Unit 3 - Array

CONTENT-Array: Array Illustration, Multi, Dimensional arrays, Strings, Array of Strings, Function prototype, function return data type, parameter passing, Default argument, Inline function, Function Overloading, Array Function, Operator Overloading,

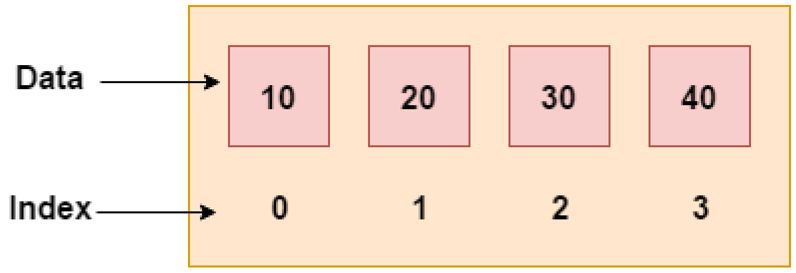
Outcome of these topics:
Better understanding of how arrays work and how to manipulate them.
Ability to work with strings and arrays of strings efficiently.
Understanding of function declaration, parameter passing, and return types.
Improved code organization and readability through function prototypes and default arguments.
Efficiency improvements through the use of inline functions.
Enhanced code expressiveness and flexibility via function overloading and operator overloading.
Ability to write more complex and efficient algorithms, especially those dealing with arrays and user-defined data types.

C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.

A collection of related data items stored in adjacent memory places is referred to as an array in the C/C++ programming language or any other programming language for that matter. Elements of an array can be accessed arbitrarily using its indices. They can be used to store a collection of any type of primitive data type, including int, float, double, char, etc. An array in C/C++ can also store derived data types like structures, pointers, and other data types, which is an addition. The array representation in a picture is provided below.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
        //traversing array
```

```
    for (int i = 0; i < 5; i++)
    {
        cout<<arr[i]<<"\n";
    }
}
```

Output:

```
10
0
20
0
30
```

What is two-dimensional array?

Each element in this kind of array is described by two indexes, the first of which denotes a row and the second of which denotes a column.

As you can see, the components are arranged in a two-dimensional array using rows and columns; there are I number of rows and j number of columns.

What is a multi-dimensional array?

A two-dimensional array is the most basic type of multidimensional array; it also qualifies as a multidimensional array. There are no restrictions on the array's dimensions.

How to insert it in array?

```
int mark[5] = {19, 10, 8, 17, 9}
// change 4th element to 9
mark[3] = 9;
// take input from the user
// store the value at third position
cin >> mark[2];
// take input from the user
// insert at ith position
cin >> mark[i-1];

```

```
// print first element of the array
cout << mark[0];
// print ith element of the array
cout >> mark[i-1];
```

How to display the sum and average of array elements?

```
#include <iostream>
using namespace std;
int main() {
// initialize an array without specifying the size
double numbers[] = {7, 5, 6, 12, 35, 27};
double sum = 0;
double count = 0;
double average;
cout << "The numbers are: ";
 //  print array elements
 // use of range-based for loop
 for (const double &n : numbers) {
  cout << n << "  ";
//  calculate the sum
sum += n;
// count the no. of array elements
++count;
  }
// print the sum
cout << "\nTheir Sum = " << sum << endl;
// find the average
average = sum / count;
cout << "Their Average = " << average << endl;

  return 0;
}
```

Output:

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

```
The numbers are: 7  5  6  12  35  27
Their Sum = 92
Their Average = 15.3333
```

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```cpp
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
        int arr1[5] = { 10, 20, 30, 40, 50 };
        int arr2[5] = { 5, 15, 25, 35, 45 };
        printArray(arr1); //passing array to function
        printArray(arr2);
}
void printArray(int arr[5])
{
    cout << "Printing array elements:"<< endl;
    for (int i = 0; i < 5; i++)
    {
                cout<<arr[i]<<"\n";
    }
}
```

Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C++. It can be two dimensional or three dimensional. The data is stored in tabular form (row * column) which is also known as matrix.

C++ Multidimensional Array Example

Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
  int test[3][3];  //declaration of 2D array
    test[0][0]=5;  //initialization
    test[0][1]=10;
    test[1][1]=15;
    test[1][2]=20;
    test[2][0]=30;
    test[2][2]=10;
    //traversal
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<<" ";
        }
        cout<<"\n"; //new line at each row
    }
    return 0;
}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Strings

In C++, string is an object of **std::string** class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

C++ String Example

Let's see the simple example of C++ string.

```
#include <iostream>
using namespace std;
int main( ) {
    string s1 = "Hello";
        char ch[] = { 'C', '+', '+'};
        string s2 = string(ch);
        cout<<s1<<endl;
        cout<<s2<<endl;
}
```

Output:

```
Hello
C++
```

C++ String Compare Example

Let's see the simple example of string comparison using strcmp() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
```

```
{
  char key[] = "mango";
  char buffer[50];
  do {
     cout<<"What is my favourite fruit? ";
     cin>>buffer;
  } while (strcmp (key,buffer) != 0);
 cout<<"Answer is correct!!"<<endl;
  return 0;
}
```

Output:

```
What is my favourite fruit? apple
What is my favourite fruit? banana
What is my favourite fruit? mango
Answer is correct!!
```

C++ String Concat Example

Let's see the simple example of string concatenation using strcat() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char key[25], buffer[25];
    cout << "Enter the key string: ";
    cin.getline(key, 25);
    cout << "Enter the buffer string: ";
     cin.getline(buffer, 25);
    strcat(key, buffer);
    cout << "Key = " << key << endl;
    cout << "Buffer = " << buffer<<endl;
    return 0;
}
```

Output:

```
Enter the key string: Welcome to
Enter the buffer string:  C++ Programming.
Key = Welcome to C++ Programming.
Buffer =  C++ Programming.
```

C++ String Copy Example

Let's see the simple example of copy the string using strcpy() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char key[25], buffer[25];
    cout << "Enter the key string: ";
    cin.getline(key, 25);
    strcpy(buffer, key);
    cout << "Key = "<< key << endl;
    cout << "Buffer = "<< buffer<<endl;
    return 0;
}
```

Output:

```
Enter the key string: C++ Tutorial
Key = C++ Tutorial
Buffer = C++ Tutorial
```

C++ String Length Example

Let's see the simple example of finding the string length using strlen() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
```

```
{
    char ary[] = "Welcome to C++ Programming";
    cout << "Length of String = " << strlen(ary)<<endl;
    return 0;
}
```

Output:

```
Length of String = 26
```

Function Prototype in C++

A function prototype is a declaration of the function that informs the program about the number and kind of parameters, as well as the type of value the function will return. One incredibly helpful aspect of C++ functions is function prototyping. A function prototype provides information, such as the number and type of parameters and the type of return values, to explain the function interface to the compiler.

The prototype declaration resembles a function definition exactly, with the exception that it lacks a body, or its code. At this point, you were aware of the distinction between a statement and a definition.

A definition is a declaration that also informs the program what the function is doing and how it is doing, as opposed to a declaration, which simply introduces a (function) name to the program. Therefore, the examples above are function definitions, and the examples that follow are declarations, or perhaps a better term would be function prototypes:

```
int valAbs ( int x ) ;
int greatcd ( int a1 , int a2 ) ;
```

Consequently, the components of a function prototype are as follows:

- return type
- name of the function
- argument list

Let's look at the prototype for the following function:

```
int add ( int a1 , int a2 ) ;
```

Here,

- return type - int
- name of the function - add
- argument list - (int a1, int a2)

Since a semicolon follows every function prototype, there will eventually be; just like in the previous function prototype.

Usage of Void

As you are aware, the void data type is used as the return type for functions that do not return a value and describes an empty collection of values. Consequently, the declaration of a function that doesn't return a value is as follows:

```
1. void func_name ( parameter x ) ;
```

One ensures that a function cannot be utilized in an assignment statement by defining the return type of the function to be void.

NOTE: Declare the result type as void if a function does not return a value.

A function can be defined as follows if it has no parameters and has an empty argument list:

```
1. return_type func_name ( void ) ;
```

NOTE: You should declare void in a function's prototype if it doesn't accept any arguments.

As was already noted, if a function's type specifier is omitted, it is presumed that it will return int values. The type specifier must be provided for functions yielding non-integer values.

A function prototype may exist either before or after the definition of invoking the function (these prototypes are referred to as global prototypes) (such prototypes are known as local prototypes). The C++ Scope Rules tutorial has a separate tutorial that describes both the global and local prototypes.

Let us look at an example of function prototype in c++:

```
1. # include < iostream >
2. using namespace std ;
```

```
// function prototype
void divide ( int , int ) ;
int main ( ) {
    // calling the function before declaration.
    divide ( 10 , 2 ) ;
    return 0 ;
}
// defining function
void divide ( int a , int b ) {
    cout < < ( a / b ) ;
}
```

OUTPUT:

```
5
???.
Process executed in 0.11 seconds
Press any key continue.
```

Explanation

In the above example the prototype is:

```
void divide ( int , int ) ;
```

The function prototype feature in C++ allows us to call the function before it has been declared since in the example above, the compiler is given information about the function name and its parameters.

Another example to better understanding:

```
# include < iostream >
using namespace std ;
// Function prototype
void best_site ( ) ; //using function with void return type
int main ( ) {
    best_site ( ) ;
    return 0 ;
}
Void best_site ( ) {
```

```
10.   cout < < " Welcome to JavaTpoint " ;
11. }
```

OUTPUT:

```
Welcome to JavaTpoint
?????????????.
Process executed 0.22 seconds
Press any key to continue.
```

Explanation

The function in the code above has a void return type, so it returns nothing. We have performed function prototyping, called the function, and then declared it, and as a result, we are receiving an output free of errors.

C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

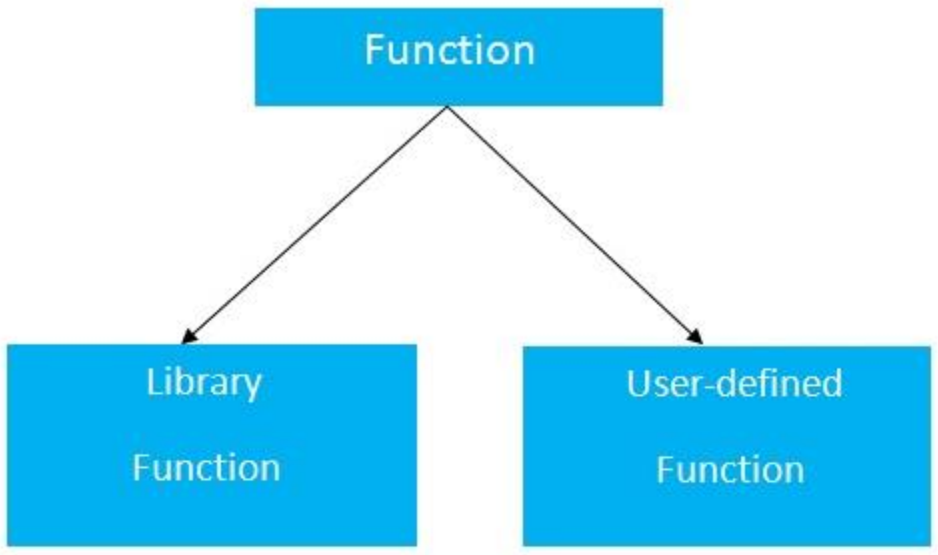
But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Declaration of a function

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

C++ Function Example

Let's see the simple example of C++ function.

```
#include <iostream>
using namespace std;
void func() {
   static int i=0; //static variable
   int j=0; //local variable
   i++;
   j++;
   cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
}
```

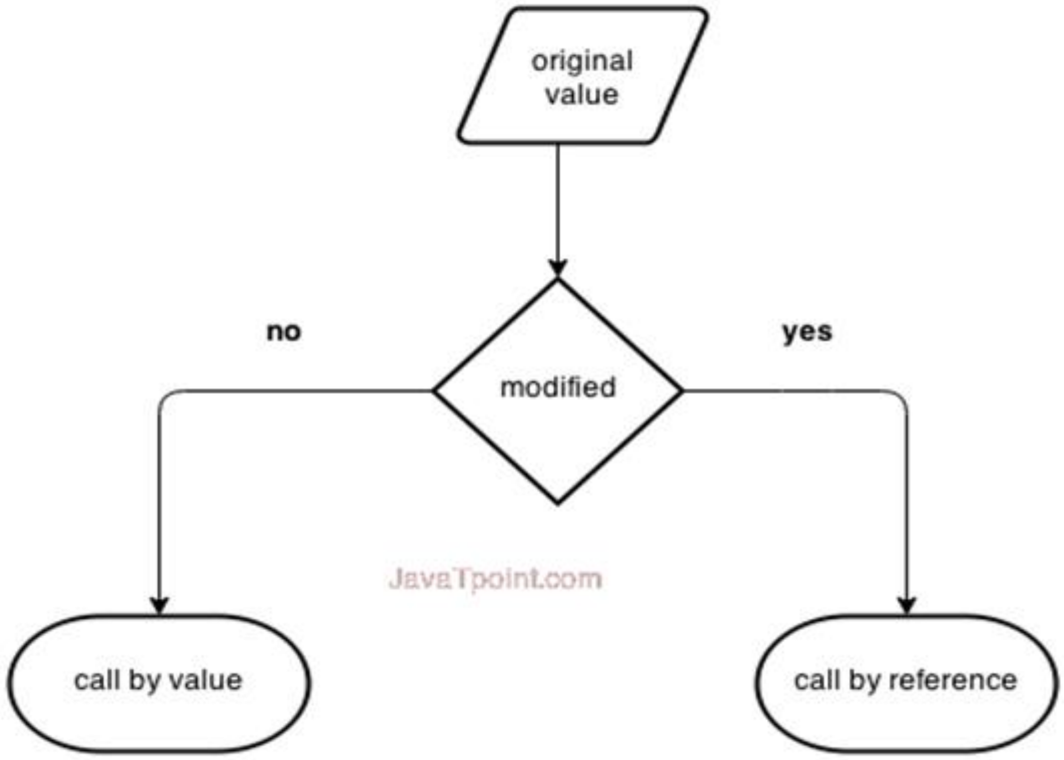
Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

Next →← Prev

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
}
void change(int data)
{
data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
```

```
int x=500, y=100;
swap(&x, &y);  // passing value to function
cout<<"Value of x is: "<<x<<endl;
cout<<"Value of y is: "<<y<<endl;
return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ Function Overloading

Two or more functions may share the same name but not the same list of arguments when functional overloading occurs. It is a key characteristic of C++. Compile-time polymorphism and function overloading are similar concepts. A close examination reveals that the name stays the same, although the list of arguments, data type, and order all change. Take a look at a C++ function overloading example.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                                        //    class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
    return 0;
}
```

Output:

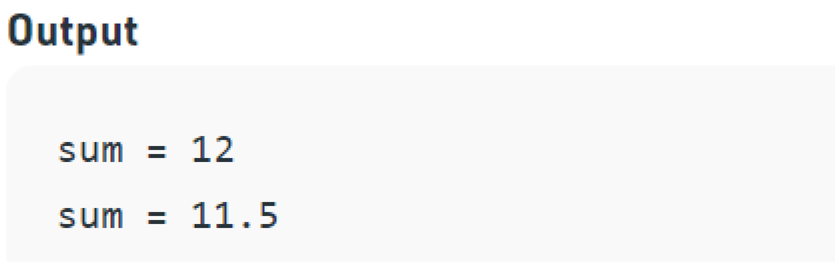
```
30
55
```

Example

```
#include <iostream>
using namespace std;
void add(int a, int b)
{
cout << "sum = " << (a + b);
}
void add(double a, double b)
```

```
{
    cout << endl << "sum = " << (a + b);
}
int main()
{
    add(10, 2);
    add(5.3, 6.2);
          return 0;
}
```

Output:



```
Output

sum = 12
sum = 11.5
```

In the aforementioned example, we can see that two functions are defined as a piece of code. The names of the functions are the same-"addPodium"-but the return type, the list of input arguments, and the data types for those arguments have been altered.

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
   float r2 = mul(0.2,3);
    std::cout << "r1 is : " <<r1<< std::endl;
   std::cout <<"r2 is : "  <<r2<< std::endl;
    return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

Function with pass by reference

Let's see a simple example.

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
18. }
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the us

Why is C++ Using Function Overloading?

Over traditional structured programming languages, the OOPS ideas offer a number of benefits. Overloading functions is regarded as compile-time polymorphism. With the help of this OOPS idea, a programmer can create a function with the same name but a distinct execution pattern, enabling the code to be more understandable and reusable.

What are the rules of function overloading in C++?

When overloading a function in C++, there are some guidelines that must be observed. Let's take a closer look at a few of them:

1) The parameters for each function must be in a distinct order.

2) The functions have to be named the same.

3) The parameters for the functions must be distinct.

4) The parameters for the functions must be of various sorts.

What are the types of function overloading in C++?

In C++, there are two types of function overloading. Those are

1) Overloading at compile time occurs when alternative signatures are used to overload the functions. The function's return type, number, and type of parameters are all regarded as the function's signature.

2) Overloading that occurs during runtime refers to the overloading of the functions. Time overloading occurs when a different number of parameters are added to the function during execution.

What are the advantages of function overloading in C++?

Here are a few benefits of function overloading in C++.

1) The programmer can create functions with distinct purposes but the same name by using function overloading.

2) It speeds up the program's execution.

3) The code is clearer and simpler to comprehend.

4) It reduces memory utilization and makes programs reusable.

What are the disadvantages of function overloading in C++?

The following are some drawbacks of C++ function overloading.

1) The primary drawback of function overloading is that it prevents the overloading of functions with various return types.

2) The identical parameters cannot be overloaded in the case of a static function.

What are the differences between function overloading and operator overloading?

Function overloading	Operator overloading
It is possible to use various parameters in an overload of a function with the same name.	One can overload operators such as +, -, /
A function might have more than one execution with various parameters when it is overloaded.	Operation reliance on operands occurs when an operator is overloaded.
The user can call using a variety of methods.	In addition to the predetermined meaning, it enables the user to have a more expansive meaning.

What are the differences between function overloading and function overriding?

Function overloading	**Function overriding**
It allows the programmer to have many functions with the same name but distinct arguments.	Method overriding occurs when two methods with the same name and parameters are present in different classes, one in the parent and one in the child.
They have the same scope.	They have different scopes.
Even without inheritance, function overloading is possible.	It only happens when there is inheritance.

What is operator overloading in C++?

Operators for user-defined classes can be made to function in C++. This indicates that the operator overloading feature of C++ allows it to give the operators a special meaning specific to a data type. As an illustration, we can overload the operator "+" in a class like "String" such that we can append two strings with simply the letter "+." The classes Big Integer, Complex Numbers, and Fractional Numbers are a few others where mathematical operators may be overloaded.

Compile-time polymorphisms include operator overloading. It is the concept of providing an existing C++ operator with additional meaning while maintaining its original meaning.

Example:

```
int x;
float y, sum;
sum=x+y;
```

The variables "x" and "y" in this example are of the built-in data types "int" and "float." The contents of "x" and "y" can thus be added simply using the addition operator "+." Because only variables with built-in data types are predefined to be added by the addition operator "+," this is the case.

Now, consider another example

```
class B
{
};
int main()
{
B   b1,b2,b3;
 b3= b1 + b2;
 return 0;
}
```

Three variables of type "class B" are used in this example: "b1," "b2," and "b3." Using the "+" operator, we are attempting to combine two objects, "b1" and "b2," that are of a user-defined type, or type "class B." This is prohibited because the addition operator "+" can only operate on built-in data types by default. However, since "class B" is a user-defined type, in this case, the compiler produces an error. In this situation, the idea of "Operator overloading" is relevant.

The "+" operator must now be redefined so that it produces two class objects if the user desires it to add two class objects. Utilizing the idea of "operator overloading," this is accomplished. Thus, the primary principle of "Operator Overloading" is to employ C++ operators with class variables or class objects. Operators' original meaning is not really altered by redefining them; rather, they have been given new meanings in addition to their previous ones.

```
#include <iostream>
using namespace std;
class Count {
private:
int value;
public:
Count() : value(5) {}
void operator ++ () {
++value;
}
void display() {
cout << "Count: " << value << endl;
}
};
int main() {
Count count1;
++count1;
count1.display();
return 0;
}
```

Output:

```
Count: 6
```