C++

- Created by Danish Computer Scientist Bjarne Stroustrup in 1979.
- It is an extension of C language.
- It is used for writing fast programs.
- It gives you more control over system resources.
- It has high memory management.
- High performances.

Low Level & High Level Language

Low-level languages are closer to the computer's hardware and provide more control but are more difficult to understand, while high-level languages are easier to learn and use, providing greater abstraction and portability.

Data Types

Data types defined the type of data a variable can hold, for example an integer variable can hold integer data, a float variable can hold float data and so on.

Rules :

- Variable name in C++ can range from 1 - 255 characters.
- All variable names must begin with an alphabet or an underscore.
- Variable names are case sensitive for example int a and int A, both will be treated as different variable.
- No spaces or special characters allowed.

int	5 , 1 , 3, 7
float	5.99 , 1.25 , 2.87
double	99.98
char	'D' , ‘A’ , ‘g’
bool	true / false
string	"Hello World"

There are typically three categories of data types: built-in data types, user-defined data types, and derived data types.

Built-in Data Types: Built-in data types, also known as primitive data types, are provided by the programming language itself. They are typically predefined and supported natively by the language. Examples of built-in data types include:

- **Integer:** Used to store whole numbers (e.g., 1, 2, -3).
- **Floating-point:** Used to store decimal numbers (e.g., 3.14, -0.5).
- **Boolean:** Used to store true or false values.
- **Character:** Used to store individual characters (e.g., 'a', 'B', '$').
- **String:** Used to store sequences of characters (e.g., "Hello", "World").

User-defined Data Types: User-defined data types are created by the programmers themselves to suit the needs of their applications. These data types are defined using existing data types or a combination of built-in and user-defined data types. Examples of user-defined data types include:

Structure: A collection of related variables grouped together under a single name.

Enumerations: A set of named constant values that represent a collection of related values.

Class: A blueprint for creating objects that encapsulates data and behavior.

Derived Data Types: Derived data types are created by applying operations or transformations to existing data types. These data types derive their properties and behaviors from the underlying data types. Examples of derived data types include:

Arrays: A collection of elements of the same data type, accessed using an index.

Pointers: Variables that store the memory addresses of other variables.

Functions: A sequence of instructions that can be called and executed.

Comments

```
// Single line comments
```
→ For Single Sentences

```
/*
Multi line comment
*/
```
→ For Mutiple Sentences

Variables

Variables are like containers to store data in computer memory.

Syntax

// Syntax

```
#include<iostream> // Header File

int main(){
   std::cout<<"new york city";
   return 0;
}
```

Basic input/output in C++

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

int main(){

   int num1, num2;
   cout<<"enter the value of num1 : " ; // '<<' is called insertion operator
   cin>>num1;  // '>>' is called extraction operator

   cout<<"enter the value of num2 : " ;
   cin>>num2;

   cout<<"the sum is : "<< num1+num2;

   return 0;
}
```

Operators in C++

In C++, operators are symbols that are used to perform operations on data.

Arithmetic Operators

- Addition (+): Adds two operands.
- Subtraction (-): Subtracts the second operand from the first.
- Multiplication (*): Multiplies two operands.
- Division (/): Divides the first operand by the second.
- Modulus (%): Returns the remainder after division.

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

int main(){

   int a = 5;
   int b = 2;

   cout<<"The value of a + b is : "<< a + b <<endl;
   cout<<"The value of a - b is : "<< a - b <<endl;
   cout<<"The value of a * b is : "<< a * b <<endl;
   cout<<"The value of a / b is : "<< a / b <<endl;
   cout<<"The value of a % b is : "<< a % b <<endl;

return 0;
}
```

Operator Precedence

```
First : ( )

Second : * , /, %

Third : +, -
```

Assignment Operator

- Assignment (=): Assigns a value to a variable.
- Compound Assignment (+=, -=, *=, /=, %=): Performs an operation and assigns the result to the left operand.

Increment & Decrement Operator

- Increment (++) : Increases the value of a variable by 1.
- Decrement (--) : Decreases the value of a variable by 1.

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

int main(){

   int a = 5;
   int b = 2;


    cout<<"The value of a++ is : "<< a++ <<endl; // First print then increment
    cout<<"The value of a-- is : "<< a-- <<endl; // First print then decrement

    cout<<"The value of ++a is : "<< ++a <<endl; // First increment then print
    cout<<"The value of --a is : "<< --a <<endl; // First decrement then print

return 0;
}
```

Comparison Operator

- Equality (==): Checks if two operands are equal.
- Inequality (!=): Checks if two operands are not equal.
- Greater than (>): Checks if the first operand is greater than the second.
- Less than (<): Checks if the first operand is less than the second.
- Greater than or equal to (>=): Checks if the first operand is greater than or equal to the second.
- Less than or equal to (<=): Checks if the first operand is less than or equal to the second.

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

int main(){

   int a = 5;
   int b = 2;


   cout<<"The value of a==b is : "<< (a==b) <<endl;
   cout<<"The value of a!=b is : "<< (a!=b) <<endl;
   cout<<"The value of a>=b is : "<< (a>=b) <<endl;
   cout<<"The value of a<=b is : "<< (a<=b) <<endl;
   cout<<"The value of a<b is : "<< (a<b) <<endl;
   cout<<"The value of a>b is : "<< (a>b) <<endl;

return 0;
}
```

Logical Operators

- Logical AND (&&): Returns true if both operands are true.
- Logical OR (||): Returns true if either of the operands is true.
- Logical NOT (!): Inverts the logical state of an operand.

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

int main(){

   int a = 5;
   int b = 2;


   cout<<"The value of ((a==b) && (a<b)) is : "<<((a==b) && (a<b))<<endl;
   cout<<"The value of ((a==b) || (a<b)) is : "<<((a==b) || (a<b))<<endl;
   cout<<"The value of (!(a==b)) is : "<<(!(a==b))<<endl;

return 0;
}
```

Reference Variables

When you declare a reference variable, you essentially create an alias or another name for an existing variable.

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

int main(){

   float x = 455;
   float &y = x;
   cout<<x<<endl;
   cout<<y<<endl;

return 0;
}

----------------------------------------------------------------------------------------

// Output of both x and y will be 455
```

Typecasting

Type casting, also known as type conversion, is the process of changing the data type of a variable from one type to another. It allows you to convert a value from one data type to another compatible data type.

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

int main(){

   float a = 7.9;

   cout<<"The value of float a is : "<<(int)a<<endl;
   cout<<"The value of float a is : "<<int(a)<<endl;

return 0;
}

----------------------------------------------------------------------------------------

// Output will be as an int, 7 because the data type was converted from float to int
```

Constants

Constants are used to define values that cannot be modified or changed during the execution of a program.

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

int main(){

const float pi = 3.14;
float pi = 3.17;

cout<<pi<<endl;

return 0;
}

----------------------------------------------------------------------------------------

// Here the program will give you an error at execution because here we have declared pi as a (const) means, it's value will remain constant
```

Manipulators

In C++, manipulators are functions or objects that can be used to modify the behavior of input/output streams. They are used to control various aspects of formatting, such as setting field widths, controlling precision, and manipulating the alignment of data.

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

int main(){

int a=7, b=127, c=36000;

cout<<"The value of a is : "<<a<<endl;
cout<<"The value of b is : "<<b<<endl;
cout<<"The value of c is : "<<c<<endl;

cout<<"The value of a is : "<<setw(5)<<a<<endl;
cout<<"The value of b is : "<<setw(5)<<b<<endl;
cout<<"The value of c is : "<<setw(5)<<c<<endl;

return 0;
}

----------------------------------------------------------------------------------------

// The value of a is : 7
// The value of b is : 127
// The value of c is : 36000
// The value of a is :     7
// The value of b is :   127
// The value of c is : 36000

// Here every number is occupying a space of 5
```

Control Structures

In C++, control structures are used to alter the flow of program execution based on certain conditions.

Conditional Statements

- If statement: It allows you to execute a block of code if a given condition is true.
- If-else statement: It allows you to execute one block of code if a condition is true and another block if it's false.
- Nested if-else statement: It is a combination of if and else statements where an if statement is nested inside another if or else statement.

EXAMPLE 1

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

int main(){

int a = 20;
int b = 18;

if (20 < 18) {
  cout << "20 is greater than 18";
}
else {
  cout << "20 is not greater than 18";
}

return 0;
}
```

EXAMPLE 2

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

int main(){

    int age;

    cout<<"Enter your age : ";
    cin>>age;

    if(age > 18){
        cout<<"you are an adult"<<endl;
    }
    else if(age>=18){
        cout<<"congrats! you're an adult now"<<endl;
    }
    else{
        cout<<"you're still a kid"<<endl;
    }

    return 0;
}
```

Short hand if else

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

int main(){

int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
cout << result << endl;

return 0;
}
```

Switch Statement

Switch statement: It provides a way to select one of many code blocks to be executed based on the value of a variable or an expression.

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

int main(){

int d = 4;
switch (d) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;
}

return 0;
}
```

Switch case using input from user

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

int main(){

int d;

cout<<"Enter the number of your day (1-7) : ";
cin>>d;

switch (d) {
case 1:
   cout << "Monday";
   break;
case 2:
   cout << "Tuesday";
   break;
case 3:
   cout << "Wednesday";
   break;
case 4:
   cout << "Thursday";
   break;
case 5:
   cout << "Friday";
   break;
case 6:
   cout << "Saturday";
   break;
case 7:
   cout << "Sunday";
   break;
default :
   cout<< "Invalid number"<<endl;
}

return 0;
}
```

Looping Statements

- While loop: It executes a block of code as long as a specified condition is true.
- Do-while loop: It is similar to the while loop but ensures that the block of code is executed at least once, even if the condition is initially false.
- For loop: It allows you to execute a block of code repeatedly for a specific number of times.

<u>While loop</u>

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

int main(){

int i = 0;
while (i < 5) {
  cout << i << endl;
  i++;
}

return 0;
}
```

<u>Do while loop</u>

```
#include<iostream>
```

```
using namespace std;

int main(){

int i = 0;
do {
  cout << i << endl;
  i++;
}
while (i < 5);

return 0;
}
```

For loop

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

int main(){

int i = 0;

for (int i = 0; i < 5; i++) {
  cout << i << endl;
}

return 0;
}
```

Example 1

Write a C++ program to print table of 7 using for loop

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

int main(){
int num = 7;

    for(int i=1; i<=10; i++){
        cout << num << " x " << i << " = " << num*i << endl;
    }

    return 0;
}
```

Example 2

Write a C++ program to take a number input from user and print its table using for loop

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

int main(){
    int num;

    cout << "Enter any number (1-9) : " ;
    cin >> num;

    cout << "The table of " << num << " is " << endl;
    for(int i = 1; i <= 10; i++){
        cout << num << " x " << i << " = " << num * i << endl;
    }

    return 0;
}
```

Jump Statements

- Break statement: It terminates the execution of the innermost loop or switch statement and transfers control to the next statement.
- Continue statement: It skips the rest of the code in the current iteration of a loop and moves to the next iteration.
- Return statement: It terminates the execution of a function and returns a value (if any) to the caller.

Break in for loop

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

int main(){

int i = 0;

for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  cout << i << "\n";
}

return 0;
}

-----------------------------------------------------------------------------------------

// 0
// 1
// 2
// 3

// Here when the value of i became 4, the execution of the program stopped
```

Continue in for loop

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

int main(){

int i = 0;

for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  cout << i << "\n";
}

return 0;
}

-----------------------------------------------------------------------------------------

// 0
// 1
// 2
// 3
// 5
// 6
// 7
// 8
// 9

// Here when the value of i became 4 the program skips that line and it printed all the numbers except that value
```

Pointers

A pointer is a variable that stores the memory address of another variable. It allows you to indirectly access and manipulate the value stored in that memory location. Instead of working with the actual value directly, you use the pointer to point to the memory location where the value is stored.

// & -> Address of operator
// * -> Dereference operator

#include<iostream>
using namespace std;

```
#include<iostream>
int main(){

int a = 7;
int *b = &a;

cout<<"The address of a is : "<<&a<<endl;
cout<<"The address of a is : "<<b<<endl;

cout<<"The value at address b is : "<<*b<<endl;

return 0;
}
```

Pointer to Pointer

A pointer to a pointer is a variable that stores the address of another pointer. It adds an additional level of indirection. By using a pointer to a pointer, you can indirectly modify the value of a pointer variable.

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

int main(){

int a = 7;
int *b = &a;
int  **c = &b;
cout<<"The address of b is : "<<&b<<endl;
cout<<"The address of b is : "<<c<<endl;
cout<<"The value at address c is : "<<*c<<endl;

cout<<"The value at address value_at(value_at(c)) is : "<<**c<<endl;

return 0;
}
```

Arrays

An array is like a special container that can hold multiple items of the same data type, such as numbers or words. It's like having a row of boxes connected to each other. Each box in the row is called an element, and each element can store a specific piece of information.

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

int main(){

int marks[] = {70, 67, 49, 55, 60};

cout<<marks[0]<<endl;
cout<<marks[1]<<endl;
cout<<marks[2]<<endl;
cout<<marks[3]<<endl;
cout<<marks[4]<<endl;

return 0;
}
```

Changing value of an array

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

int main(){

int marks[] = {70, 67, 49, 55, 60};

cout<<marks[0]<<endl;
cout<<marks[1]<<endl;

marks[2]=45;

cout<<marks[2]<<endl;
cout<<marks[3]<<endl;
cout<<marks[4]<<endl;

return 0;
}
```

Printing array values with loop

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

int main(){

int marks[] = {70, 67, 49, 55, 60};

for(int i=0; i<5; i++){
   cout<<marks[i]<<endl;
}

return 0;
}
```

Pointer Arithmetic

Pointer arithmetic is a way to perform calculations using memory addresses in programming. It allows you to manipulate the addresses that pointers point to and navigate through arrays and data structures efficiently. Let's break it down in a simple way:

Pointers: A pointer is a variable that stores the memory address of another variable. Think of it as a way to "point" to the location where data is stored.

Memory addresses: In computer memory, data is stored at specific locations, each with a unique address. These addresses can be thought of as numbered slots in a huge storage unit.

Pointer Arithmetic: With pointer arithmetic, you can perform operations on pointers using basic arithmetic operators like addition (+) and subtraction (-). These operations allow you to move the pointer to different memory locations based on the size of the data it points to.

Incrementing and decrementing: When you increment a pointer, you are moving it to the next memory location based on the size of the data it points to. For example, if you have a pointer to an integer (4 bytes), incrementing the pointer will move it to the next integer in memory.

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

int main(){

int marks[] = {70, 67, 49, 55, 60};
int *p = marks;

cout<<"The value of marks[0] is : "<<*p<<endl;
cout<<"The value of marks[1] is : "<<*(p+1)<<endl;
cout<<"The value of marks[2] is : "<<*(p+2)<<endl;
cout<<"The value of marks[3] is : "<<*(p+3)<<endl;
cout<<"The value of marks[4] is : "<<*(p+4)<<endl;

return 0;
}
```

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

int main(){

int marks[] = {70, 67, 49, 55, 60};
int *p = marks;

cout<<*(++p)<<endl; // Here *(++p) means that the value was incremented and it pointing to the next value : 67, instead of the first value : 70
cout<<*p<<endl;    // Here the value will point at 67

return 0;
}
```

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

int main(){

int marks[] = {70, 67, 49, 55, 60};
int *p = marks;

cout<<*(p++)<<endl; // Here the *(p++) means that the current value will first print : 70, then it will point at another location
cout<<*p<<endl;    // Here the value was changed and it was pointing at the next value : 67

return 0;
}
```

Structure

The term "structure" refers to a user-defined data type that allows you to group data items together.

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

int main(){

   typedef struct employee{
      int eId;
      float salary;
      char favChar;
   }ep;             // Here by writing "ep" we are defining a short name for employee

   ep sudipto;       // Here instead of writing "struct employee sudipto" we can simply write "ep sudipto" in short
   ep muffin;        // Here instead of writing "struct employee muffin" we can simply write "ep muffin" in short

   sudipto.eId = 06;
   sudipto.salary = 4500000;
   sudipto.favChar = 'z';

   muffin.eId = 18;
   muffin.salary = 4500000;
   muffin.favChar = 's';

   cout<<"The employee id of muffin is : "<<muffin.eId<<endl;
   cout<<"The employee id of sudipto is : "<<sudipto.eId<<endl;

return 0;
}
```

Union

A union is a special data type that allows storing different types of data in the same memory location. Unlike a structure, which allocates separate memory for each member, a union shares the memory among all its members. The members of a union occupy the same memory space, so the size of the union is determined by the largest member within it.

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

int main(){

   union share{
      int rice;
      float car;
      char pounds;
   };

   union share m1;

   m1.rice = 34;

   cout<<"The value of m1.rice is :"<<m1.rice<<endl;


return 0;
}

----------------------------------------------------------------------------------------

/*
Here we are using only one type so union program will execute properly by instead if
we have used car and pounds as well, then it will provide us garbage value of other two.
Because union acceps one type and one value at a time.
*/
```

Enums

Enums in C++ are a way to define a list of named values. They are like a group of options or choices that you can use in your code.

Imagine you have a program that deals with colors. Instead of using random numbers to represent colors, you can create an enum called Color and define different colors as options.

In this enum, we have three colors: red, green & blue.

Now, instead of using numbers like 1, 2, or 3 to represent these colors, you can use the enum values. For instance, you can declare a variable currentColor of type Color and assign it a color.

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

int main(){

enum color{red, green, blue};

color then = red;
cout<<then<<endl;

color now = green;
cout<<now<<endl;

color later = blue;
cout<<later<<endl;

return 0;
}

----------------------------------------------------------------------------------------

// Here the outputs will be 0, 1, 2 respectively according to the color order.
```

Functions

Functions are like mini-programs within a program. They are created to perform specific tasks. You can think of them as a set of instructions that you can reuse whenever you need to do the same task again and again.

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

int sum(int a, int b){   // Here int a, int b are formal parameters
   int c = a + b;

   return c;
}

int main(){
   int num1, num2;
   cout<<"Enter the value of num1 : ";
   cin>>num1;

   cout<<"Enter the value of num2 : ";
   cin>>num2;

   cout<<"The sum of num1 & num2 is : "<<sum(num1, num2)<<endl; // Here the value of num1, num2 are actual parameters

return 0;
}
```

Function Prototyping

Function prototyping is a way to let the computer and other parts of your program know about a function before it's fully defined. It helps with organization and makes sure everything works together smoothly.

Formal Parameters & Actual Parameters

Formal parameters, also known as formal arguments or simply parameters, are placeholders or variables declared in the function definition. They represent the values that a function expects to receive when it is called.

Actual parameters, also known as actual arguments or simply arguments, are the values provided to a function when it is called. These values are passed to the function to be used in place of the formal parameters.

Swapping values using pointer

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

void swapPointer(int *a, int *b){
int temp = *a;
   *a = *b;
   *b = temp;

}

int main(){

int x = 4, y = 5;

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
swapPointer(&x, &y);
cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

return 0;
}
```

Inline Functions

Inline functions are a feature in programming languages that allow the compiler to insert the code of a function directly at the call site, rather than executing a separate function call.

When a function is declared as inline, the compiler replaces the function call with the actual body of the function during the compilation process. This eliminates the overhead of the function call.

Inline functions are commonly used for small, frequently called functions or for performance-critical code sections where the elimination of function call overhead can have a noticeable impact on execution time.

Static Variable

Imagine you have a class called "Car" that represents different cars. Each car has its own properties like color, model, and speed. However, there is also a piece of information that is the same for all cars, such as the total number of cars created.

In this case, the total number of cars is a static variable. It's like a shared counter that keeps track of how many cars have been created, regardless of the individual properties of each car.

Whenever a new car is created, the static variable is incremented by one. All instances of the "Car" class can access and update this static variable, and they all see the same value. So, if you create three car objects, the static variable will show a value of 3.

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

int product(int a, int b){
   static int c = 0;      // This will execute only once
   c = c + 1;             // Next time when the function will run it will retain the value of c
   return a*b+c;
}
int main(){
   int x, y;

   cout<<"Enter the value of x and y :"<<endl;
   cin>>x>>y;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;
   cout<<"The product of x and y is : "<<product(x,y)<<endl;

return 0;

}
```

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

float moneyReceived(int currentMoney, float interestRates = 1.04){
    return currentMoney * interestRates;

}
int main(){

   int money = 100000;
   cout<<"If you have "<<money<<" Rs in your bank account then you will receive "<<moneyReceived(money)<<" Rs after 1 year"<<endl;
   cout<<"If you are a V.I.P and you have "<<money<<" Rs in your bank account then you will receive "<<moneyReceived(money, 1.1)<<" Rs after 1 year";

return 0;
}
```

Constant Arguments

Constant arguments, also known as read-only arguments or immutable arguments, are function parameters that cannot be modified within the function. This is particularly useful when you want to prevent accidental modifications to certain parameters.

Recursion

Recursion refers to a process where a function calls itself repeatedly to solve a problem. It's like a never-ending loop that gradually breaks down a complex problem into simpler subproblems until it reaches a base case that can be easily solved.

Think of it as a puzzle where you have a big picture made up of smaller puzzle pieces. Instead of trying to solve the entire puzzle at once, recursion allows you to focus on solving each smaller piece and then combining them to solve the bigger puzzle.

EXAMPLE 1

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

int factorial(int n){
   if(n<=1){
      return 1;
   }
   return n * factorial(n-1);
}

// factorial(4) = 4 * factorial(3)
// factorial(4) = 4 * 3 * factorial(2)
// factorial(4) = 4 * 3 * 2 * factorial(1)
// factorial(4) = 4 * 3 * 2 * 1
// factorial(4) = 24

int main(){

   int a;
   cout<<"Enter a number : ";
   cin>>a;

   cout<<"The factorial of a is : "<<factorial(a)<<endl;

return 0;
}
```

EXAMPLE 2

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

int fibo(int n){
   if(n<2){
      return 1;
   }
   return fibo(n-2) + fibo(n-1);
}

int main(){

   int a;
   cout<<"Enter a number : ";
   cin>>a;

   cout<<"The term in fibonacci sequence at position "<<a<<" is " <<fibo(a)<<endl;

return 0;
}
```

Function Overloading

Function overloading is a feature in programming languages that allows you to create multiple functions with the same name but with different parameters. It's like having several versions of a function that perform similar tasks but can handle different types of inputs.

When you call an overloaded function, the programming language figures out which version of the function to use based on the arguments you provide. It matches the arguments you pass with the parameters of the different versions of the function and selects the one that best matches.

By using function overloading, you can write more flexible and reusable code. It allows you to use the same function name for related operations, making your code easier to understand and maintain.

EXAMPLE 1

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

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

int sum(int a, int b, int c){
   return a + b + c;
}

int main(){
   cout<<" The sum of 5 and 2 is : "<<sum(5,2)<<endl;
   cout<<" The sum of 5, 3 and 2 is : "<<sum(5,3,2)<<endl;

return 0;
}
```

EXAMPLE 2

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

// Calculate the volume of a cylinder
int volume(double r, int h){
   return (3.14 * r * r * r);
}

// Calculate the volume of a cube
int volume(int a){
   return (a * a * a);
}

// Calculate the volume of a rectangle box(cuboid)
int volume(int l, int b, int h){
   return (l * b * h);
}

int main(){
   cout<<"The volume of a cylinder of radius 3 and height 6 is : "<<volume(3,6)<<endl;
   cout<<"The volume of a cube of side 3  is : "<<volume(3)<<endl;
   cout<<"The volume of a cuboid 3, 7 and 6 is : "<<volume(3,7,6)<<endl;

return 0;
}
```

Object Oriented Programming (OOP)

- Works on concept of classes and objects.
- A class is a template to create objects.
- Treats data as a critical element.
- **Classes** - Basic template for creating objects.
- **Objects** - Basic run time entities.
- **Data Abstraction & Encapsulation** - Wrapping data and functions into single unit.
- **Inheritance** - Properties of one class can be inherited into others.
- **Polymorphism** - Ability to take more than one forms.
- **Dynamic Binding** - Code which will execute is not known until the program runs.
- **Message Passing** - Object.message(information) call format.

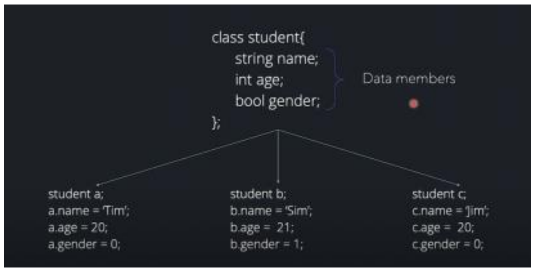
1. **Classes and Objects:** A class is a blueprint or template that defines the structure and behavior of objects. An object is an instance of a class.
2. **Encapsulation:** It is the mechanism that binds data and functions (methods) together in a class, preventing direct access to the data from outside the class. Encapsulation helps achieve data hiding and provides abstraction.
3. **Abstraction:** It refers to the process of simplifying complex systems by breaking them down into smaller, more manageable components. In OOP, classes provide abstraction by exposing only relevant information and hiding internal details.
4. **Inheritance:** It allows a class to inherit properties and behaviors from another class, known as the base or parent class. Inheritance promotes code reuse and the creation of hierarchical relationships between classes.
5. **Polymorphism:** It refers to the ability of objects of different classes to respond differently to the same function or method call. Polymorphism allows methods to be overridden in derived classes, enabling dynamic binding at runtime.
6. **Polymorphic Behavior through Virtual Functions:** In C++, polymorphic behavior is achieved through virtual functions. A virtual function is a function declared in a base class that can be overridden by derived classes. This enables the correct function to be called based on the object's actual type at runtime.
7. **Data Abstraction and Encapsulation:** OOPs in C++ emphasizes data abstraction, which involves exposing only the necessary interfaces to manipulate data while hiding the implementation details. Encapsulation is the mechanism used to achieve this abstraction by bundling data and methods together in a class.
8. **Message Passing:** Objects communicate with each other by sending messages. A message is a request to execute a method on an object. The object then responds by executing the appropriate method.

Classes

Classes are a fundamental building block of object-oriented programming (OOP). A class is a user-defined data type that encapsulates data (attributes) and functions (methods) into a single unit. It provides a blueprint or template for creating objects, which are instances of the class.

Public: Members declared as public are accessible from anywhere in the program. They can be accessed by objects of the class, as well as from outside the class.

Private: Members declared as private are only accessible within the same class. They cannot be accessed directly by objects of the class or from outside the class. They can only be accessed through public member functions, which provide controlled access to the private members.


```
class student{
    string name;
    int age;
    bool gender;
};
```

```
student a;           student b;           student c;
a.name = 'Tim';      b.name = 'Sim';      c.name = 'Jim';
a.age = 20;          b.age =  21;         c.age =  20;
a.gender = 0;        b.gender = 1;        c.gender = 0;
```

EXAMPLE 1

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

class Car {
  // Data members or attributes
  string brand;
  string model;
  int year;

public:
  // Member functions or methods
  void setBrand(string b) {
    brand = b;
  }

  void setModel(string m) {
    model = m;
  }

  void setYear(int y) {
    year = y;
  }

  void displayInfo() {
    cout << "Brand: " << brand << endl;
    cout << "Model: " << model << endl;
    cout << "Year: " << year << endl;
  }
};

int main(){

  Car myCar;  // Creating an object of the Car class

  myCar.setBrand("Toyota");
  myCar.setModel("Camry");
  myCar.setYear(2022);

  myCar.displayInfo();

return 0;
}
```

EXAMPLE 2

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

class employee{
    private:
    int a, b, c;

    public:
    int d, e;

    void setdata(int a1, int b1, int c1);
    void getdata(){
        cout<<"The value of a is : "<<a<<endl;
        cout<<"The value of b is : "<<b<<endl;
        cout<<"The value of c is : "<<c<<endl;
        cout<<"The value of d is : "<<d<<endl;
        cout<<"The value of e is : "<<e<<endl;
    }
};

void employee :: setdata(int a1, int b1, int c1){
     a = a1;
     b = b1;
     c = c1;
}

int main(){
```

```
employee sudipto;

sudipto.d = 7;
sudipto.e = 3;
sudipto.setdata(5,3,2);
sudipto.getdata();

return 0;
}
```

Nesting of Member Function

Nesting of member functions refers to the concept of defining one member function inside another member function within a class.

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

class binary{
    string s;
    public:
        void read(void);
        void chk_bin(void);

};

void binary :: read(void){
    cout<<"Enter a binary number : ";
    cin>>s;
}

void binary :: chk_bin(void){
    for(int i=0; i< s.length(); i++)
    {
        if(s.at(i)!= '0' && s.at(i)!= '1'){
            cout<<"Incorrect Binary Format "<<endl;
            exit(0);
        }
    }
}

int main(){

    binary b;
    b.read();
    b.chk_bin();

return 0;
}
```

Object Memory Allocation

Memory allocation for objects can be done using two main approaches: stack allocation and heap allocation.

<u>**ARRAYS IN CLASSES**</u>

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

class shop{
    int itemId[100];
    int itemPrice[100];
    int counter;

    public:
        void initCounter(void){counter = 0;}
        void setPrice(void);
        void displayPrice(void);
};

void shop :: setPrice(void){
    cout<<"Enter the ID of your item no : "<< counter + 1 <<endl;
    cin>>itemId[counter];
    cout<<"Enter the price of your item : "<<endl;
    cin>>itemPrice[counter];
    counter++;
}

void shop :: displayPrice(void){
    for(int i=0; i<counter; i++){
        cout<<"The price of item with ID "<<itemId[i]<<" is "<<itemPrice[i]<<endl<<endl;
    }
}
int main(){
    shop dukaan;

    dukaan.initCounter();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.displayPrice();

return 0;
}
```

Static Data Members

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

class employee{
    static int count;
    int id;

    public:
     void setData(void){
        cout<<"Enter the id : ";
        cin>>id;
        count++;
     }

     void getData(void){
        cout<<"The id of this employee is "<<id<<" and this is employee number "<<count<<endl;
     }
};

int employee :: count; // Default value is 0

int main(){
    employee sudipto, muffin;

    sudipto.setData();
    sudipto.getData();

    muffin.setData();
    muffin.getData();

return 0;
}
```

Array of Objects

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

class employee{
    int id;
    int salary;

  public:

     void setid(void){
        cout<<"Enter employee ID : ";
        cin>>id;
     }

    void getid(void){
        cout<<"The ID os this employee is : "<<id<<endl;
    }
};

int main(){

    employee facebook[4];

    facebook[0].setid();
    facebook[0].getid();

    for(int i=0; i<4; i++){
        facebook[i].setid();
        facebook[i].getid();
    }

return 0;
}
```