables. 41
22. Random Number Generator 42
23. Default Arguments/Parameters 44
24. Unary Scope Resolution Operator 45
25. What is the difference between int and void functions?....................47
26. Function Overloading....................48
27. Recursion....................49
27.1 Bad way to use recursive function. 49
27.2. Good way to use recursive function: Finding the factorial of a number....................50
28. Arrays 50

28.1 Creating an array using loops....51
28.2 Using Arrays in calculations....52
28.3 Passing Arrays to functions....53
29. Multi-Dimensional Array....54
29.1. How to print a multi-dimensional array....54
30. Pointers....55
30.1 Pass by references with pointers....55
31. Size of function....57
32. Pointers and Math....58
33. Arrow Member selection Operator....59
34. Deconstructors / Destructors....62
35. const Objects....64
35.1 Regular variable....64
35.2 Constant Variable:....64
35.3 How to call constant function using constant object....65
36. Member Initializers....67
36.1. What is the meaning of first two lines of a header file?....69
#ifndef SALLY_H....69
#define SALLY_H....69
36.2 Why Sally.h becomes SALLY_H here?....69
37. Composition (Do revise!)....70
37.1. What is the difference between ‘bo’, ‘dateOfBirth’ and ‘birthObject’ here?....73
38. Friend....75
39. this....76
40. Operator Overloading....78
40.1 Explanation of operator overloading part....80
41. Inheritance....80

42. Protected Members....83
43. Derived class Constructors and Destructors 86
44. Polymorphism 89
45. Abstract Classes and pure Virtual Functions 92
46. Function Templates....94
46.1. what is the difference between ‘double’ and ‘float’ datatypes? 95
46.2. Function Templates with Multiple Parameters....96
47. Class Templates....97
48. Template Specializations....99
48. Exceptions Handling 100
49. Working with files.... 102
49.1 Tips for File Handling.... 103
49.2 Writing Custom File Structures 104
49.3. Reading Custom File Structures 106
49.4. Cool program working with file.... 107
50. String Class and Functions.... 110
50.1. Print a user input string (one word) 110
50.2. Print a user input string (entire line) 111
50.3 Copy Strings.... 112
50.4. Strings are Arrays! 113
50.5 Substring Functions.... 113
50.6 swap function 114
50.7 Finding a substring in a string.... 115
50.8 String erase, replace and insert functions 116

C:MinGW => Where is this downloaded

1.1 Running C++ file in google Collaboratory.

```
# Write the C++ code to a file
cpp_code = """
#include <iostream>

int main() {
    std::cout << "Hello, Google Colab with C++!" <<
std::endl;
    return 0;
}
"""

with open("main.cpp", "w") as file:
    file.write(cpp_code)

# Compile and run the C++ code
!g++ main.cpp -o main
!./main
```

```
⇨ Hello, Google Colab with C++!
```

The code you provided is for writing a simple C++ program to a file, compiling it using the g++ compiler, and then running the compiled program. This code assumes that you are using a Linux-based environment or a similar setup that supports the g++ compiler. This code does the following:

- It defines a C++ program inside the `cpp_code` string.
- It opens a file named "main.cpp" and writes the C++ code into that file.
- It then compiles the "main.cpp" file using the `g++` compiler, generating an executable named "main".
- Finally, it runs the compiled program using `!./main`.

Make sure you have the `g++` compiler installed and configured in your environment to execute this code successfully.

1.2. How to run a C^{++} programme in windows

To run a C++ program named **tutorial.cpp** in Windows PowerShell, you'll need to compile the source code into an executable file and then execute that file. Here are the steps:

1. **Compile the C++ program:** Use a C++ compiler, such as **g++** from MinGW or Visual C++'s **cl**, to compile your **tutorial.cpp** file. Open PowerShell and navigate to the directory where your **tutorial.cpp** file is located. Then, compile it with a command similar to this:

For MinGW's g++:

```
g++ -o tutorial.exe tutorial.cpp
```

For Visual C++'s cl:

cl /EHsc tutorial.cpp

This will create an executable file named **tutorial.exe**.

2. **Run the compiled executable:** Once you've successfully compiled your program, execute it in PowerShell using this command:

```
.\tutorial.exe
```

This assumes you're working with the **tutorial.cpp** file and have the necessary C++ compiler (like MinGW or Visual C++) installed and correctly configured in your system's environment variables.

Make sure you are in the correct directory within PowerShell where **tutorial.cpp** is located, or provide the full path to the file in the compilation command if it's in a different directory.

Remember, the specific commands might differ based on the compiler you're using and the actual name and location of your source code. Adjust the commands accordingly to match your specific environment and file names.

2.What is compiler?

A compiler is a software program that translates code written in a *high-level programming language (such as C, Java, or Python)* into a low-level language that can be directly executed by a computer's CPU (central processing unit). This low-level language is typically machine code, which is a series of binary instructions that the CPU can understand.

The compilation process typically involves several steps:

Lexical analysis: The compiler breaks the source code into a sequence of tokens, which are basic units of the language such as keywords, identifiers, and operators.

Syntax analysis: The compiler analyses the sequence of tokens to determine the *grammatical structure* of the program.

Semantic analysis: This phase checks for semantic errors and enforces language-specific rules. It ensures that variables are declared before use, data types are used correctly, and functions are called with the correct arguments.

Intermediate code generation: The compiler generates an intermediate representation of the program, which is a more machine-friendly form of the program.

Code optimization: Compilers can perform various optimization techniques to improve the efficiency and speed of the generated code. Common optimizations include constant folding, loop unrolling, and dead code elimination.

Code generation: The compiler generates machine code from the intermediate code.

Linking: In the case of larger programs or when using multiple source files, the compiler may need to link various object files and libraries to create the final executable program.

Once the compilation process is complete, the resulting machine code can be executed by the CPU.

Compilers are essential tools for software development. They allow programmers to write code in high-level languages that are easier for humans to understand and maintain. The compiler then takes care of translating the code into a form that the computer can understand.

In contrast to compilers, interpreters are programs that execute code directly, without first translating it into machine code. Interpreters are typically used for scripting languages such as Python and JavaScript.

Common examples of compilers include GCC (GNU Compiler Collection), Clang, and Microsoft Visual C++ Compiler.

Tutorial-2:

3. Understanding a simple C++ code

```
#include <iostream>
```

// This is called preprocessor directive. We need this file to run the programme!!

```
using namespace std;
```

// This includes standard libraries!

```
// main() is where program execution begins. These below lines are entirely called as function.

int main()
{
```

// Every computer programme starts with a function main. main always works with integers.

```
   cout << "Hello World. I am Sayak" << endl; // prints Hello World
```

*//cout is called ‘**output stream object** ’or ‘**alpha object’**. It is used to write characters on the computer screen.*
*// ‘<<’ is called ‘**stream insertion operator’**. It takes everything right to that and prints everything in the screen.*

```
// 'endl' simply tells end this line and go to the next line.

    return 0;
// Called as return statement. main function should always have return 0. When
it retuned 0, it ran fine without any errors.
}
```

```
Output=> Hello World. I am Sayak
```

Tutorial 3:

4. More about printing text

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

int main()
{
    cout << "Boy, I love bacon ";
    cout<<"and ham";
    return 0;
}
```

```
⇨ Boy, I love bacon and ham
```

❖ When the 1st line is written, you are not telling to go to the next line. So, both are being printed in the same line. So, you should add 'endl' this:

```
cout << "Boy, I love bacon "<<endl;
cout<<"and ham";
```

or, you can write like this:

```
 cout << "Boy, I love bacon \n";
cout<<"and ham";
```

Output:

```
⇨ Boy, I love bacon
  and ham
```

❖ If you want a blank line between these two lines:

```
cout << "Boy, I love bacon \n\n";
cout<<"and ham";
```

Output:

```
Boy, I love bacon

and ham
```

❖ If you write like this:

```
 cout << "Boy, \n I \n love \n bacon \n";
cout<<"and ham";
```

Output:

```
Boy,
 I
 love
 bacon
and ham
```

Tutorial 4:

5. Variables

❖ variables are just basically the placeholders.

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

int main()
{
 int tuna=6;
 cout << tuna;

   return 0;
}
Output => 6
```

❖ Let’s do some arithmetic operations.

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

int main()
{
 int a=6;

 int b=15;
 int sum=a+b;
 int difference=a-b;
 cout << sum <<endl;
 cout << difference;
   return 0;
}
```

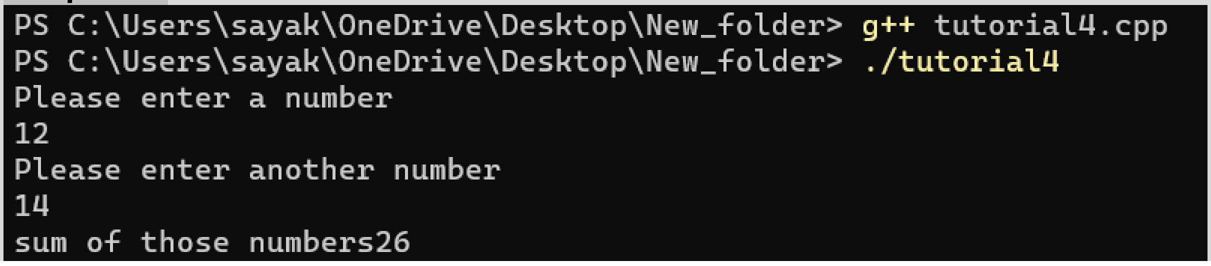
```
Output => 21
          -9
```

Tutorial-5:

5.1 Creating a basic calculator using variables

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

int main()
{
 int a; //These are called as Declaring a variable
 int b;
 int sum;

cout<< "Please enter a number \n";
//'cout' takes information from the computer and give it to user.
cin>>a;
//'cin' takes information from the user and give it to computer. 'cin' is called as input stream object; '>>' is called as stream extraction operator.
cout<< "Please enter another number \n";
cin>>b;

sum =a+b;
cout << "sum of those numbers"<< sum<<endl;
   return 0;

}
```

Output=>



```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ tutorial4.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> ./tutorial4
Please enter a number
12
Please enter another number
14
sum of those numbers26
```

Tutorial-6:

5.2 Overwriting the value of a variable.

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

int main()
{
 int tuna=54;
```

//In the previous line, you have specified 'int' type of data. So, in the 2nd line you need not to specify 'int' before variable 'tuna'

.

```
 tuna =76;
```

//Computer will wipe out the old value and create a room for the new one. So, latest assigned value will be in variable's memory.

```
 cout << tuna;

   return 0;

}
```

```
Output=> 76
```

Tutorial-7:

6. Basic Arithmetic

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

int main()
{
 int x=8-4;
 cout<< x <<endl;
 int y=8*4;
 cout<<y <<endl;
 int a=18/4;
//This result is not correct. Decimal value is eliminated for 'int' datatype.
 cout<<a <<endl;
 int b=28 % 5;
//Modular operator
 cout<<b <<endl;

   return 0;

}
```

```
Output =>
4
32
4
3
```

Tutorial-8

7. if statement

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

int main()
```

```
{
 int x=8;
 if (x>=3)
 {
    cout<<"Sayak is awesome!! \n";
 }
 if (x==8)
//Single '=' sign is used for assign the value of a variable. '==' is used for conditional operator.
 {
    cout<<"Sayak is good!! \n";
 }
 if (x!=3)
 {
    cout<<"Sayak is a good boy!! \n";
 }

   return 0;
}
```

```
Output=>
Sayak is awesome!!
Sayak is good!!
Sayak is a good boy!!
```

Tutorial-9

8. Functions

main is itself a function in every C++ programme. let's create another function outside the main function. First you should write the function and then call it in the main function. Otherwise, you may use *Function Prototyping*.

8.1 Void functions

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

void printSomething () {

    cout << "some text on the screen"<< endl;
}

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

```
Output=> some text on the screen
```

In C++, the ‘void’ keyword is used in a function declaration to specify that the function doesn't return a value. When you see void in a function declaration, it may do something, like printing a message to the console or performing some actions, but it doesn't return a value that you can store in a variable or use in an expression.

8.2 Function Prototyping

If you write any function after the ‘main’ function block, you should specify the function name before ‘main’ function block; so that it does not throw any error.

```
#include <iostream>
using namespace std;
void printSomething ();
// Now, computer understands, there is a function named ‘printSomething’
somewhere in the programme.

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

void printSomething () {
    cout << "some text on the screen"<< endl;
}
Output=> some text on the screen
```

Tutorial-10

8.3 Creating functions that uses parameter / argument.

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

void printcrap(int x){
    cout<<"Sayak's favorite number is "<< x <<endl;
}

int main()
{
   printcrap(20);
    return 0;
}
```

```
Output=> Sayak's favorite number is 20
```

It defines a function named **printcrap** that takes an integer parameter **x** and prints a message to the console with the value of **x**.

In the **main** function, it calls **printcrap** with the argument **20**. The code effectively demonstrates how to define and call a function in C++ with an argument.

When you are making a parameter you have to define ‘type of data’ and a ‘variable name’(i.e., ‘int x’ for this case).

Tutorial-11

8.4 Creating functions that uses multiple parameters / arguments.

Will see using multiple parameters in a single function. Right now, we can not use void; because will do some calculations!!

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

int addNumbers(int x, int y, int a, int b){
    int answer=x+y+a+b;
    return answer;
//after calculating answer, we need to return it. That's why we can't use void in this case.
}

int main()
{
    cout << addNumbers(45,35,20,10);
    return 0;
}
```

```
Output=> 110
```

Tutorial-12

9. Classes and Objects

Grouping similar functions in a class is useful.

The object is how you *access the stuffs which are inside the class*. Question is, why we would not use simple function call instead of object. Reason is: in large computer programmes, there may have functions with same name under different function classes (like 'hydraulic_radius' function under multiple channel's classes).

```
#include <iostream>
using namespace std;
class SayakClass{
    public:
```

//Public access specifier. It means anything you use after that ':', you may use those outside of the class. So, main() can also use that.
```
        void coolsaying(){
            cout << "preachin to the choir" <<endl;
        }
};

int main()
{
    SayakClass sayakObject;
```
//Creating an object for a specific class (almost similar to creating a variable). Later, we will call any function of that class using this object.
```

    sayakObject.coolsaying();
```
//calling the function using object.
```
    return 0;
}
```

```
Output=> preachin to the choir.
```

Tutorial-13

10. Using variables in classes.

10.1 Wrong way (As per YouTuber)

In this method, variables are made public. This is unwanted.

```
#include <iostream>
#include <string>
```
// You include the necessary header files for input and output (`<iostream>`) and for using strings (`<string>`).
```
using namespace std;

class SayakClass{
    public:
    string name;
};
```
// You define a class called `SayakClass` with a public member variable `name`, which is of type `string`.

```
int main()
```

```
{
    SayakClass so;
    so.name="Sayak Karmakar";
    cout<< so.name;
// In the main function, you create an object so of the SayakClass class. Then,
assign a value to the name member variable of the so object, and then you print
the value to the console.

    return 0;
}
```

```
Output=> Sayak Karmakar
```

10.2 Correct Way.

Now, if you make your variables from ‘public:’ to ‘private:’, we can’t use them in the main(). But we want to make variables private, because we do not want that some outsiders play with the programmes. so, follow this way:

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

class SayakClass{
    public:
        void setName(string x){
        name=x;
        }
// A public member function setName(string x). This is called as setter
method or setter function. It allows you to set the value of the name member
variable.
    string getName(){
        return name;
    }
// A public member function getName(), which is a getter method. It allows
you to retrieve the value of the name member variable.

    private:
```

```
    string name;
// A private member variable name, which cannot be directly accessed from outside the class.
};

int main()
{
    SayakClass so;
// you created an object so of the SayakClass class.
    so.setName("Mr. Sayak Karmakar");
    cout<< so.getName();
    return 0;
}
Output=> Mr. Sayak Karmakar
```

This approach is used to control access to the **name** variable. By making it private, you ensure that the variable can only be modified or accessed through the setter and getter methods, providing a level of data encapsulation and abstraction.

Tutorial-14

11. Constructors

Constructor is a function that gets called automatically as soon as you create an object. Creating constructor is almost similar as creating a function. Constructor does not have a return type, so you need not write something like int, void etc. Constructor name is exactly same as the class name.

11.1 Example 1

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

class SayakClass{
    public:
        SayakClass(){   //Constructor.
            cout<<"this will get printed automatically";

        }
        void setName(string x){
        name=x;
        }
    string getName(){
        return name;
    }

    private:
    string name;
};

int main()
{
    SayakClass so;
    return 0;
}
```

```
Output=> this will get printed automatically
```

The reason why people make constructors to give variables an initial value.

11.2. Example 2

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

class SayakClass{
    public:
        SayakClass(string z){
            setName(z);
        }
```

//The class has a public constructor. A constructor is a special member function in a class that is called when an object of the class is created. In this case, the constructor takes a string parameter z. The constructor immediately calls the setName method, passing the value of z. This way, when an object of SayakClass is created with a string argument (later, "Lucky Bucky Roberts"!), it automatically sets the name member variable.

```
        void setName(string x){
        name=x;
        }
```

```
    string getName(){
        return name;
    }

    private:
    string name;
};

int main()
{
    SayakClass so("Lucky Bucky Roberts");
    cout<<so.getName();
//You create an object of the SayakClass class named so and provide the string
"Lucky Bucky Roberts" as an argument to the constructor. When you create the
object, the constructor is called, and it sets the name member variable to "Lucky
Bucky Roberts."
//You then call the getName method on the so object, which retrieves the value
of the name member variable and prints it to the console using cout.

    SayakClass so2("Sayak Soham Ishani");
    cout<<so2.getName();
//Though two objects so and so2 are from same class it is not changing the same
variable (i.e., 'name' here). So, you can create multiple objects from a same
class and they don't overwrite each other. They are each assigned a set of
variables.
    return 0;
}
```

```
Output=> Lucky Bucky RobertsSayak Soham Ishani
```

Tutorial-15

12. Placing classes in a separate file

How to separate the classes and add them in different files. It helps to edit and manage later on.

files with .h extension is called as header file and with .cpp extension is called as C++ source file.

Header file is where we are going to put all of our classes, functions, prototypes, and variable decorations.

//Create header file (tutorial15.h)

```
#ifndef TUTORIAL15_H
#define TUTORIAL15_H

class tutorial15
{
    public:
        tutorial15();

};
#endif
```

- This is a header file (tutorial15.h) that starts with include guards (#**ifndef**, #**define**, and #**endif**). These guards ensure that the contents of the header file are included only once in a source file.
- It declares a class called **tutorial15**. Inside this class, there's a single public constructor (**tutorial15()).**

//Create C++ source file (tutorial15.cpp)

```
#include "tutorial15.h"
#include <iostream>

using namespace std;
tutorial15::tutorial15()
//:: is called as binary scope resolution operator. Here, 'tutorial15()' is a function or constructor which is a member of the class 'tutorial15'. If you do not put :: then it will not understand in which class tutorial15() belongs to.
{
    cout << "I eat mango"<< endl;

}
```

- This is the source file (**tutorial15.cpp**) that implements the functionality of the **tutorial15** class.

- It includes the **tutorial15.h** header file, allowing it to access the class declaration.
- The constructor **tutorial15::tutorial15()** is defined. It prints the message "I eat mango" to the console when an object of the **tutorial15** class is created. The **::** operator is used to indicate that this function is a member of the **tutorial15** class.

//Main programme (main.cpp)

```
#include <iostream>
#include "tutorial15.h"
//In order to use objects from different classes in our main.cpp, we need to
include headers.

using namespace std;
int main() {
    tutorial15 so;
//These are the class name and object name.
    return 0;
}
```

- This is the main program (**main.cpp**) where the program execution starts.
- It includes the necessary header files: **<iostream>** for input and output operations and **"tutorial15.h"** to access the **tutorial15** class.
- Inside the **main** function, an object of the **tutorial15** class is created with the name **so**. This object's constructor is automatically called, displaying "I eat mango" on the console.
- The program then returns **0** to indicate successful execution.

In summary, this code demonstrates the basic structure of a C++ program with a header file defining a class, a source file implementing the class, and a main program that uses the class by creating an object of it. When you run the program, it will print "I eat mango" to the console.

Output=>(See 'heading2 – 33' for more details)

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe main.cpp tutorial15.cpp

PS C:\Users\sayak\OneDrive\Desktop\New_folder> ./output.exe
```

```
I eat mango
```

Tutorial 16:

13. If statement

Now we want to have the computer make some simple choice.

Basic decision that each computer can make is made using the if statement. The code inside the 'if' block will run only when your test is true.

```
#include <iostream>

using namespace std;
int main() {
    int x=10;
    int y=23;
    if(x==10){    //Single'=' is for setting a value of variable.
        cout<<"IIT Kharagpur"<<endl;
    }
    if(x!=y){    //not equal
        cout<<"IIT Kanpur"<<endl;
    }
    if(x<=y){
        cout<<"IIT Madras"<<endl;
    }
    return 0;
}
```

```
Output=>    IIT Kharagpur
            IIT Kanpur
            IIT Madras
```

Tutorial 17:

14. If-else statement

'else' is used when you want to show some output when your test is false. See below:

```
#include <iostream>

using namespace std;
int main() {
    int age=65;
    if(age>60){
        cout << "wow! you are old."<<endl;
        cout << "Take rest."<<endl;
    }
    else{
        cout<<"you are young! get a job!"<<endl;
    }
    return 0;
}
```

```
Output=> wow! you are old.
         Take rest.
```

14.1 Nested 'if' statement.

```
#include <iostream>

using namespace std;
int main() {
    int age=165;
    if(age>60){
        if(age>100){
            cout<<"Wow! You made a record!"<<endl;
        }else{
            cout << "you are old."<<endl;
        }
    }
    else{
        cout<<"you are young! get a job!"<<endl;
    }
    return 0;
}
```

```
Output=> Wow! You made a record!
```

Note that, you may make very complex nested ‘if’ loop for as many times as you want.

15. While loop

15.1 Make an infinite while loop to finite.

```
#include <iostream>

using namespace std;
int main() {
    int bacon=0;
    while(bacon<=5){
        cout<<"bacon is"<<bacon<<endl;
    }
    return 0;
}
```

Inside the loop, it attempts to print the value of **bacon** along with the message "bacon is" to the console using **cout**. The problem with this code is that there's no code to update the value of **bacon** within the loop. As a result, if **bacon** starts at 0 and the loop runs, it will keep printing "bacon is 0" endlessly, and the loop will never terminate. This is an infinite loop. Let’s update the code:

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

int main() {
    int bacon = 0;

    while (bacon <= 5) {
        cout << "bacon is " << bacon << endl;
        bacon++;
// Increment 'bacon' by 1 in each iteration. you can write ‘bacon=bacon+1’ in place of ‘bacon++’.
    }

    return 0;
}
```

```
Output=> bacon is 0
         bacon is 1
         bacon is 2
         bacon is 3
         bacon is 4
         bacon is 5
```

To fix the code, you need to update the value of **bacon** within the loop to ensure that the condition **bacon <= 5** eventually becomes false, allowing the loop to exit. You can do this by incrementing **bacon** within the loop.

Tutorial 19:

15.2 Take 5 numbers entered by the user and get the sum using While loop.

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

int main() {
    int x = 1;
// Initializing a counter variable 'x' to 1
    int number;
// Declaring a variable 'number' to store input
    int total = 0;
// Initializing 'total' to 0 to store the sum of numbers

    while (x <= 5) {
// Enters a loop that runs five times
```

```
        cin >> number;
// Takes an integer input from the user
        total = total + number;
// Adds the input to the total
        x++;
// Increments the counter variable
    }

    cout << "sum = " << total << endl;
// Outputs the total sum
    return 0;
}
```

```
Output=>
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o tutorial19.exe tutorial19.cpp

PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\tutorial19.exe
```

```
13
45
23
67
2
sum=150
```

Tutorial 20:

16. Sentinel Controlled program

In the context of input processing, a sentinel-controlled program typically involves a loop that continues to process input until a specific sentinel (e.g., -1) value is encountered, at which point the loop terminates. The sentinel value is used as a signal or flag to indicate the termination of a loop or process.

- Write a program to calculate how many people we entered and calculate the average age of the people.

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

int main() {
    int age;
    int ageTotal=0;
    int noOfPeople=0;

    cout<< "Enter 1st person's age or -1 to quit"<<endl;
    cin>>age;
//user will enter a number and will be stored in the variable 'age'.

    while(age != -1){
        ageTotal=ageTotal+age;
        noOfPeople++;

        cout<< "Enter next person's age or -1 to
quit"<<endl;
        cin>>age;
    }
    if (noOfPeople > 0){
// Check to avoid division by zero
        cout << "Number of people = " << noOfPeople <<
endl;
        cout << "Average age: " << ageTotal / noOfPeople
<< endl;
    } else {
        cout << "No age data provided." << endl;
    }
    return 0;
}
```

Output (When, `noOfPeople > 0`)=>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o
test.exe test.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\test.exe
Enter 1st person's age or -1 to quit

14
```

Enter next person's age or -1 to quit

```
45
```

Enter next person's age or -1 to quit

```
23
```

Enter next person's age or -1 to quit

```
-1
Number of people = 3
Average age: 27
```

Output (When, `noOfPeople = 0) =>`

PS C:\Users\sayak\OneDrive\Desktop\New_folder> **`g++ -o test.exe test.cpp`**

PS C:\Users\sayak\OneDrive\Desktop\New_folder> **`.\test.exe`**

Enter 1st person's age or -1 to quit

```
-1

No age data provided.
```

Tutorial 21:

17. Assignment and increment operators

17.1 C++ works without ‘return 0’!

In new C++ programs, we don’t need to write “return 0” after the last statement. It automatically assumes that. Look, the below code works!

```
#include <iostream>

using namespace std;
int main() {
    cout<<"Hello, world!"<<endl;
}
```

```
Output=> Hello, world!
```

17.2. Example with increment operator.

```
#include <iostream>

using namespace std;
int main() {
    int x =10;
    int y=20;
    int z=6;
    int a=9;
    int b=20;
    x+=10; //means, x=x+10
    y-=5; //means, y=y-5
    z*=4; //means, z=z*4
    a/=3; //means, a=a/3
    b%=3; //Gives the remainder when b is divided by 3.
    cout<<x<<endl;
    cout<<y<<endl;
    cout<<z<<endl;
    cout<<a<<endl;
    cout<<b<<endl;
}
```

```
Output =>
20
15
24
3
2
```

17.3 Difference between x++ and ++x

These are called These are called **post-increment** and **pre-increment** operators respectively.

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

int main() {
    int x = 20;
    cout << x++ << endl;
// Outputs the current value of x (20) and then increments x
    cout << x << endl;
// Outputs the updated value of x (21)
}
```

```
Output=> 20
          21
```

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

int main() {
    int x = 20;
    cout << ++x << endl;
// Increments x first and then outputs its value (x becomes 21)
    cout << x << endl;
// Outputs the updated value of x (21)
}
```

```
Output=> 21
          21
```

//Similarly, these things work for x - - and - - x also.

Tutorial 22:

18. For Loop

A Basic example:

```
#include <iostream>

using namespace std;
int main(){
    for (int x=1; x<10; x+=1){
        cout<<x<<endl;
    }
//for loop requires 3 information in (). Starting value (initialization of a variable), ending value (loop continuation condition) and increment value. Here, the loop runs for x=1 to 9 with increment=1. you may use 'x++' in place of 'x+=1'.
}
```

```
Output=>
1
2
3
4
5
6
7
8
9
```

Tutorial 23:

18.1 Making a stock market simulator using 'for' loop (Compound Interest Problem)

```
#include <iostream>
#include <cmath> //needed to include power

using namespace std;
// This line is used to avoid specifying std:: before standard functions, allowing you to directly use cin, cout, endl etc.
int main() {
    float a; //We want to work with the decimal numbers
    float p=10000;
    float r=0.01;

    for(int day=1; day<=20; day++){
        a=p * pow(1+r, day);
        cout<<day<<"-----"<<a<<endl;
    }
}
```

```
Output=>
1-----10100
2-----10201
3-----10303
4-----10406
5-----10510.1
6-----10615.2
7-----10721.4
8-----10828.6
9-----10936.9
10-----11046.2
11-----11156.7
12-----11268.2
13-----11380.9
14-----11494.7
15-----11609.7
16-----11725.8
17-----11843
18-----11961.5
19-----12081.1
20-----12201.9
```

18.2 What is the use of ‘#include <cmath>’?

In C++, the **<cmath>** header (in C it's **<math.h>**) provides a set of functions to perform mathematical operations and calculations. It's primarily used for mathematical computations involving numbers like trigonometric, logarithmic, exponential, and power functions. Here are some common functionalities provided by the **cmath** library:

1. **Basic Mathematical Functions:**

- **Trigonometric functions:** **sin()**, **cos()**, **tan()**, **asin()**, **acos()**, **atan()**, etc.
- **Hyperbolic functions:** **sinh()**, **cosh()**, **tanh()**, etc.
- **Exponential and logarithmic functions:** **exp()**, **log()**, **log10()**, **log2()**, **exp2()**, etc.

2. **Power Functions:**
   - **pow()**: Computes a number raised to a power.
   - **sqrt()**: Calculates the square root of a number.
   - **cbrt()**: Calculates the cube root of a number.
3. **Rounding and Absolute Functions:**
   - **round()**, **floor()**, **ceil()**: Rounding functions.
   - **abs()**: Returns the absolute value of a number.
4. **Constants:**
   - Constants such as **M_PI** (pi), **M_E** (Euler's number), etc.

Tutorial 24:

19. do while loop

do while loop is basically while loop flipped upside down. It runs a code first and then runs the test.

```
#include <iostream>
#include <cmath>

using namespace std;
int main() {
    int x=1;
    do{
        cout<<x<<endl;
        x++;
    }while(x<5);
//In basic while loop, there is no semicolon(;).But, here
it is.
}
```

```
Output=>
1
2
```

```
3
4
```

Why people will use do while loop instead of while loop?? Because, while loop will not run if the test condition is false. But, in a do-while loop, the loop body is executed at least once before the condition is checked. Example:

```
#include <iostream>
#include <cmath>

using namespace std;
int main() {
    int x=10;
    do{
        cout<<x<<endl;
        x++;
    }while(x<5);
}
```

```
Output=> 10
```

Tutorial 25:

20. Switch statement

We know, how to write a program using if statement:

```
#include <iostream>
#include <cmath>

using namespace std;
int main() {
    int age=21;
    if(age==21){
        cout<<"you can drive car"<<endl;
    }
    else if (age==18){
        cout<<"you can cast your vote"<<endl;
    }
```

//One point should be noted. To correctly handle multiple conditions, you should use **else if** *(not 'if' again) for subsequent conditions after the initial* **if***. If you again use if, the code executes the first* **if** *block and then, regardless of the*

outcome of the first condition, it executes the second `if` *block independently and then the* `else` *block. Otherwise, the output will be:*

```
you can drive car
sorry, you get nothing
```

//It means default message will be always printed.

```
    else{
        cout<<"sorry, you get nothing"<<endl;
    }
}
```

```
Output=> you can drive car
```

Let's write the above program using switch statement.

```
#include <iostream>
#include <cmath>

using namespace std;
int main() {
    int age=21;
    switch(age){
        case 21:
        cout<<"you can drive car"<<endl;
        break;
        case 18:
        cout<<"you can cast your vote"<<endl;
        break;
        default:
        cout<<"sorry, you get nothing"<<endl;
    }
}
```

```
Output=> you can drive car
```

Tutorial 26:

21. Logical Operators

First, let’s write a programme to show whether a person is allowed to enter a club (using nested ‘if’ statement):

```
#include <iostream>
#include <cmath>

using namespace std;
int main() {
    int age=23;
    int money=10000;
    if (age>21){
        if (money>500){
            cout<<"you are allowed in"<<endl;
        }
    }
}
Output=> you are allowed in
```

Let’s write the programme using logical operator so the we can convert all the tests in a single line.

```
#include <iostream>
#include <cmath>

using namespace std;
int main() {
    int age=23;
    int money=10000;
    if (age>21 && money>500){
//You may include multiple tests like (test && test && test......)
        cout<<"you are allowed in"<<endl;
    }
}
Output=> you are allowed in
```

21.1. Example

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

```
int main() {
    int x = 5;
    int y = 10;
    bool condition1 = (x < 3);
    bool condition2 = (y < 20);
//These 'condition1' and 'condition2' are called Boolean variables. See 21.2 for more.

    // Logical AND
    if (condition1 && condition2) {
        cout << "Logical AND: Both conditions are true."
<< endl;
    }

    // Logical OR
    if (condition1 || condition2) {
        cout << "Logical OR: At least one condition is
true." << endl;
    }

    // Logical NOT
    if (!condition1) {
        cout << "Logical NOT: Condition1 is false." <<
endl;
    }

    return 0;
}
```

```
Output=> Logical OR: At least one condition is true.
         Logical NOT: Condition1 is false.
```

21.2 Boolean Variables.

```
#include <iostream>
using namespace std;
int main() {
    int x = 5;
    int y = 10;
    bool condition1 = (x < 3);
```

//This line checks whether the value of the variable `x` is less than `3` and stores the result (either `true` or `false`) in the Boolean variable `condition1`. If true returns 1; if false returns 0. Here, it is false, so returns 0.

```
    bool condition2 = (y < 20);
    cout<<condition1<<endl;
    cout<<condition2<<endl;
    cout<<(condition1 && condition2)<<endl;
    cout<<(condition1 || condition2)<<endl;
```

//Without bracket (), above two lines will not work.

```
    cout<<!condition1<<endl;
}
```

```
Output=>
0
1
0
1
1
```

Tutorial 27:

22. Random Number Generator

```
#include <iostream>
#include <cstdlib>
```

// stands for 'c standard library'. Required to run function 'rand'.

```
#include <ctime>
```

//Required to generate different random numbers with time changes.

```
using namespace std;

int main() {
    cout<<"Generate a random number"<<endl;
    cout<<rand()<<endl; //Generates a random number.


    cout<<"Generate 4 random numbers"<<endl;
    for (int x=1; x<5; x++){
        cout<< rand() << endl;
    }


    cout<<"Generate 7 random results of rolling
dice"<<endl;
    for (int x=1; x<=7; x++){
        cout<< 1+(rand()%6) << endl;
    }
//note that, every time you run this code, it will generate same values! So, it is
not truely random, computer follows a certain (complex!) algorithm.


    cout<<"making different set of random number using
srand(seed)"<<endl;
    srand(252);
//If you change this number within bracket(), you will get different set of random
numbers each time.
    for (int x=1; x<=5; x++){
        cout<< 1+(rand()%6) << endl;
    }


    cout<<"generate different random numbers with time
changes."<<endl;
    cout<<"perfect random number generator"<<endl;
    srand(time(0));
// Seeds the random number generator using the current time. This is useful for
generating different random numbers each time the program runs.
    for (int x=1; x<=5; x++){
        cout<< 1+(rand()%6) << endl;
    }
}
```

```
Output=>
Generate a random number
41
Generate 4 random numbers
18467
6334
26500
19169
Generate 7 random results of rolling dice
5
1
1
5
3
6
```

```
6
making different set of random number using srand(seed)
4
2
6
5
5
generate different random numbers with time changes.
perfect random number generator
6
6
1
3
5
```

Tutorial 28

23. Default Arguments/Parameters

Generally, we call a function with all of its arguments. See the below example:

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

//Function Declaration
int volume(int l, int w, int h);
//This is called function prototype.

int main(){
    cout<<volume(4,5,3);
}

//Function Definition
int volume(int l, int w, int h){
    return l*w*h;
}
```

```
Output => 60
```

Let’s now use default arguments and see what happens.

```
#include <iostream>
```

```
using namespace std;

//Function Declaration
int volume(int l=3, int w=2, int h=1);
//You have to specify default values in function prototype only, not in function definition. If we do not explicitly specify any values while calling a function, default values would be used there.

int main(){
    cout<<"with no argument: "<<volume()<<endl;
    cout<<"with one argument: "<<volume(5,6)<<endl;
//function uses these values for 1st and 2nd arguments respectively.
    cout<<"with two arguments: "<<volume(5)<<endl;
//function uses this value for 1st argument
    cout<<"explicitly specifying all the arguments: 
"<<volume(10,5,4)<<endl;
//all three arguments are changed.
}

//Function Definition
int volume(int l, int w, int h){
    return l*w*h;
}
```

```
Output=>
with no argument: 6
with one argument: 30
with two arguments: 10
explicitly specifying all the arguments: 200
```

Tutorial 29

24. Unary Scope Resolution Operator

If any function has a local variable with the same name as global variable, it outputs the local value of that variable. For example, in this below code, main() function displays local tuna.

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

```
void bucky();
```
//This line is a function prototype or forward declaration of the function bucky(). It tells the compiler that such a function will be defined later in the code.

```
int tuna = 20;
```
// Global variable. Every function can use it.

```
int main() {
    int tuna = 69;
```
*//variable defined in local **scope** of the main function; called local variable. This one can be used by the main() function only.*
```
    cout << "Output of main(): "<< tuna << endl;
```
// Outputs the value of the local variable tuna (not global one) in the main function. because, function uses the most recent value.
```
    bucky();
```
*// **Calls the function bucky**. To execute a function in a program, that function need to be explicitly called from main().*
```
}

void bucky() {
   cout << "Output of bucky(): "<<  tuna << endl;
```
// Outputs the value of the global variable tuna.
```
}
```

```
Output:
Output of main(): 69
Output of bucky(): 20
```

Now, using **Unary Scope Resolution Operator (::)** you can use the global tuna in your main() function. See below:

```
 #include <iostream>

using namespace std;
void bucky();

int tuna = 20;

int main() {
    int tuna = 69;
    cout << "local: "<< tuna << endl;
    cout<<"Global: "<< ::tuna << endl;
}
```

```
Output:
local: 69
Global: 20
```

25. What is the difference between int and void functions?

The difference between `int` and `void` functions in C++ pertains to their return types and how they handle and return values.

1. **`int` functions:**

An `int` function is expected to return an integer value after performing its operations.

When defining an `int` function, you specify the type of value it will return.

2. **`void` functions:**

A `void` function, on the other hand, doesn't return any value. It's used when a function performs an operation but doesn't need to return a value.

See the below example:

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

int add(int a, int b) {
    return a + b; // Returns the sum of a and b
}

void greet() {
    cout << "Hello!"; // Outputs a greeting message
}
```

```
int main(){
    cout<<"sum= "<<add(4,5)<<endl;
    //cout<<greet<<endl; --->this is the wrong way.
    greet();
//The function greet() is a void function, so it doesn't return any value explicitly.
To execute it and produce its intended behavior (in this case, printing "Hello!"),
you simply call it by using greet();.
}
```

```
Output:
sum= 9
Hello!
```

Tutorial 30

26. Function Overloading

Function overloading, where multiple functions with the same name but different parameter types are defined within the same scope, allowing the appropriate version to be executed based on the type of argument passed.

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

void printNumber(int x){
    cout<<"I am printing an integer--> "<< x <<endl;
}
void printNumber(float x){
    cout<<"Now I am printing a float--> "<< x <<endl;
}
int main(){
    int a=54;
    float b=32.458;
    printNumber(a);
    printNumber(b);
}
```

```
Output=> I am printing an integer--> 54
    Now I am printing a float--> 32.458
```

In this case, the `printNumber` function is overloaded to accept both integer and float data types. Depending on the type of argument passed to the function, it will execute different versions of the function accordingly.

Tutorial 31

27. Recursion

A function can call itself. See below:

27.1 Bad way to use recursive function.

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

void sayak(){
    cout<<" IITKGP ";
    sayak();
}

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

```
Output=>
IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP
IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP
IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP
IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP IITKGP   IITKGP
IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP  IITKGP…………………………….
```

The `sayak()` function is called within the `main()` function. Inside the `sayak()` function, it prints " IITKGP " to the console. Then, it calls itself (`sayak()`) again.

As there is no **base case or condition** that would stop this recursive process, the `sayak()` function will continue to call itself endlessly. This leads to an infinite loop of function calls, causing what is known as a **stack overflow**.

27.2. Good way to use recursive function: Finding the factorial of a number.

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

int factorialFinder(int x){
    if(x==1){
        return 1;
    }else{
        return x*factorialFinder(x-1);
    }
}

int main(){
    cout<<"5! = "<<factorialFinder(5)<<endl;
    return 0;
}
```

```
Output=> 5! = 120
```

For **`factorialFinder(5)`**, the function **`factorialFinder()`** is called for each value from 5 down to 1, performing the necessary multiplications until it reaches the base case (when x is equal to 1) and it starts to combine the results back up through the chain of function calls.

Tutorial 32

28. Arrays

Array is almost like a variable, but it can store multiple values. See the simplest way to create an array:

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

int main(){
    int bucky[5]={22,45,7,23,98};
//Arry initializer list??!
    cout<<"First element= "<<bucky[0]<<endl;
//Array index. Counting starts from 0.
}
```

```
Output=> First element= 22
```

Tutorial 33

28.1 Creating an array using loops

Let’s create an array using ‘for’ loop and use the array to generate a multiplication table of 11.

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

int main(){
    int bucky[10];
// directs compiler to hold enough space to accommodate 10 integers.
    cout<<"Multiplication Table of 11"<<endl;

    for(int x=0;x<=9;x++){
//"x<=9" and "x<10" means same for int type.
    bucky[x]=(x+1)*11;
    cout<< "11*"<< x+1 <<"= "<< bucky[x] <<endl;
    }
    //bucky[]; ---> This will not show the array. See below.
    cout<<"bucky[]= ";
//not changing the line. we want next results in the same line.
    for (int i = 0; i < 10; i++) {
    cout << bucky[i] << " ";
// "  " is there to provide some space between the results.
    }
}
```

```
Output =>
Multiplication Table of 11
11*1= 11
11*2= 22
11*3= 33
11*4= 44
11*5= 55
```

```
11*6= 66
11*7= 77
11*8= 88
11*9= 99
11*10= 110
bucky[]= 11 22 33 44 55 66 77 88 99 110
```

Tutorial 34

28.2 Using Arrays in calculations

Let’s write a code to get sum of all array elements.

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

int main(){
    //Calculate 'sum' manually
    int tuna[5]={22,45,7,23,98};
    int oldSum=tuna[0]+tuna[1]+tuna[2]+tuna[3]+tuna[4];
    cout<<"sum by tedious way= "<<oldSum<<endl;

    //'sum' using for loop
    int sum=0;
    for(int x=0; x<5; x++){
        sum+= tuna[x];
        cout<<sum<<endl;
//This displays the cumulative sum at each iteration
    }
    cout<<"sum by better way = "<<sum<<endl;
//this shows final 'sum' value.
}
```

```
Output=>
sum by tedious way= 195
22
67
74
97
195
sum by better way = 195
```

Tutorial 35

28.3 Passing Arrays to functions

Let’s write a code to print the elements of an array using function.

```
#include <iostream>
using namespace std;
void printArray(int theArray[], int sizeOfArray);

int main(){
    int bucky[5]={22,45,7,23,98};
    int jessica[3]={2,45,52};
    printArray(bucky,5);
```

//Here, we are ***passing an array*** *named 'bucky' and a variable (value=5) as arguments to the function 'printArray'. This will print the elements of the bucky array in console.*

```
    //cout<<"the array= "<<printArray(bucky,5)<<endl;
```

---> won't give the correct output as it attempts to print the return value of the printArray function, which is non-existent for 'void' functions.

```
}

void printArray(int theArray[], int sizeOfArray){
```

//The square bracket is provided to identify theArray[] as an array, not a variable.

```
    for(int x=0; x < sizeOfArray; x++){
        cout << theArray[x]<<endl;
    }
}
```

//'printArray' function prints each element of 'theArray' array in each iteration.'sizeOfArray' argument is required to know for how many times the iteration will run.

```
Output=>
22
45
7
```

```
23
98
```

Tutorial 36

29. Multi-Dimensional Array

This is like array made of arrays. Combination of rows and columns. See example:

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

int main(){
    int sally[2][3]={{2,3,4},{8,9,10}};
//This has 2 rows and 3 columns.
    cout<<sally[0][0];
//This shows the element of 1st row and 1st column. Index starts from 0, not 1.
}
```

```
Output=> 2
```

Tutorial 37:

29.1. How to print a multi-dimensional array

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

int main(){
    int bertha[2][3]={{2,3,4},{7,8,9}};

    for(int row=0;row<=1;row++){
        for(int column=0;column<=2;column++){
            cout<<bertha[row][column]<<"   ";
//Here, we don't put endl. Because, inner 'for' loop will evaluate elements of a particular row. We just need a space between them.
        }
        cout<<endl;
//Here, endl is used, beacause now we will go to the next row.
    }
}
Output=>
2  3  4
7  8  9
```

Tutorial 38

30. Pointers

Pointers are variables that contains a memory address as their value.

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

int main(){
    int fish=5;
    cout<< &fish <<endl;
//'&' sign is called the address operator. This line will output the hexadecimal memory address of the variable fish in console.

    int *fishPointer;
//asterik(*) sign tells that we are making pointer, not regular variable.
     fishPointer= &fish;
//Assigns the memory address of the variable fish to the pointer variable fishPointer using &fish.
     cout<<fishPointer<<endl;
}
```

```
Output=>
0x61ff08
0x61ff08
```

Tutorial 39

30.1 Pass by references with pointers

Two different ways to pass arguments to the function. They are: passed by value and passed by references. Previously, we passed it by value; means we pass a copy of that variable into the function. The original variable can’t be changed until we set it to another value. So, function has access to the copy of that variable, not that original variable.

But, whenever you pass a variable to a function by reference, you pass the variable address. So, you give function the direct access to that variable. This gives less stress to the computer.

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

void passByValue(int x);
void passByReference(int *x);
//The function prototypes are declared. passByValue takes an integer argument by value, while passByReference accepts an integer argument by reference through a pointer.

int main(){
    int belly =13;
    int sandy=13;
    passByValue(belly);
//the passByValue function takes the copy of the belly. So, any changes made within the function (x = 99) do not affect the original belly variable in main. This is due to passing by value, which creates a local copy of the variable.
    passByReference(&sandy);
// this function takes memory address. So, we should put '&' before that. The passByReference function takes the memory address of sandy, enabling it to change the value of the original sandy variable in main to 66. By using a pointer and dereferencing it (*x = 66), the function modifies the value stored at the memory location where sandy is stored.
    cout<<"belly is now "<<belly<<endl;
    cout<<"sandy is now "<<sandy<<endl;
}

void passByValue(int x){
    x= 99;
}

void passByReference(int *x){
    *x=66;
}
```

```
Output=>
belly is now 13
sandy is now 66
```

Tutorial 40

31. Size of function

'sizeof()' determines the size of array, constants or variables in terms of bytes. See below:

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

int main(){
    char c;
    int i;
    double d;
    double bucky[10];

    cout<<sizeof(c)<<endl;
//Any character in computer occupies 1 byte in computer.

    cout<<sizeof(i)<<endl;
//this value may vary in different computers. here, it's 4 bytes.
    cout<<sizeof(d)<<endl;
//double takes more memory than integers because they are more precise. here,
it's 8 bytes.
    cout<<sizeof(bucky)<<endl;
//each doble takes 8 bytes. so. 8*10=80.
    cout<<sizeof(bucky)/sizeof(bucky[0])<<endl;
//This is a basic practical application of 'sizeof()' function. In this way, we can
know how many elements are there in an array.
}
```

```
Output=>
1
4
8
80
```

```
10
```

Tutorial 41

32. Pointers and Math

The code demonstrates the manipulation of pointers associated with the array `bucky`. It initializes several pointers, each pointing to distinct elements of the `bucky` array. These pointers are then manipulated to showcase how their values change when incremented or modified.

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

int main(){
    int bucky[5];
    int *bp0 = &bucky[0];
//Assigns the memory address of the first element of the array bucky to the
pointer bp0.
    int *bp1 = &bucky[1];
    int *bp2 = &bucky[2];
    int *bp3 = &bucky[3];
    int *bp4 = &bucky[4];

    cout<<"The memory address stored by the pointer bp0
is= "<<bp0<<endl;
    cout<<"The memory address stored by the pointer bp1
is= "<<bp1<<endl;
    cout<<"The memory address stored by the pointer bp2
is= "<<bp2<<endl;
    cout<<"The memory address stored by the pointer bp3
is= "<<bp3<<endl;
    cout<<"The memory address stored by the pointer bp4
is= "<<bp4<<endl;

    bp0+=2;
// Moves bp0 to the third element by incrementing the pointer by 2 (not by
changing the address mathematically).
    bp2++;
// Moves bp2 to the next element.
    cout<<"Now, bp0 stores= "<<bp0<<endl;
    cout<<"Now, bp2 stores= "<<bp2<<endl;
}
```

```
Output=>
The memory address stored by the pointer bp0 is= 0x61fee8
The memory address stored by the pointer bp1 is= 0x61feec
The memory address stored by the pointer bp2 is= 0x61fef0
The memory address stored by the pointer bp3 is= 0x61fef4
The memory address stored by the pointer bp4 is= 0x61fef8
Now, bp0 stores= 0x61fef0
Now, bp2 stores= 0x61fef4
```

This is an important concept where pointer arithmetic affects the elements a pointer refers to rather than the actual memory address.

Tutorial 42

33. Arrow Member selection Operator

We can access functions in a class using **object** or **pointer**. Whenever you will use a pointer to access functions and other stuffs inside a class you need to use **Arrow Member selection Operator** (->). See below code:

In Header file (TUTORIAL42.h), we will make our prototypes. in TUTORIAL42.cpp, we will define our member functions (called as source file). In main.cpp we will use Arrow Member selection Operator (->).

```
// TUTORIAL42.h (Header file)

#ifndef TUTORIAL42_H
#define TUTORIAL42_H

class TUTORIAL42
{
    public:
        TUTORIAL42(); //Constructor
        void printCrap();
```

```
//method or member function that doesn't return any value (void) and does not take any parameters.
	protected:
	private:

};
#endif


// TUTORIAL42.cpp (Source file)
#include "TUTORIAL42.h" //Includes the TUTORIAL42.h header file to use the TUTORIAL42 class in this file.
#include <iostream>
using namespace std;
TUTORIAL42::TUTORIAL42(){
} //TUTORIAL42::TUTORIAL42() { ... } is the constructor definition for the TUTORIAL42 class. Constructors are special member functions that are automatically called when an object of the class is created. Constructors are written without specifying any data type before them (such as: int, void etc) unlike functions. Here, it's an empty constructor.
void TUTORIAL42::printCrap(){
// Defines printcrap function belongs to class TUTORIAL42.
	cout<<"Did you say something?"<<endl;
}



// main.cpp

#include <iostream>
#include "TUTORIAL42.h"
using namespace std;

int main(){
	TUTORIAL42 TUTORIAL42Object;
//'TUTORIAL42' class makes the object named as 'TUTORIAL42Object'
	TUTORIAL42 *TUTORIAL42Pointer= &TUTORIAL42Object;
//This line of code establishes a pointer, TUTORIAL42Pointer, that points to the memory location where the TUTORIAL42Object is stored. 'TUTORIAL42' at the beginning specifies the class name (like int x, float y; where int, float are the classes of the variable).
```

```
    TUTORIAL42Object.printCrap();
//Using object 'TUTORIAL42Object', we can call 'printCrap' function.
   TUTORIAL42Pointer->printCrap();
// Using pointer 'TUTORIAL42Pointer', we can call 'printCrap' function.
Whenever you using a pointer to access stuff inside a class you need to use
Arrow Member selection Operator (->), not dot(.) like object.

}
```

Output (in windows PowerShell using MinGW compiler) =>

```
PS C:\Users\sayak> cd OneDrive\Desktop\New_folder

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o
output.exe main.cpp TUTORIAL42.cpp

PS C:\Users\sayak\OneDrive\Desktop\New_folder>
./output.exe

Did you say something?

Did you say something?
```

With pointer, you can use it to access and manipulate the `TUTORIAL42Object` indirectly by dereferencing the pointer (as shown in above code). Remember that using pointers requires careful handling to avoid issues such as null pointer dereferencing or memory access violations. Always ensure that the pointer is pointing to a valid object before using it.

Remember, before executing, you should save all three files i.e., **TUTORIAL42.cpp**, **TUTORIAL42.h** and **main.cpp.** In the output portion, "`g++ -o output.exe main.cpp TUTORIAL42.cpp`" using this command, `MinGW` compiler `g++` compile the source code files into an executable file named as `output.exe`. Then using "`./output.exe`" that executable file is executed to show the output.

Tutorial 43

34. Deconstructors / Destructors

Deconstructors are code that run automatically upon the destruction of your object. When you create your object, constructor runs and when you delete it deconstructor run.

Unlike constructor, Deconstructor has no parameter, don’t have a return value (even you can’t give it void). Also, there is no deconstructor overloading; you can’t have different versions of deconstructor.

//Header file (TUTORIAL43.h)

```
#ifndef TUTORIAL43_H
#define TUTORIAL43_H

class TUTORIAL43
{
    public:
        TUTORIAL43(); //Constructor
        ~TUTORIAL43();
//Deconstructor. It has same name as constructor, with a tilde (~) sign before it.
    protected:
    private:

};
#endif
```

//Source file (TUTORIAL43.cpp)

```
#include "TUTORIAL43.h"
#include <iostream>
using namespace std;
TUTORIAL43::TUTORIAL43(){
    cout<<"I am the constructor!"<<endl;
//TUTORIAL43() is the constructor in TUTORIAL43 class.
}
TUTORIAL43::~TUTORIAL43(){
    cout<<"Now object is destroyed; I am the deconstructor over here!"<<endl;
//~TUTORIAL43() is the deconstructor in TUTORIAL43 class.
}
```

//main.cpp

```
#include <iostream>
#include "TUTORIAL43.h"
using namespace std;

int main(){
    TUTORIAL43 to;
//'to' is the object name.
    cout<<"This will be printed after constructor is called."<<endl;
}
//After closing curly bracket, program ends and all objects are destroyed. Now Deconstructor will be called.
```

Output=>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe main.cpp TUTORIAL43.cpp

PS C:\Users\sayak\OneDrive\Desktop\New_folder> ./output.exe

I am the constructor!

This will be printed after constructor is called.

Now object is destroyed; I am the deconstructor over here!
```

Tutorial 44

35. const Objects

Keyword ‘const’ stands for constant. When keyword ‘const’ is placed before a variable it remains remains constant throughout your program, it can’t be changed or modified. This is the difference with regular variable. See below:

35.1 Regular variable

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

int main(){

    int x=3;
    x=5;
    cout<<"x= "<<x<<endl;
}
Output=> x= 5
```

35.2 Constant Variable:

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

int main(){
    const int y=3; //variable 'y' with 'const' keyword.
    y=5; //The value can't be changed. Throws error.
    cout<<"y= "<<y<<endl;
}
```

```
Output=>
tempCodeRunnerFile.cpp: In function 'int main()':
tempCodeRunnerFile.cpp: 12:7: error: assignment of read-only
variable 'y'
     y=5; //The value can't be changed. Throws error.
       ^

[Done] exited with code=1 in 1.769 seconds
```

35.3 How to call constant function using constant object

You cannot call a regular function or variable with a constant object. You need a constant function to use constant object. See below:

//Header file (Sally.h)

```
#ifndef SALLY_H
#define SALLY_H

class Sally
{
    public:
        Sally();
//Prototype of empty Constructor.
        void printShiz();
//prototype of regular function.
        void printShiz2() const;
//prototype of 'const' function.
    protected:
    private:

};
#endif
```

//Source file (Sally.cpp)

```
#include "Sally.h"
#include <iostream>
using namespace std;
Sally::Sally(){
//This is a void constructor! Not necessary. Still it’s there.
}
void Sally::printShiz(){
    cout<<"I am a regular function"<<endl;
}

void Sally::printShiz2() const{
```

```
//This is the way to build a constant function.
    cout<<"I am a constant function"<<endl;

}
```

//main.cpp

```
#include <iostream>
#include "Sally.h"
using namespace std;

int main(){
    Sally salObj;
//'salObj' is the regular object of class 'Sally'
    salObj.printShiz();

    const Sally constObj;
//'constObj' is the constant object of class 'Sally'
    //constObj.printShiz(); --> This will not work. Because, you
created an object that is constant; and you try to use a function that is not
constant. constant object can only use constant function.
    constObj.printShiz2();

}
```

Output =>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o
output.exe main.cpp Sally.cpp

PS C:\Users\sayak\OneDrive\Desktop\New_folder>
./output.exe

I am a regular function

I am a constant function
```

Tutorial 45

36. Member Initializers

Anything in a class is called as Member of that class, like variables, functions.

Constant variables are need to be initialized using member initializer list!! You need to make the list in between parameters and the body of the constructor.

//Header file (Sally.h)

```
#ifndef SALLY_H
#define SALLY_H

class Sally
{
    public:
        Sally(int a , int b);
//Declares a Constructor that would take two arguments to initialize 'regVar' and 'constVar'.
        void print();
//declares a member function to print the variables.
    protected:
    private:
        int regVar;
//Declares regular variable.
        const int constVar;
// Declares constant variable.

};
#endif
```

//Source file (Sally.cpp)

```
#include "Sally.h"
#include <iostream>
using namespace std;
Sally::Sally(int a, int b)
```

```
:regVar(a), constVar(b)
```
*// Initializes the member variables **regVar** and **constVar** with the values of a and b, respectively. This colon (:) is called member initializer syntax. Please separate the variables using comma (,). Do not put a semicolon (;) after this. It is just a list of variables and their values.*

```
{
}
void Sally::print(){
    cout<<"regular variable is: "<<regVar<<", Constant variable is: "<<constVar<<endl;
}
```

//main.cpp

```
#include <iostream>
#include "Sally.h"
using namespace std;

int main(){
    Sally so(3,87);
```
// Creates an object `so` of the `Sally` class with arguments `3` and `87` for the constructor. Two arguments are required for regular and constant variables.

```
    so.print();
```
//Calls the `print()` method/member function of the `so` object to display the values of the member variables.

```
}
```

```
Output=>
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe main.cpp Sally.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> ./output.exe
regular variable is: 3, Constant variable is: 87
```

36.1. What is the meaning of first two lines of a header file?
#ifndef SALLY_H
#define SALLY_H

In C++, the lines `#ifndef SALLY_H` and `#define SALLY_H` are known as include guards. They are used to prevent a header file from being included multiple times in the same translation unit (source file) during the compilation process.

Here's a breakdown:

1. **#ifndef SALLY_H:**
    - **#ifndef** stands for "if not defined." It checks if the identifier **SALLY_H** has not been previously defined.
    - If **SALLY_H** has not been defined, the code between **#ifndef** and a corresponding **#endif** will be included in the compilation.
    - If **SALLY_H** has been defined (usually by a previous inclusion of the header file), the code between **#ifndef** and **#endif** will be skipped to prevent redefinition errors.

2. **#define SALLY_H:**
    - **#define** is a preprocessor directive used to define the identifier **SALLY_H**.
    - When **#define SALLY_H** is encountered, it defines **SALLY_H** if it wasn't previously defined (in combination with **#ifndef**).
    - Defining **SALLY_H** ensures that subsequent inclusions of the same header file won't process the code inside the **#ifndef** block.

36.2 Why Sally.h becomes SALLY_H here?

In C and C++, the convention is to use all capital letters for the include guard identifier in header files and replacing certain characters (like periods or underscores) with underscores. It's a common practice to avoid naming collisions or conflicts in defining the identifier used for include guards.

The specific identifier (**SALLY_H** in this case) can be chosen arbitrarily, but it should be unique within the project to avoid clashes with other include guard identifiers. Using an all-caps naming convention and appending **_H** (indicating a header file) helps make the identifier easily identifiable as an include guard.

Tutorial 46 and 47

37. Composition (Do revise!)

Aside from functions and regular variables classes may have objects of another class as its member. Let's create two classes: Birthday and People.

In the class 'People' we will store 'Birthday' (i.e., a date) as one of its members. So, we will take an object from a 'Birthday' class and store it inside the 'People' class and run the program in 'main.cpp'.

//Birthday.h

```
#ifndef BIRTHDAY_H
#define BIRTHDAY_H

class Birthday
{
    public:
        Birthday(int d, int m, int y);
//Will pass birthday information to the constructor.
        void printDate();

    private:
    int month;
    int day;
    int year;
// Private member variables representing day, month, and
year in the Birthday class.
};
#endif
```

//Birthday.cpp

```
#include "Birthday.h"
#include <iostream>
using namespace std;

Birthday::Birthday(int d, int m, int y){
    month=m;
    day=d;
    year=y;
}
```

*//The constructor (**Birthday::Birthday()**) initializes the **day**, **month**, and **year** variables using the parameters received.*

```
void Birthday::printDate(){
    cout << day<<"/"<<month<<"/"<<year<<endl;
}
```

//People.h

```
#ifndef PEOPLE_H
#define PEOPLE_H

#include <string>
//As we are going to pass name, so we need string.
#include "Birthday.h"
//Birthday header file is need to be included, because we
will store a birthday object in this class.
using namespace std;


class People
{
    public:
        People(string x, Birthday bo);
// This constructor allows you to create a People object by
providing a name (a string) and a Birthday object as
arguments. Two arguments are there: (i) parameter of type string
named x and (ii) parameter of type Birthday named bo.
        void printInfo();
    protected:
    private:
        string name;
        Birthday dateOfBirth;
//we will use Birthday class's object 'dateOfBirth' as a
member of this class.
};
#endif
```

//People.cpp

```
#include <iostream>
#include "People.h"
#include "Birthday.h"
using namespace std;

People::People(string x, Birthday bo)
: name(x), dateOfBirth(bo){
}
//// Initializes the People object with a name and a
Birthday object using the member initializer list. When
you working with a class inside another class, you need to
use the member initializer list.

void People :: printInfo(){
    cout<<name<<"was born on ";
    //we have a function in 'Birthday' class that prints
date. We have already passed an object of 'Birthday' class
(i.e., dateOfBirth) in this 'People' class. Now, 'People'
class now has access to the 'Birthday' class's functions.
    dateOfBirth.printDate();
}
```

//main.cpp

```
#include <iostream>
#include "Birthday.h"
#include "People.h"
```

```
using namespace std;

int main(){
    Birthday birthObject(7,5,1997);
//At first, you need to make a birthday object.
    People sk("Sayak Karmakar ",birthObject);
// Creates a People object 'sk' with the name "Sayak
Karmakar" and object of Birthday class 'birthObject'.
    sk.printInfo();
}
```

```
Output=>
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
Birthday.cpp -o Birthday.o
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
People.cpp -o People.o
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
main.cpp -o main.o
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ Birthday.o
People.o main.o -o MyProgram.exe
PS C:\Users\sayak\OneDrive\Desktop\New_folder>
./MyProgram.exe
Sayak Karmakar was born on 7/5/1997
```

37.1. What is the difference between ‘bo’, ‘dateOfBirth’ and ‘birthObject’ here?

In the provided code, 'bo,' 'dateOfBirth,' and 'birthObject' are three different variables with distinct roles and scopes:

1. **bo**:
    - **bo** is a local variable within the constructor of the **People** class.
    - It is a parameter of type **Birthday** used to receive a **Birthday** object when a **People** object is created.

    - **bo** is only accessible and used within the constructor for initializing the **dateOfBirth** member variable.

2. **dateOfBirth**:
    - **dateOfBirth** is a member variable of the **People** class.
    - It is a private variable declared within the class and is used to store a **Birthday** object associated with a **People** object.
    - **dateOfBirth** is part of the class and can be accessed by any member function within the **People** class.
    - This member variable represents the birth date of a person represented by a **People** object.
3. **birthObject**:
    - **birthObject** is a local variable within the **main()** function where the program starts.
    - It is an instance of the **Birthday** class created to represent a specific date of birth.
    - **birthObject** is used to initialize the **bo** parameter when creating a **People** object in the **main()** function.

Here's how they relate to each other:

- **birthObject** is created in the **main()** function to represent a specific date of birth (e.g., July 5, 1997).
- **bo** is used as a parameter in the **People** class constructor to receive the **birthObject** when a **People** object is created. It is used for initializing the **dateOfBirth** member variable of the **People** object.
- **dateOfBirth** is a member variable within the **People** class that represents the birth date associated with a **People** object. This variable is set during the construction of a **People** object using the **bo** parameter.

So, in summary, 'bo' is a local variable used within the constructor, 'dateOfBirth' is a member variable of the **People** class, and 'birthObject' is a local variable within the **main()** function used to initialize 'bo' when creating a **People** object.

Tutorial 48

38. Friend

Every class can have friend. It is totally separate from the class, but still has access to the stuff inside the class like variables and functions.

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

class StankFist{
    public:
        StankFist(){stinkyVar=0;}
    private:
        int stinkyVar;

    friend void stinkysFriend(StankFist &sfo);
//Prototyping the function 'stinkysFriend' inside the class. As a parameter it takes object of 'StankFist' class. As we put keyword 'friend' before it, now this 'stinkysFriend' has access to eveything inside the class 'StankFist'.
};


//Now let's go and change the value of 'StinkyVariable' to some other thing to check whether this friend function can access the class members or not.
void stinkysFriend(StankFist &sfo){
    sfo.stinkyVar=99;
    cout<<sfo.stinkyVar<<endl;
}

int main(){
    StankFist bob;
//Creating object of StankFist class, because, friend function takes object of the class as argument.
    stinkysFriend(bob);
//calling the friend function.
}
```

```
Output=> 99
```

Here, **stinkysFriend** is declared outside the class, but it's been declared as a friend of the **StankFist** class. As a friend function, it has access to the private members of the **StankFist** class. As a result, the **stinkysFriend** function, despite being an independent function, can modify the private member **stinkyVar** within the **StankFist** class due to its **friend** relationship, displaying the updated value of **stinkyVar** (99) to the console.

Tutorial 49

39. this

We will learn how to print variable using 'this' keyword unlike previous way. In C++, when you use keyword 'this' it identifies a special type of pointer. This pointer stores the address of the current object that is you working with.

//Header file (Hannah.h)

```
#ifndef HANNAH_H
#define HANNAH_H

class Hannah
{
    public:
        Hannah(int); //constructor
        void printCrap(); //Member function
    private:
    int h; //private member variable
};
#endif
```

//Source file (Hannah.cpp)

```
#include <iostream>
#include "Hannah.h"
using namespace std;

Hannah::Hannah(int num)
:h(num){
}
//constructor sets the value of the data member h using the parameter num passed to the constructor. This method is using a member initializer list to initialize the member variable h with the value of num when an object of the Hannah class is instantiated.

void Hannah::printCrap(){
    cout<<"h="<<h<<endl;
    cout<<"this->h="<<this->h<<endl;
// In this particular program, 'this' stores address of 'ho' object. That object has 'h' variable in it, which we accessed.
    cout<<"(*this).h="<<(*this).h<<endl;
//this is another way. Later, we will see why 'this' is useful.
}
```

//main.cpp

```
#include <iostream>
#include "Hannah.h"
using namespace std;

int main(){
    Hannah ho(23);
//'ho' is object of class 'Hannah' that takes one argument (i.e. 'int num' type!)
    ho.printCrap();
//Calling printCrap function from Hannah class.
}
```

Output =>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe main.cpp Hannah.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> ./output.exe
h=23
this->h=23
(*this).h=23
```

Tutorial 50

40. Operator Overloading

Use the operators (like +, -, *, / etc) to behave differently.

E.g., let you have a class and you want to add two objects together. Then you can use operator overloading.

//Header file (Sally.h)

```
#ifndef SALLY_H
#define SALLY_H

class Sally
{
    public:
        int num; //variable.
        Sally();
//Declares Default constructor (no arguments)
        Sally(int);
//Declares Constructor with an integer argument.
        Sally operator+(Sally);
//Declares Overloaded addition operator + to add two Sally objects.
};
#endif
```

//Source file (Sally.cpp)

```
#include "Sally.h"
#include <iostream>
using namespace std;

Sally::Sally()
{}
//Constructor without argument. The default constructor Sally::Sally()
initializes an object of the Sally class.
```

```
Sally::Sally(int a){
    num=a;
}
```
*//The constructor **Sally::Sally(int a)** initializes **num** with the value of the argument **a**.*

```
Sally Sally::operator+(Sally aso){
    Sally brandNew;
    brandNew.num = num+aso.num;
    return(brandNew);
}
```
//See explanation in 40.1.

//main.cpp

```
#include <iostream>
#include "Sally.h"
using namespace std;

int main(){
    Sally a(34);
```
//'a' is the object of 'Sally' class with parameter=34.
```
    Sally b(21);
```
//'b' is the object of 'Sally' class with parameter=21.
```
    Sally c;
```
//'c' is the object of 'Sally' class without parameter.

```
    c=a+b;
```
*//uses the overloaded **+** operator between **a** and **b** objects. This triggers the **Sally** class's **operator+** function.*
```
    //c=a.add(b);
    cout<<c.num<<endl;
}
```

Output=>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe main.cpp Sally.cpp

PS C:\Users\sayak\OneDrive\Desktop\New_folder> ./output.exe
```

40.1 Explanation of operator overloading part

```
Sally Sally::operator+(Sally aso){
```
// The function takes a `Sally` *object as input and returns a* `Sally` *object. I think, 'Sally::operator+(Sally aso)' is the output object!*
// the first `Sally` *tells the compiler that the function is a member function of the* `Sally` *class.*
The second `Sally` *tells the compiler that the function returns a* `Sally` *object.*
:: - This is the scope resolution operator. It tells the compiler where to look for the `operator+` *function.*
`(Sally aso)`*: This part of the line declares the parameter to the function. The parameter is a* `Sally` *object called* `aso`*.*

```
    Sally brandNew;
```
// It creates a new `Sally` *object called* `brandNew`
```
    brandNew.num = num+aso.num;
```
// This line of code adds the `num` *member variables of the two* `Sally` *objects,* `this` *and* `aso`*, and assigns the result to the* `num` *member variable of* `brandNew`*.*
```
    return(brandNew);
```
// This line of code returns the new `Sally` *object,* `brandNew`*.*
```
}
```

Tutorial 52

41. Inheritance

Inherit things (functions, variables) from one class to another class. But it can inherit from public and protected portion; not from '**private**' portion.

//Mother.h

```
#ifndef MOTHER_H
#define MOTHER_H

class Mother
```
*//Mother is called '**base class**' from where you are inheriting.*

```
{
    public:
        Mother(); //Constructor
        void sayName(); //Member function


};
#endif
```

//Mother.cpp

```
#include "Mother.h"
#include "Daughter.h"
#include <iostream>
using namespace std;
Mother::Mother()
{
}

void Mother::sayName(){
    cout<<"I am Sayak"<<endl;
}
```

//Daughter.h

```
#ifndef DAUGHTER_H
#define DAUGHTER_H
#include "Mother.h"
```
//this line is needed to call 'Mother' class

```
class Mother;
```
// This line is optional. This is called 'forward declaration' of the mother class.
```
class Daughter: public Mother
```
*//To inherit things from mother class. 'Daughter' is called as '**Derived Class'**.*
```
{
    public:
        Daughter();
};
#endif
```

//Daughter.cpp

```
#include "Daughter.h"
#include "Mother.h"
#include <iostream>
using namespace std;
Daughter::Daughter()
{
}
```
*//**Note,** Daughter class doesn't have any additional methods or members beyond those inherited from **Mother**. Constructor is also empty.*

//main.cpp

```
#include <iostream>
#include "Mother.h"
#include "Daughter.h"
using namespace std;

int main(){
    Mother mom;
//creating object for Mother class
    mom.sayName();

    Daughter tina;
// creating object for Daughter class.
    tina.sayName();
//Since Daughter inherits from Mother, it can access and use the sayName() method inherited from the Mother class.
}
```

Output=>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c Mother.cpp -o Mother.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c Daughter.cpp -o Daughter.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c main.cpp -o main.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ Mother.o Daughter.o main.o -o MyProgram.exe
PS C:\Users\sayak\OneDrive\Desktop\New_folder>
./MyProgram.exe

I am Sayak

I am Sayak
```

Suggestion from AI: A virtual destructor in the base class is advisable when working with inheritance to ensure proper destruction of derived class objects.

Tutorial 53:

42. Protected Members

//Mother.h

```
#ifndef MOTHER_H
#define MOTHER_H

class Mother
//Mother is called 'base class' from where you are inheriting.
{
    public:
        int publicv;
    protected:
//accessible from inherited 'derived class' and 'friend class'; not by other classes. So, this is a mix between public and private,
        int protectedv;
    private:
        int privatev;
//only inside this class, you have access to this variable. We need to create 'method/ member function' in 'public' part to access this variable outside.

};
#endif
```

//Mother.cpp

```
#include "Mother.h"
#include "Daughter.h"
#include <iostream>
using namespace std;
```

//Note:

```
int publicv;
int protectedv;
int privatev;
```

→ *this is not the correct way to* ***initialize variable****. The actual initialization or assignment of values for these variables will occur within methods or the constructor defined in the source file (Mother.cpp) See below:*
In Mother.h, add this thing inside the class.

```
public:
        Mother(); // Constructor declaration
```

and in Mother.cpp, define the constructor.

```
Mother::Mother() {
    publicv = 0;
    protectedv = 0;
    privatev = 0;
}
```

*But in this code, we don't need any method or constructor in Mother class to initialize its variables. Because, 'Daughter' class already has a function named '****doSomething****()' which itself initialize the variables from 'Mother' class (as it has access to Mother's variables).*

//Daughter.h

```
#ifndef DAUGHTER_H
#define DAUGHTER_H
#include "Mother.h"
class Daughter : public Mother
//Daughter is inheriting from mother
{
    public:
        void doSomething();
};
#endif
```

//Daughter.cpp

```
#include "Daughter.h"
#include "Mother.h"
#include <iostream>
using namespace std;
void Daughter::doSomething(){
    publicv=1;
//'Daughter' accessing public variable.
    protectedv=2;
//'Daughter' accessing protected variable.
    cout<<"public variable="<<publicv<<" and protected
variable="<<protectedv<<endl;
    //privatev=3; -->cannot access private variable.
}
```

//main.cpp

```
#include <iostream>
#include "Mother.h"
#include "Daughter.h"
using namespace std;

int main(){
    Daughter tina;
    tina.doSomething();
}
```

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
Mother.cpp -o Mother.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
Daughter.cpp -o Daughter.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
main.cpp -o main.o
```

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ Mother.o Daughter.o main.o -o MyProgram.exe

PS C:\Users\sayak\OneDrive\Desktop\New_folder> ./MyProgram.exe

public variable=1 and protected variable=2
```

Tutorial 54

43. Derived class Constructors and Destructors

Derived class does not inherit constructors and destructors from the base class. However, when you create an object of the derived class, the constructors and destructors of both the base and derived classes are invoked automatically as part of the object creation and destruction process.

The constructors and destructors for both the base and derived classes are called in the following order:

1. The base class constructor.
2. The derived class constructor.
3. The derived class destructor.
4. The base class destructor.

//Mother.h

```
#ifndef MOTHER_H
#define MOTHER_H

class Mother {
    public:
        Mother(); //Constructor
        ~Mother(); //Destructors.
    protected:
    private:
};
#endif
```

//Mother.cpp

```
#include "Mother.h"
#include "Daughter.h"
#include <iostream>
using namespace std;

Mother::Mother() {
    cout<<"I am the mother Constructor!"<<endl;
}

Mother::~Mother() {
    cout<<" This is the mother destructor!"<<endl; //This
runs after the object gets destroyed in main.cpp program.

}
```

//Daughter.h

```
#ifndef DAUGHTER_H
#define DAUGHTER_H
#include "Mother.h"
class Daughter : public Mother
{
    public:
        Daughter();
        ~Daughter();
};
#endif
```

//Daughter.cpp

```
#include "Daughter.h"
#include "Mother.h"
#include <iostream>
using namespace std;

Daughter::Daughter() {
    cout<<"I am the Daughter Constructor!"<<endl;
}

Daughter::~Daughter() {
    cout<<"This is the Daughter destructor!"<<endl; //This
runs after the object gets destroyed in main.cpp program.
}
```

//main.cpp

```
#include <iostream>
#include "Mother.h"
#include "Daughter.h"
using namespace std;

int main(){
    Daughter tina; //object of 'Daughter' class
}
```

Output =>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
Mother.cpp -o Mother.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder>  g++ -c
Daughter.cpp -o Daughter.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -c
main.cpp -o main.o

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ Mother.o
Daughter.o main.o -o MyProgram.exe

PS C:\Users\sayak\OneDrive\Desktop\New_folder>
./MyProgram.exe

I am the mother Constructor!

I am the Daughter Constructor!

This is the Daughter destructor!

 This is the mother destructor!
```

Tutorial 55 and 56

44. Polymorphism

Polymorphism allows you to have a function and have many different outcomes.

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

class Enemy{
    protected:
        int attackPower;
    public:
        void setAttackPower(int a){
            attackPower=a;
        }
};
```

*//The **Enemy** class has an **attackPower** data member that's protected, and it has a public method **setAttackPower()** to set the **attackPower**.*

```
class Ninja: public Enemy{
    public:
        void attack(){
            cout<<"I am a Ninja, ninja cop!-" <<
attackPower << endl;
        }
};
```

*//Here, the **Ninja** class is defined, inheriting publicly from the **Enemy** class. The **Ninja** class has a method **attack()** that prints a message indicating it's a ninja and displays the **attackPower** of the Ninja.*

```
class Monster: public Enemy{
    public:
        void attack(){
            cout<<"The monster over here. I will eat you.
-"<<attackPower<<endl;
        }
};
```

*//Above segment defines the **Monster** class, which also inherits publicly from the **Enemy** class. The **Monster** class contains an **attack()** method that prints a message indicating it's a monster and displays the **attackPower** of the Monster.*

```
int main(){
    Ninja n;
    Monster m;
    Enemy *enemy1 = &n;
    Enemy *enemy2 = &m;
```

*//Pointers of type **Enemy** are created (**enemy1** and **enemy2**) to store the addresses of the **Ninja** and **Monster** objects, respectively.*

```
    enemy1-> setAttackPower(29);
    enemy2-> setAttackPower(99);
```

*// The -> operator is used to access a member of an object through a pointer. Even though **enemy1** is declared as a pointer to the base class **Enemy**, it's pointing to an actual **Ninja** object. In this case, it's used to call the **setAttackPower()** method on **Ninja** object n. **This demonstrates polymorphism, where you can use a base class pointer to interact with derived class objects.***

```
    n.attack();
    m.attack();
}
```

```
Output =>

I am a Ninja, ninja cop!-29
The monster over here. I will eat you. -99
```

Let’s go for another example which clearly demonstrates polymorphism.

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

class Enemy{
    public:
        virtual void attack(){

            cout<<"I am the enemy class"<<endl;
        }
    };
```

*//The **virtual** keyword used in the **Enemy** class before the **attack()** method declaration means that any derived class can provide its own implementation of the **attack()** method, enabling polymorphism.*

```
class Ninja: public Enemy{
    public:
        void attack(){
            cout<<"ninja attack!"<<endl;
        }
};

class Monster: public Enemy{
    public:
        void attack(){
            cout<<"Monster attack!"<<endl;
        }
};
```

*//Above portions define the **Ninja** class and **Monster** class as two derived classes from **Enemy**. It overrides the **attack()** function to output "ninja attack!" and "Monster attack!" respectively when called.*

```
int main(){
    Ninja n;
    Monster m;
    Enemy *enemy1 = &n;
    Enemy *enemy2 = &m;
```

*//Pointers of type **Enemy** are created (‘**enemy1**’ and ‘**enemy2**’), which store the addresses of **Ninja** and **Monster** objects (‘**n**’ and ‘**m**’), respectively.*

```
    enemy1->attack();
    enemy2-> attack();
```

*//Calls the **attack()** method using the base class pointers **enemy1** and **enemy2**. Since the **attack()** method is marked as **virtual**, the appropriate method based on the object they point to will be called due to polymorphism. These will call **attack**() method from ‘**Ninja**’ and ‘**Monster**’ class; not from ‘**Enemy**’ class. If you want to call **attack**() method from ‘**Enemy**’ class; you need to create object of that and then call. see below:*

```
     Enemy e;
```

```
    Enemy *enemy3 =&e;
    enemy3->attack();

}
```

```
Output =>

ninja attack!
Monster attack!
I am the enemy class
```

Tutorial 57

45. Abstract Classes and pure Virtual Functions

First note one thing. In the previous code (tutorial 56), if you do not define ‘attack()’ method in derived classes ‘Ninja’ and ‘Monster’; then the ‘attack()’ method will be automatically called from ‘base class’ ‘Enemy’. This is a **regular virtual function**. See below:

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

class Enemy{
    public:
        virtual void attack(){
            cout<<"I am the enemy class"<<endl;
        }
    };


class Ninja: public Enemy{
};
class Monster: public Enemy{
};
//No definition ‘attack()’ method in derived classes ‘Ninja’ and ‘Monster’.


int main(){
    Ninja n;
    Monster m;
    Enemy *enemy1 = &n;
    Enemy *enemy2 = &m;
    enemy1->attack();
    enemy2-> attack();
}
```

```
Output =>
I am the enemy class
I am the enemy class
```

But above one is not a ‘Pure virtual Function’.

Pure virtual function does not give you option to inherit function value from the base class. It does not have a body at all. So, every ‘derived class’ **must** overwrite the function of the base class. Otherwise, you will get error message. The class with pure virtual function is called as ‘**Abstract Class**’.

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

class Enemy{
    public:
        virtual void attack()=0;
//It has no body. This is called as pure virtual function.
    };

class Ninja: public Enemy{
    public:
        void attack(){
            cout<<"ninja attack!"<<endl;
        }
};

class Monster: public Enemy{
    public:
        void attack(){
            cout<<"Monster attack!"<<endl;
        }
};

int main(){
    Ninja n;
    Monster m;
    Enemy *enemy1 = &n;
    Enemy *enemy2 = &m;
    enemy1->attack();
    enemy2-> attack();
}
```

```
Output=>

ninja attack!
Monster attack!
```

Tutorial 58

46. Function Templates

Let us make a code to add two integers.

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

int addCrap(int a , int b){
    return a+b;
}

int main(){
    int x=7, y=34, z;
    z=addCrap(x,y);
    cout << z << endl;
}
```

```
Output >>
41
```

Now, question is, can we build functions that can handle multiple types of data at the same time (int, float, double etc)??

Ans: Yes!

To do this, we need something called template definition. See below:

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

template <class bucky>
//Now, bucky is a generic type of data. It is not just an integer, float, double or
character! It can handle everything.
bucky addCrap(bucky a , bucky b){
    return a+b;
}

int main(){
    int x=7, y=34, z;
    z=addCrap(x,y);
    cout << z << endl;

    double a=7.55, b=34.78, c;
    c=addCrap(a,b);
    cout << c << endl;
}
```

```
Output >>

41
42.33
```

46.1. what is the difference between ‘double’ and ‘float’ datatypes?

Both double and float are data types used in programming to represent floating-point numbers, which are numbers that can have decimal places. However, there are some key differences between the two types.

Feature	Double	Float
Precision	15 decimal places	6-7 decimal places
Storage	64 bits	32 bits
Performance	More computationally expensive	Less computationally expensive
Usage	Scientific calculations, financial applications	Graphics, game programming

Tutorial 59

46.2. Function Templates with Multiple Parameters

If we want to work with different types of data in the same function, we need more than one generic classes. If we want specific type of data (say double) as function return of a templated function, we should pass the value of the argument in main() function (that has same generic class name as that of function) with that specific data type. See below:

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

template <class FIRST, class SECOND>
    FIRST smaller (FIRST a, SECOND b){
        return (a<b?a:b);
    }

int main(){
    double x=99.3;
    int y=107;
    cout<<smaller(x,y)<<endl;
}
```

```
Output >>

99.3
```

In this above example, we wanted function output as a ‘double’ type, So in main() function, we provided a double value to the first argument. See another one:

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

template <class FIRST, class SECOND>
    FIRST smaller (FIRST a, SECOND b){
        return (a<b?a:b);
    }

int main(){
    int x=99;
    double y=10.23;
    cout<<smaller(x,y)<<endl;
}
```

```
Output >>

10
```

Here, first argument provided in main() was an int type data. So, ‘smaller’ function will return a ‘int’ value.

Tutorial 60

47. Class Templates

This example demonstrates how to create a templated class and use it with a specific data type (**double** in this case) by instantiating the class with that type and calling the member functions accordingly. The code allows the use of the same class with different data types without rewriting the class for each specific type.

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

template <class T>
//Bucky is a template class with a generic type T.
class Bucky{
    T Sayak , Soham;
//It has two member variables, Sayak and Soham, both of type T.
    public:
        Bucky(T a, T b){
            Sayak=a;
            Soham=b;
        }
//Constructor Definition. The constructor initializes these member variables
with the values passed as arguments of type T.
        T bigger();
//The class includes a function declaration bigger() that will return a value of
type T.
};


//We are now giving function definition of function
bigger() of Bucky class at outside the class.
template <class T>
//We need function template again to define every single function.
T Bucky<T>::bigger(){
//Pattern:  "return type" "classname" "<return type>"::"function name". That
blank generic data type "<return type>" is required after class name because
we need to tell C++ that functions template parameter is the same one that we
are using for class.
    return (Sayak>Soham?Sayak:Soham);
//This is called Ternary operator.
}


int main(){
    Bucky <double> bo(69.32,105.2);
//When we work with class templates, we need to explicitly tell the object what
type of data we want to substitute for 'T'.
    cout<<bo.bigger();
}
```

```
Output >>

105.2
```

Tutorial 61

48. Template Specializations

It is a way to make a class that has a different implementation when a specific type is passed into it. Let's compare between regular generic template and specialized template.

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

template <class T>
class Spunky{
//This is generic template class. This class will handle every kind of numeric
data besides the characters; because function for 'char' type of data was
specially mentioned in another class below. Otherwise, this also could have
executed for 'char' type arguments.
    public:
        Spunky(T x){
            cout<<x<<" is not a character"<<endl;
        }
};


template<>
class Spunky<char>{
//This class will handle only 'character' type of data. This is of specialized
template class.
    public:
        Spunky(char x){
            cout<<x<<" is indeed a character!"<<endl;
        }
};
```

```
int main(){
    Spunky <int> obj1(7);
    Spunky<double> obj2(3.45);
    Spunky<char> ('A');
}
```

```
Output >>

7 is not a character
3.45 is not a character
A is indeed a character!
```

Tutorial 62 and 63

48. Exceptions Handling

This the way of handling the error that might occur when program is running. Like if some number is divided by zero, when a person provides his age more than his mother etc. It allows programmers to isolate the code that handles errors from the code that performs the normal flow of the program. This makes the code more robust and easier to maintain.

48.1. Example 1

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

int main(){
    try{
//This line marks the beginning of a try block. A try block is used to enclose code that may throw an exception.
        int momsAge=30;
        int sonsAge=34;
        if (sonsAge>momsAge){
        throw 99;
        }
    }catch(int x){
```

*//This line marks the beginning of a **catch** block. A **catch** block is used to handle exceptions thrown by a **try** block. If there is no error, ‘**catch**’ block is simply skipped.*

```
    cout<<"son cannot be greater than mom, ERROR number:
"<< x <<endl;
}
}
```

```
Output >>

son cannot be greater than mom, ERROR number: 99
```

48.2. Example 2

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

int main(){
    try{
        int num1;
        cout<<"Enter the first number: " << endl;
        cin>>num1;
```

//This line prompts the user to enter the first number and then stores the input in the `num1` variable.

```
        int num2;
        cout<<"Enter the second number: " << endl;
        cin>>num2;

        if (num2 == 0){
            throw 0;
        }
        cout <<"result= " << num1 / num2 << endl;
        }
catch(...){
```

//This line catches any type of exception. The ellipsis (...) operator tells the compiler that we are willing to catch any type of exception.

```
    cout<<"You can't divide by 0"<<endl;
}
}
```

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe 62.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\output.exe
Enter the first number:
98
Enter the second number:
14
result= 7
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe 62.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\output.exe
Enter the first number:
45
Enter the second number:
0
You can't divide by 0
```

Tutorial 64, 65

49. Working with files

The code is a simple example of how to write to a file in C++.

```cpp
#include <iostream>
#include <fstream>
//This header file is needed for file operations.
using namespace std;

int main(){
    ofstream sayakFile;
```

*//**ofstream** is a class in the C++ standard library that is used to write to files. It is a derived class of the **ostream** class, which is the base class for all output streams. The **ofstream** class provides a number of functions for writing data to files, such as the **open**() function, the **write**() function, and the **close**() function. To use the ofstream class, you first need to create an **ofstream** object (here, **sayakFile** is the object). Then, you can use all these functions.*

```
    sayakFile.open("tuna.txt");
//Using open() function to open a file. If the file does not exist, it creates a new file in the same directory.
    sayakFile << "I love to play Badminton.\n";
// "<<" operator will write this string to 'sayakFile'.
    sayakFile.close();
//Closing the object to free the computer's memory.
}
```

```
Output >>

No output in console. The string was added to the file named tuna.txt.
```

49.1 Tips for File Handling

Other than above mentioned way, there is a better way to associate object to the file. It takes only one line. This code also shows whether a file is open (or file is associated) with the created object or not. This is important, because file must be open to make any changes to the file.

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

int main(){
    ofstream buckyFile("Sally.txt");
//Passing the file name into the constructor. it opens or creates the file.
    if(buckyFile.is_open()){
//This line checks if the file is open using the is_open() member function.
        cout << "The desired file is open\n";
    }else{
        cout<<"Sayak you messed up" <<endl;
    }
    buckyFile << "Sally is a good girl\n";
    buckyFile.close();

    if(buckyFile.is_open()){
```

```
//After "buckyFile.close()", "is_open" function returns false.
        cout << "How the object is not closed!!Some
problem is there\n";
    }else{
        cout<<"Congrats! You successfully closed the
object\n" <<endl;
    }
}
```

```
Output >>
The desired file is open
Congrats! You successfully closed the object
```

Tutorial 66

49.2 Writing Custom File Structures

This C++ code opens an output file stream to a file named "**players.txt**" and prompts the user to input player information including ID, name, and money. The program then writes this information into the file until the user stops by inputting Ctrl+Z (on Windows) or Ctrl+D (on Unix-based systems), assuming ID is an integer, name is a string (without spaces), and money is a double.

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

int main(){
    ofstream CricketObj("players.txt");
//declares an output file stream object named CricketObj and opens a file
named "players.txt" for writing.

if (!CricketObj.is_open()) {
        cout << "Error opening the file." << endl;
        return 1;
    }
// This line checks if the file "players.txt" failed to open. If the file opening fails,
it displays an error message and exits the program by returning 1.
```

```
    cout << "Enter Players ID, names and Money "<< endl;
    cout << "Press Ctrl+Z (on Windows) or Ctrl+D (on Unix-
based systems) to quit\n"<<endl;

    int idNumber;
    string name;
    double money;
```

*// Declares variables **idNumber**, **name**, and **money** to store the player's ID, name, and money, respectively.*

```
    while (cin>>idNumber>>name>>money){
```

// Enters a loop that reads input from the user for ID, name, and money. It expects these values to be entered separated by spaces or newlines. It continues until the input fails (for instance, when Ctrl+Z or Ctrl+D is entered to signify the end of input).

```
        CricketObj << idNumber << ' '<<name<<'
'<<money<<endl;
```

// Writes the entered data (ID, name, and money) to the "players.txt" file.

```
    }
    CricketObj.close();
    cout << "Data written to players.txt. Program ended."
<< endl;
    return 0;
}
```

Output in console >>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o
output.exe 66.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\output.exe
Enter Players ID, names and Money
Press Ctrl+Z (on Windows) or Ctrl+D (on Unix-based systems) to
quit
1 Sachin 123.21 2 Sourav 234.5
3 Dravid
234.7^Z
Data written to players.txt. Program ended.
PS C:\Users\sayak\OneDrive\Desktop\New_folder>
```

```
Text added in player.txt >>
1 Sachin 123.21
2 Sourav 234.5
3 Dravid 234.7
```

Tutorial 67

49.3. Reading Custom File Structures

The provided C++ code reads data from a file named "players.txt" using an input file stream (**ifstream**). It then retrieves and stores this data in variables **idNumber**, **name**, and **money** and finally displays in console.

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

int main(){
    ifstream CricketObj("players.txt");
//'ifstream' allows reading data from a file. This line declares an input file stream object named CricketObj and opens the file named "players.txt" for reading. If the file does not exist or cannot be opened, it may cause issues with reading the file contents. Also note that, 'ofstream' cannot be used because this is for outputting or writing to a file.

if (!CricketObj.is_open()) {
        cout << "Error opening the file." << endl;
        return 1;
    }


    int idNumber;
    string name;
    double money;
```

*// These lines declare variables **idNumber**, **name**, and **money**. These variables will hold the data read from the file.*

```
    while(CricketObj >>idNumber>>name>>money){
```
*// This loop uses the file input stream (i.e., **CricketObj**) to read data and assigns it to the variables **idNumber**, **name**, and **money**.*
```
        cout << idNumber << ' '<<name<<' '<<money<<endl;
```
*// This line displays the data that has been read from the file onto the standard output (usually the console). It prints the **idNumber**, **name**, and **money**, separated by spaces and followed by a newline character, effectively displaying each set of data from the file on a new line in the console.*
```
}
}
```

```
Output >>

1 Sachin 123.21
2 Sourav 234.5
3 Dravid 234.7
```

Tutorial 68, 69, 70

49.4. Cool program working with file

```
#include <iostream>
#include <fstream> //required to access outside files.
using namespace std;

int getWhatTheyWant();
void displayItems(int x); //prototype of the function.

//main function
int main(){
    int whatTheyWant;
    whatTheyWant = getWhatTheyWant();
```
//getWhatTheyWant function gets 1/2/3/4 inputs from keyboard and stores it in the variable 'whatTheyWant'.

```
    while(whatTheyWant != 4){
        switch(whatTheyWant){
            case 1:
                displayItems(1);
                break;
            case 2:
                displayItems(2);
                break;
            case 3:
                displayItems(3);
                break;
        }
        whatTheyWant = getWhatTheyWant();
```

*// After processing the user's choice, it gets the next choice from the user by calling **getWhatTheyWant()**. This will be a loop until user enters 4.*

```
    }

}

//getWhatTheyWant function
int getWhatTheyWant(){
    int choice;
    cout << "1 -just plain items" << endl;
    cout << "2 -helpful items" << endl;
    cout << "3 -harmful items" << endl;
    cout << "4 -quit program" << endl;
```

*// **getWhatTheyWant()**: This function displays a menu to the user and prompts them to enter a choice.*

```
    cin >> choice;
    return choice;
```

// Reads the user's choice from the standard input and returns it. So, getWhatTheyWant() function returns 1/2/3/4.

```
}

//displayItems functions. (These functions will just print
out something, no return required. So, void functions)
void displayItems(int x){
    ifstream objectFile("objects.txt");
    string name;
    double power;
```

//making two variables to store values from "objects.txt"

```
    if(x==1){
        while(objectFile>>name>>power){
            if(power==0){
                cout<<name<<' '<<power<<endl;
            }
        }
    }
    if(x==2){
        while(objectFile>>name>>power){
            if(power>0){
                cout<<name<<' '<<power<<endl;
            }
        }
    }
    if(x==3){
        while(objectFile>>name>>power){
            if(power<0){
                cout<<name<<' '<<power<<endl;
            }
        }
    }
```

*// Depending on the value of **x**, it prints items with certain characteristics: If **x == 1**, it prints items with power equal to 0. If **x == 2**, it prints items with power greater than 0. If **x == 3**, it prints items with power less than 0.*

```
}
```

Output >>

```
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe 68.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\output.exe
1 -just plain items
2 -helpful items
3 -harmful items
4 -quit program
1
chair 0
```

```
shoe 0
pencil 0
1 -just plain items
2 -helpful items
3 -harmful items
4 -quit program
2
fruit 43
candy 87
soda 32
1 -just plain items
2 -helpful items
3 -harmful items
4 -quit program
3
ninja -54
math -44
dirtyneedle -99
1 -just plain items
2 -helpful items
3 -harmful items
4 -quit program
4
PS C:\Users\sayak\OneDrive\Desktop\New_folder>
```

Tutorial 71, 72

50. String Class and Functions

50.1. Print a user input string (one word)

The following C++ code reads a string from the user using **cin** and prints the entered string. But, when you are using **'cin'** to read data from keyboard, user or file; end of the input is determined by first whitespace character it comes across. So, if you enter a sentence, as an output you only can get the first word.

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

int main(){
    string bucky;
    cout<<"Enter the text you want to print: "<<endl;
    cin >> bucky;
    cout<<"The string you entered is: " << bucky<< endl;
    }
```

```
Output >>

PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe test.cpp

PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\output.exe

Enter the text you want to print:

Indian Institute of Technology

The string I entered is: Indian
```

50.2. Print a user input string (entire line)

To print the entire sentence, we should use getline() function. See below:

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main(){
    string x;
    cout<<"Enter the text you want to print: "<<endl;
    getline(cin,x);
```
*//Uses **getline(cin, x)** to read an entire line of text from the standard input (usually from the keyboard) and assigns it to the variable **x**. Unlike **cin >> x;**, **getline** reads the entire line, including spaces.*
```
    cout<<"the string I entered is: " << x<< endl;
    }
```

```
Ouput >>
PS C:\Users\sayak\OneDrive\Desktop\New_folder> g++ -o output.exe
71.cpp
PS C:\Users\sayak\OneDrive\Desktop\New_folder> .\output.exe
Enter the text you want to print:
Indian Institute of Technology
the string I entered is: Indian Institute of Technology
```

50.3 Copy Strings

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

int main(){
    string s1("hampster");
```
*//Initializes **s1** with the value "hamster".*
```
    string s2;
    string s3;

    s2=s1;
```
*//Copies the contents of **s1** into **s2** using the assignment operator (**=**).*
```
    s3.assign(s1);
```
*//Copies the contents of **s1** into **s3** using the **assign** member function.*
```
    cout<<"s1: "<<s1<<", "<<"s2: "<<s2<<", "<<"s3:
"<<s3<<endl;
    }
```

```
Output >>
s1: hampster, s2: hampster, s3: hampster
```

50.4. Strings are Arrays!

Strings are arrays of characters. So, anything you can do to an array, you can do to a string.

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

int main(){
    string s1="Floccinaucinihilipilification";
    cout<<s1.at(0)<<endl;
    cout<<s1.at(5)<<endl;
//Uses the at() function to access and print individual characters at specific positions in the string s1.

    for (int x=0;x<s1.length();x++){
        cout << s1.at(x);
    }
//Initiates a for loop that iterates through each character in the string using the loop variable x. The loop prints each character using s1.at(x).

}
```

```
Output >>

F
i
Floccinaucinihilipilification
```

50.5 Substring Functions

Substring is a part from a bigger string.

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

int main(){
    string s1("OMG! only one tutorial is left!");
    cout<<s1.substr(10,7)<<endl;
//Substring function.First argument of substr() takes the index of beginning
character and 2nd argument implies how many characters you want to go after
that. Remember, whitespace is also considered as a character.
}
```

```
Output >>

one tut
```

50.6 swap function

The **swap()** function is used to exchange the contents of two strings. See below:

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

int main(){
    string one("apples");
    string two("beans");
    cout<<"one= "<<one<<", "<<"two= "<<two<<endl;

    one.swap(two);
    cout<<"one= "<<one<<", "<<"two= "<<two<<endl;
}
```

```
Output >>

one= apples, two= beans
one= beans, two= apples
```

50.7 Finding a substring in a string

First, we will see how to find the index of first and last occurrences of a substring.

```
#include <iostream>

#include <string>
using namespace std;

int main(){
    string s1("I am amazed at how amazing the program
is!");
    cout<<s1.find("am")<<endl;
//These outputs the index of first occurrence of sub-string 'am'.
    cout<<s1.rfind("am")<<endl;
//'rfind' Stands for reverse find. This outputs the index of last occurrence of sub-
string 'am'.
}
```

```
Output >>

2
36
```

Now, we will see how to find the index of all the occurrences of a substring.

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

int main() {
    string s1("I am amazed at how amazing the program
is!");

    size_t found = s1.find("am");
```

*// Uses the **find** function to locate the first occurrence of the substring "am" in the string **s1**. The result is stored in the variable **found**. If "am" is not found, **found** is set to **string::npos** (a special constant representing "no position").*

```
    while (found != string::npos) {
```
*// Initiates a **while** loop that continues as long as there are more occurrences of "am" (**found** is not equal to **string::npos**).*

```
        cout << "Found at index: " << found << endl;
```
// Prints the index of the found occurrence.

```
        found = s1.find("am", found + 1);
```
*// Updates the **found** variable to search for the next occurrence of "am" starting from the position immediately after the current occurrence. This ensures that the loop finds all occurrences in the string.*
```
    }

    return 0;
}
```

```
Output =>

Found at index: 2
Found at index: 5
Found at index: 19
Found at index: 36
```

50.8 String erase, replace and insert functions

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

int main(){
    //Erase Function
    string s1("I am amazed at how amazing the program
is!");
    cout<<s1<<endl;
    s1.erase(20);
```
//This will erase characters from index=20 (including) upto last.
```
    cout<<s1<<endl;
    //replace function
    string s2("My name is Sayak and I love Samosa!");
    cout<<s2<<endl;
    s2.replace(11, 5, "Partha");
```
//We want to replace 'Sayak' with 'Partha'. The arguments are → arg1: The index number of 'S' in 'Sayak'; arg2: How many characters do you want to replace after that; arg3: With which word you want to replace with.
```
    cout<<s2<<endl;

    //find and replace (try later)!!!

    //insert function
    string s3("I study in Kharagpur");
    cout<<s3<<endl;
    s3.insert(11,"IIT ");
```
//arg1: At which index position you want to insert the new substring. arg2=the new substring you want to insert.
```
    cout<<s3<<endl;
}
```

```
Output >>

I am amazed at how amazing the program is!
I am amazed at how a
My name is Sayak and I love Samosa!
My name is Partha and I love Samosa!
I study in Kharagpur
I study in IIT Kharagpur
```