Part: 11

Break and Continue Statements in C++

In this series of our C++ tutorials, we will visualize Break and continue statements in C++ language in this lecture. In our last lesson, we discussed for loop, while loop and do-while loop structures in C++.

In this C++ tutorial, the topics which we are going to cover today are given below:

- **Break Statements in C++**
- **Continue Statements in C++**

Break Statements

We had already discussed a little bit about break statements in switch statements. Today we will see the working of break statements in loops. Break statements in loops are used to terminate the loop. An example program for Break's statement is shown in figure 1.

```
4  int main(){
5      for (int i = 0; i < 40; i++)
6      {
7          /* code */
8          cout<<i<<endl;
9          if(i==2){
10             break;
11         }
12     }
13
```

Figure 1: Break Statement Program

As shown in figure 1, this is how the break statement program will be executed:

- Initialize integer variable **“i”** with value **“0”**
- Check the condition if the value of the variable **"i”** is smaller than **“40”**
- If the condition is true go into the loop body
- Execute **“cout”** function
- Check the condition if the value of the variable **"i”** is equal to **“2”**, if it is equal terminate the loop and get out of loop body
- Update the value of **“i”** by one
- Keep repeating these steps until the loop condition gets false, or the **“if”** condition inside the loop body gets true.

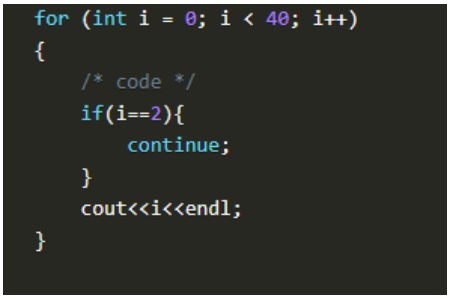
The output of the following program is shown in figure 2.

```
PS D:\Business\cod
f ($?) { .\tut11 }
0
1
2
```

Figure 2: Break Statement Program output

Continue Statements in C++

Continue statements are somewhat similar to break statements. The main difference is that the break statement entirely terminates the loop, but the continue statement only terminates the current iteration. An example program for continue statements is shown in figure 3.



```
for (int i = 0; i < 40; i++)
{
    /* code */
    if(i==2){
        continue;
    }
    cout<<i<<endl;
}
```

Figure 3: Continue Statement Program

As shown in figure 3, this is how the continue statement program will be executed:

- Initialize integer variable **"i"** with value **"0"**
- Check the condition if the value of the variable **"i"** is smaller than **"40"**
- If the condition is true go into the loop body
- Check the condition if the value of the variable **"i"** is equal to **"2"**, if it is equal terminate the loop for the current iteration and go to the next iteration
- Execute **"cout"** function
- Update the value of **"i"** by one
- Keep repeating these steps until the loop condition gets false.

Code as described/written in the video

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

int main(){
    // for (int i = 0; i < 40; i++)
    // {
    //     /* code */
    //     if(i==2){
    //         break;
    //     }
    //     cout<<i<<endl;
    // }
    for (int i = 0; i < 40; i++)
    {
        /* code */
        if(i==2){
            continue;
        }
        cout<<i<<endl;
    }


    return 0;
}
```

Part: 12

Pointers in C++

In this series of our C++ tutorials, we will visualize pointers in the C++ language in this lecture. In our last lesson, we discussed break statements and continue statements in C++.

Pointers in C++

A pointer is a data type which holds the address of other data type. The **“&”** operator is called **“address off"** operator, and the **"*”** operator is called **“value at”** dereference operator. An example program for pointers is shown in figure 1.

```
int a=3;
int* b = &a;
cout<<b;
```

Figure 1: Pointer Program

As shown in figure 1, at 1st line an integer variable **“a”** is initialized with the value **“3"**. At the 2nd line, the address of integer variable **"a”** is assigned to the integer pointer variable **“b"**. At the 3rd line, the address of the integer pointer variable **"b”** is printed. The output of the following program is shown in figure 2.

```
PS D:\Business\code pl
0x61ff08
PS D:\Business\code pl
```

Figure 2: Pointer Program Output

As shown in figure 2, the address of the integer pointer variable **"b”** is printed. The main thing to note here is that the address printed by the variable **“b"** is the address of integer variable **"a”** because we had assigned the address of variable **“a”** to the integer pointer variable **“b"**. To clarify, we will print both variable "a" and variable "b" addresses, which are shown in figure 3.

```
int a=3;
int* b = &a;
cout<<"The address of a is "<<&a<<endl;
cout<<"The address of a is "<<b<<endl;
```

Figure 3: Pointer Program Example 2

As shown in figure 3, now we printed both variable “a” and variable “b” addresses. The output for the following program is shown in figure 4.

```
PS D:\Business\code playground
The address of a is 0x61ff08
The address of a is 0x61ff08
```

Figure 4: Pointer Program Example 2 Output

As shown in figure 4, both variables **"a”** and **“b”** have the same addresses, but in actual, this is the address of the variable **"a”**, the variable **“b"** is just pointing to the address of the variable **"a”**.

To see the value of variable **"a"** using a pointer variable, we can use the **"*"** dereference operator. An example of the dereference operator program is shown in figure 5.

```
// * ---> (value at) Dereference operator
cout<<"The value at address b is "<<*b<<endl;
```

Figure 5: Dereference Operator example

As shown in figure 5, the value at address **"b"** is printed. The main thing to note here is that the value printed by the pointer variable **"b"** will be the value of variable **"a"** because the pointer variable **"b"** is pointing to the address of the variable **"a"**. The output for the following program is shown in figure 6.

```
// * ---> (value at) Dereference operator
cout<<"The value at address b is "<<*b<<endl;
```

Figure 6: Dereference Operator Example

Pointer to Pointer

Pointer to Pointer is a simple concept, in which we store the address of one Pointer to another pointer. An example program for Pointer to Pointer is shown in figure 7.

```
// Pointer to pointer
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;
```

Figure 7: Pointer to Pointer Example Program

As shown in figure 7, at the 1st line, the address of the pointer variable **"b"** is assigned to the pointer variable **"c"**. At 2nd line, the address of the pointer variable **"b"** is printed. At the 3rd line, the address of the pointer variable **"c"** is printed. At line 4th, the value at the pointer variable **"c"** is printed. At line 5th, the pointer variable **"c"** will be dereferenced two times, and it will print the value at pointer variable **"b"**. The output of the following program is shown in figure 2. The output for the following program is shown in figure 8.

```
The address of b is 0x61ff04
The address of b is 0x61ff04
The value at address c is 0x61ff08
The value at address value_at(value_at(c)) is 3
```

Figure 8: Pointer to Pointer Example Program Output

Code as described/written in the video

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

int main(){
    // What is a pointer? ----> Data type which holds the address of other data
types
    int a=3;
    int* b;
    b = &a;

    // & ---> (Address of) Operator
    cout<<"The address of a is "<<&a<<endl;
    cout<<"The address of a is "<<b<<endl;

    // * ---> (value at) Dereference operator
    cout<<"The value at address b is "<<*b<<endl;

    // Pointer to pointer
    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;
}
```

Part: 13

Arrays & Pointers Arithmetic in C++

In this tutorial, we will discuss arrays and pointer arithmetic in C++

What are Arrays in C++

- An array is a collection of items which are of the similar type stored in contiguous memory locations.
- Sometimes, a simple variable is not enough to hold all the data.
- For example, let's say we want to store the marks of 2500 students; initializing 2500 different variable for this task is not feasible.
- To solve this problem, we can define an array with size 2500 that can hold the marks of all students.
- For example **`int marks[2500];`**

An example program for an array is shown in code snippet below.

```
int marks[] = {23, 45, 56, 89};
cout<<marks[0]<<endl;
cout<<marks[1]<<endl;
cout<<marks[2]<<endl;
cout<<marks[3]<<endl;
```

Code Snippet 1: Array Program 1

As shown in the code snippet, we initialized an array of size 4 in which we have stored marks of 4 students and then printed them one by one. The main point to note here is that array store data in continuous block form in the memory, and array indexes start from 0. Output for the following program is shown in figure 1.

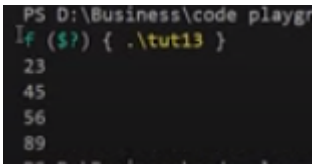


Figure 1: Array Program 1 Output

Another example program to declare an array is shown in code snippet 2.

```
int mathMarks[4];
   mathMarks[0] = 2278;
   mathMarks[1] = 738;
   mathMarks[2] = 378;
   mathMarks[3] = 578;

   cout<<"These are math marks"<<endl;
   cout<<mathMarks[0]<<endl;
   cout<<mathMarks[1]<<endl;
   cout<<mathMarks[2]<<endl;
   cout<<mathMarks[3]<<endl;
```

Code Snippet 2: Array Program 2

As shown in code snippet 2, we have declared an array of size 4 and then assigned values one by one to each index of the array. Output for the following program is shown in figure 2.

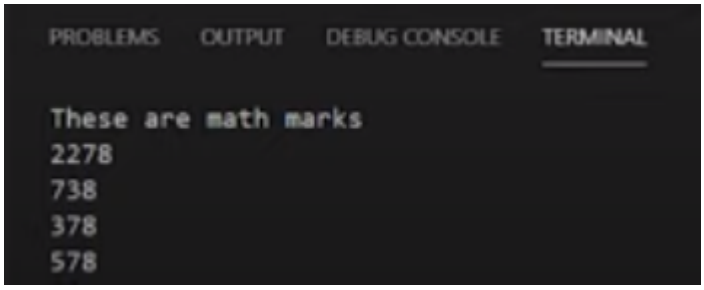


Figure 2: Array Program 2 Output

To change the value at the specific index of an array, we can simply assign the value to that index. For example: **“marks[2] = 333”** can place the value **“333”** at the index **“2”** of the array. We can use loops to print the values of an array, instead of printing them one by one. An example program to print the value of the array with "for" loop is shown in code snippet 3.

```
for (int i = 0; i < 4; i++)
    {
        cout<<"The value of marks "<<i<<" is "<<marks[i]<<endl;
    }
```

Code Snippet 3: Array program with a loop

As shown in code snippet 3, we initialized an integer variable “i" with the value 0 and set the running condition of the loop to the length of an array. In the loop body, each index number and the value at each number is being printed. Output for the following program is shown in figure 3.

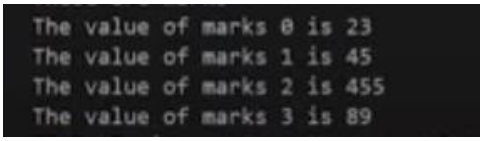


Figure 3: Array program with loop output

Pointers and Arrays

Storing the address of an array into pointer is different than storing the address of a variable into the pointer because the name of the array is an address of the first index of an array. So to use ampersand "&" with the array name for assigning the address to a pointer is wrong.

- &Marks --> Wrong
- Marks --> address of the first block

An example program for storing the starting address of an array in the pointer is shown in code snippet 4.

```
int* p = marks;
cout<<"The value of marks[0] is "<<*p<<endl;
```

Code Snippet 4: Pointer and Array Program

As shown in code snippet 7, we have assigned the address of array “marks” to the pointer variable “*p” and then printed the pointer “*p”. The main thing to note here is that the value at the pointer “*p” is the starting address of the array “marks”. The output for the following program is shown in figure 4.

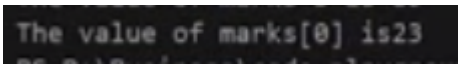


Figure 4: Pointer and Array Program Output

As shown in figure 4, we have printed the value at pointer **"*p"**, and it has shown us the value of the first index of the array **"marks"** because the pointer was pointing at the first index of an array and the value at that index was **"23"**. If we want to access the 2nd index of an array through the pointer, we can simply increment the pointer with 1. For example: **"*(p+1)"** will give us the value of the 2nd index of an array. An example program to print the values of an array through the pointer is shown in code snippet 5.

```
int* p = marks;
    cout<<"The value of *p is "<<*p<<endl;
    cout<<"The value of *(p+1) is "<<*(p+1)<<endl;
    cout<<"The value of *(p+2) is "<<*(p+2)<<endl;
    cout<<"The value of *(p+3) is "<<*(p+3)<<endl;
```

Code Snippet 5: Pointer and Array Program 2

As shown in code snippet 5, 1st we have printed the value at pointer **"*p"**; 2nd we have printed the value at pointer **"*(p+1)"**; 3rd we have printed the value at pointer **"*(p+2)"**; 4th we have printed the value at pointer **"*(p+3)"**. This program will output the values at **"0, 1, 2, 3"** indices of an array **"marks"**. The output of the following program is shown in figure 5.

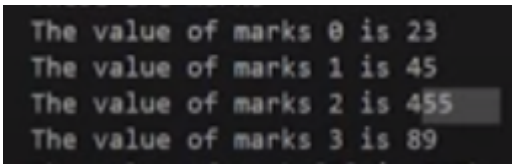


Figure 5: Pointer and Array Program 2 Output

Code as described/written in the video

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

int main(){
    // Array Example
    int marks[] = {23, 45, 56, 89};

    int mathMarks[4];
    mathMarks[0] = 2278;
    mathMarks[1] = 738;
    mathMarks[2] = 378;
    mathMarks[3] = 578;

    cout<<"These are math marks"<<endl;
    cout<<mathMarks[0]<<endl;
    cout<<mathMarks[1]<<endl;
    cout<<mathMarks[2]<<endl;
    cout<<mathMarks[3]<<endl;

    // You can change the value of an array
    marks[2] = 455;
    cout<<"These are marks"<<endl;
    // cout<<marks[0]<<endl;
    // cout<<marks[1]<<endl;
    // cout<<marks[2]<<endl;
    // cout<<marks[3]<<endl;

    for (int i = 0; i < 4; i++)
    {
```

```
        cout<<"The value of marks "<<i<<" is "<<marks[i]<<endl;
    }

    // Quick quiz: do the same using while and do-while loops?

    // Pointers and arrays
    int* p = marks;
    cout<<*(p++)<<endl;
    cout<<*(++p)<<endl;
    // cout<<"The value of *p is "<<*p<<endl;
    // cout<<"The value of *(p+1) is "<<*(p+1)<<endl;
    // cout<<"The value of *(p+2) is "<<*(p+2)<<endl;
    // cout<<"The value of *(p+3) is "<<*(p+3)<<endl;

    return 0;
}
```

Part: 14

Structures, Unions & Enums in C++

In this tutorial, we will discuss structures, unions & enums in C++

Structures in C++

The structure is a user-defined data type that is available in C++. Structures are used to combine different types of data types, just like an array is used to combine the same type of data types. An example program for creating a structure is shown in Code Snippet 1.

```
struct employee
{
    /* data */
    int eId;
    char favChar;
    float salary;
};
```

Code Snippet 1: Creating a Structure Program

As shown in Code Snippet 1, we have created a structure with the name **"employee"**, in which we have declared three variables of different data types **(eId, favchar, salary)**. As we have created a structure now we can create instances of our structure employee. An example program for creating instances of structure employees is shown in Code Snippet 2.

```
int main() {
    struct employee harry;
    harry.eId = 1;
    harry.favChar = 'c';
    harry.salary = 120000000;
    cout<<"The value is "<<harry.eId<<endl;
    cout<<"The value is "<<harry.favChar<<endl;
    cout<<"The value is "<<harry.salary<<endl;
    return 0;
}
```

Code Snippet 2: Creating Structure instances

As shown in Code Snippet 2, 1st we have created a structure variable "`harry`" of type "`employee`", 2nd we have assigned values to `(eId, favchar, salary)` fields of the structure employee and at the end we have printed the value of "`salary`".

Another way to create structure variables without using the keyword "`struct`" and the name of the struct is shown in Code Snippet 3.

```
typedef struct employee
{
    /* data */
    int eId; //4
    char favChar; //1
    float salary; //4
} ep;
```

Code Snippet 3: Creating Structure Program 2

As shown in Code Snippet 3, we have used a keyword **"typedef"** before struct and after the closing bracket of structure, we have written **"ep"**. Now we can create structure variables without using the keyword **"struct"** and name of the struct. An example is shown in Code Snippet 4.

```
int main(){
ep harry;
    struct employee shubham;
    struct employee rohanDas;
    harry.eId = 1;
    harry.favChar = 'c';
    harry.salary = 120000000;
    cout<<"The value is "<<harry.eId<<endl;
    cout<<"The value is "<<harry.favChar<<endl;
    cout<<"The value is "<<harry.salary<<endl;
    return 0;
}
```

Code Snippet 4: Creating Structure instance 2

As shown in Code Snippet 4, we have created a structure instance **"harry"** by just writing **"ep"** before it.

Unions in C++

Unions are similar to structures but they provide better memory management then structures. Unions use shared memory so only 1 variable can be used at a time. An example program to create unions is shown in Code Snippet 5.

```
union money
{
    /* data */
    int rice; //4
    char car; //1
    float pounds; //4
};
```

Code Snippet 5: Creating Unions Program

As shown in Code Snippet 5, we have created a union with the name "money" in which we have declared three variables of different data types (rice, car, pound). The main thing to note here is that:

- We can only use 1 variable at a time otherwise the compiler will give us a garbage value
- The compiler chooses the data type which has maximum memory for the allocation.

An example program for creating an instance of union money is shown in Code Snippet 6.

```
int main(){
        union money m1;
        m1.rice = 34;
        cout<<m1.rice;
        return 0;
}
```

Code Snippet 6: Creating a Union Instance

As shown in Code Snippet 6, 1st we have created a union variable "m1" of type "money", 2nd we have assigned values to (rice) fields of the union money, and in the end, we have printed the value of "rice". The main thing to note here is that once we have assigned a value to the union field "rice", now we cannot use

other fields of the union otherwise we will get garbage value. The output for the following program is shown in figure 1.

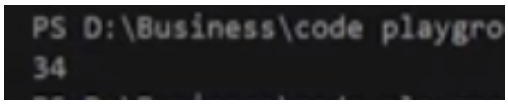


Figure 1: Creating Union Instance Output

Enums in C++

Enums are user-defined types which consist of named constants. Enums are used to make the program more readable. An example program for enums is shown in Code Snippet 8.

```
int main(){
    enum Meal{ breakfast, lunch, dinner};
    Meal m1 = lunch;
    cout<<m1;
    return 0;
}
```

Code Snippet 7: Enums Program

As shown in Code Snippet 7, 1st we have created an enum **"Meal"** in which we have stored three named constants (breakfast, lunch, dinner). 2nd we have assigned the value of **"lunch"** to the variable **"m1"** and at the end, we have printed **"m1"**. The main thing to note here is that (breakfast, lunch, dinner) are constants; the value for **"breakfast"** is **"0"**, the value for **"lunch"** is **"1"** and the value for **"dinner"** is **"2"**. The output for the following program is shown in figure 2.

Figure 2: Enums Program Output

Code as described/written in the video

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

typedef struct employee
{
    /* data */
    int eId; //4
    char favChar; //1
    float salary; //4
} ep;

union money
{
    /* data */
    int rice; //4
    char car; //1
    float pounds; //4
};


int main(){
    enum Meal{ breakfast, lunch, dinner};
    Meal m1 = lunch;
    cout<<(m1==2);
    // cout<<breakfast;
```

```
    // cout<<lunch;
    // cout<<dinner;
    // union money m1;
    // m1.rice = 34;
    // m1.car = 'c';
    // cout<<m1.car;

    // ep harry;
    // struct employee shubham;
    // struct employee rohanDas;
    // harry.eId = 1;
    // harry.favChar = 'c';
    // harry.salary = 120000000;
    // cout<<"The value is "<<harry.eId<<endl;
    // cout<<"The value is "<<harry.favChar<<endl;
    // cout<<"The value is "<<harry.salary<<endl;
    return 0;
}
```

Part: 15

Functions & Function Prototypes in C++

In this tutorial, we will discuss functions and functions prototype in C++

Functions in C++

Functions are the main part of top-down structured programming. We break the code into small pieces and make functions of that code. Functions help us to reuse the code easily. An example program for the function is shown in Code Snippet 1.

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

Code Snippet 1: Function example

As shown in Code Snippet 1, we created an integer function with the name of sum, which takes two parameters **"int a"** and **"int b"**. In the function, body addition is performed on the values of variable **"a"** and variable **"b"** and the result is stored in variable **"c"**. In the end, the value of variable **"c"** is returned to the function. We have seen how this function works now we will see how to pass values to the function parameters. An example program for passing the values to the function is shown in Code Snippet 2.

```
int main(){
    int num1, num2;
    cout<<"Enter first number"<<endl;
    cin>>num1;
    cout<<"Enter second number"<<endl;
    cin>>num2;
    cout<<"The sum is "<<sum(num1, num2);
    return 0;
}
```

Code Snippet 2: Passing Value to Function Parameters

As shown in Code Snippet 2, we have declared two integer variables **"num1"** and **"num2"**, we will take their input at run time. In the end, we called the **"sum"** function and passed both variables **"num1"** and **"num2"** into sum function. **"sum"** function will perform the addition and returns the value at the same location from where it was called. The output of the following program is shown in figure 1.

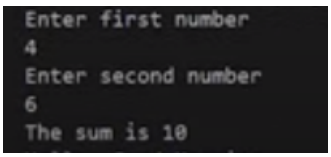


Figure 1: Function Output

Function Prototype in C++

The function prototype is the template of the function which tells the details of the function e.g (name, parameters) to the compiler. Function prototypes help us to define a function after the function call. An example of a function prototype is shown in Code Snippet 3.

```
// Function prototype
    int sum(int a, int b);
```

Code Snippet 3: Function Prototype

As shown in Code Snippet 3, we have made a function prototype of the function **"sum"**, this function prototype will tell the compiler that the function **"sum"** is declared somewhere in the program which takes two integer parameters and returns an integer value. Some examples of acceptable and not acceptable prototypes are shown below:

- int sum(int a, int b); //Acceptable
- int sum(int a, b); // Not Acceptable
- int sum(int, int); //Acceptable

Formal Parameters

The variables which are declared in the function are called a formal parameter. For example, as shown in Code Snippet 1, the variables **"a"** and **"b"** are the formal parameters.

Actual Parameters

The values which are passed to the function are called actual parameters. For example, as shown in Code Snippet 2, the variables **"num1"** and **"num2"** are the actual parameters.

The function doesn't need to have parameters or it should return some value. An example of the void function is shown in Code Snippet 4.

```
void g(){
    cout<<"\nHello, Good Morning";
}
```

Code Snippet 4: Void Function

As shown in Code Snippet 4, void as a return type means that this function will not return anything, and this function has no parameters. Whenever we will call this function it will print **"Hello, Good Morning"**

Code as described/written in the video

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

// Function prototype
// type function-name (arguments);
// int sum(int a, int b); //--> Acceptable
// int sum(int a, b); //--> Not Acceptable
int sum(int, int); //--> Acceptable
// void g(void); //--> Acceptable
void g(); //--> Acceptable

int main(){
    int num1, num2;
    cout<<"Enter first number"<<endl;
    cin>>num1;
    cout<<"Enter second number"<<endl;
    cin>>num2;
    // num1 and num2 are actual parameters
    cout<<"The sum is "<<sum(num1, num2);
    g();
    return 0;
}
```

```
int sum(int a, int b){
    // Formal Parameters a and b will be taking values from actual parameters num1
and num2.
    int c = a+b;
    return c;
}

void g(){
    cout<<"\nHello, Good Morning";
}
```

Part: 16

Call by Value & Call by Reference in C++

In this tutorial, we will discuss call by value and call by reference in C++

Call by Value in C++

Call by value is a method in C++ to pass the values to the function arguments. In case of call by value the copies of actual parameters are sent to the formal parameter, which means that if we change the values inside the function that will not affect the actual values. An example program for the call by value is shown in Code Snippet 1.

```
void swap(int a, int b){ //temp a b
    int temp = a;        //4   4  5
    a = b;               //4   5  5
    b = temp;            //4   5  4
}
```

Code Snippet 1: Call by Value Swap Function

As shown in Code Snippet 1, we created a swap function which is taking two parameters **"int a"** and **"int b"**. In function body values of the variable, **"a"** and **"b"** are swapped. An example program is shown in Code Snippet 2, which calls the swap function and passes values to it.

```
int main(){
    int x =4, y=5;
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    swap(x, y);
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Code Snippet 2: Passing Values to Swap Function

As shown in Code Snippet 2, we have initialized two integer variables **"a"** and **"b"** and printed their values. Then we called a **"swap"** function and passed values of variables **"a"** and **"b"** and again printed the values of variables **"a"** and **"b"**. The output for the following program is shown in figure 1.

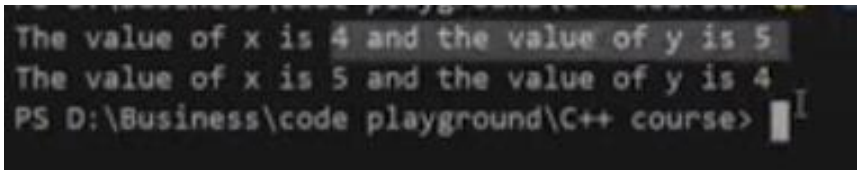


Figure 1: Call by Value Swap Function Output

As shown in figure 3, the values of **"a"** and **"b"** are the same for both times they are printed. So the main point here is that when the call by value method is used it doesn't change the actual values because copies of actual values are sent to the function.

Call by Pointer in C++

A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for the call by reference is shown in Code Snippet 3.

```
// Call by reference using pointers
void swapPointer(int* a, int* b){  //temp a b
    int temp = *a;                 //4   4  5
    *a = *b;                       //4   5  5
    *b = temp;                     //4   5  4
}
```

Code Snippet 3: Call by Pointer Swap Function

As shown in Code Snippet 3, we created a swap function which is taking two pointer parameters **"int* a"** and **"int* b"**. In function body values of pointer variables, **"a"** and **"b"** are swapped. An example program is shown in Code Snippet 4, which calls the swap function and passes values to it.

```
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);   //This will swap a and b using pointer reference
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Code Snippet 4: Passing Values to Call by Pointer Swap Function

As shown in Code Snippet 4, we have initialized two integer variables **"a"** and **"b"** and printed their values. Then we called a **"swap"** function and passed addresses of variables **"a"** and **"b"** and again printed the values of variables **"a"** and **"b"**. The output for the following program is shown in figure 2.

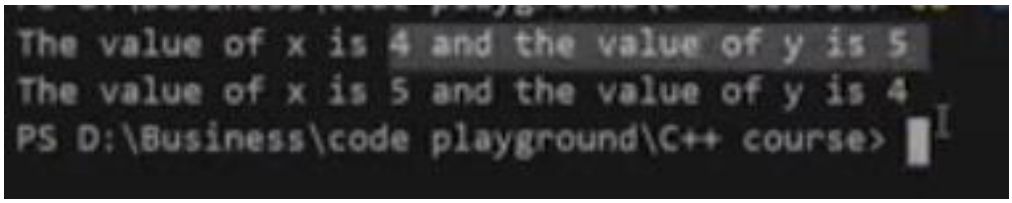


Figure 2: Call by Pointer Swap Function Output

As shown in figure 2, the values of "a" and "b" are swapped when the swap function is called. So the main point here is that when the call by pointer method is used it changes the actual values because addresses of actual values are sent to the function.

Call by Reference in C++

Call by reference is a method in C++ to pass the values to the function arguments. In the case of call by reference, the reference of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for a call by reference is shown in Code Snippet 5.

```
void swapReferenceVar(int &a, int &b){      //temp a b
    int temp = a;                           //4   4  5
    a = b;                                  //4   5  5
    b = temp;                               //4   5  4
}
```

Code Snippet 5: Call by Reference Swap Function

As shown in Code Snippet 5, we created a swap function that is taking reference of **"int &a"** and **"int &b"** as parameters. In function body values of variables, **"a"** and **"b"** are swapped. An example program is shown in Code Snippet 6, which calls the swap function and passes values to it.

```
int main(){
    int x =4, y=5;
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    swapReferenceVar(x, y); //This will swap a and b using reference variables
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Code Snippet 6: Passing Values to Call by Reference Swap Function

As shown in Code Snippet 6, we have initialized two integer variables **"a"** and **"b"** and printed their values. Then we called a **"swap"** function and passed variables **"a"** and **"b"** and again printed the values of variables **"a"** and **"b"**. The output for the following program is shown in figure 3.

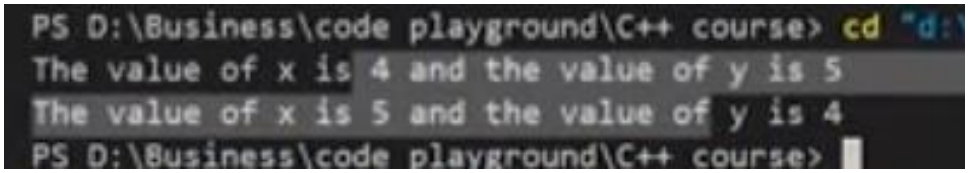


Figure 3: Call by Reference Swap Function Output

As shown in figure 3, the values of **"a"** and **"b"** are swapped when the swap function is called. So the main point here is that when the call by reference method is used it changes the actual values because references of actual values are sent to the function.

Code as described/written in the video

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

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

// This will not swap a and b
void swap(int a, int b){ //temp a b
    int temp = a;        //4   4  5
    a = b;               //4   5  5
    b = temp;            //4   5  4
}

// Call by reference using pointers
void swapPointer(int* a, int* b){ //temp a b
    int temp = *a;          //4   4  5
    *a = *b;                //4   5  5
    *b = temp;              //4   5  4
}

// Call by reference using C++ reference Variables
// int &
void swapReferenceVar(int &a, int &b){ //temp a b
    int temp = a;          //4   4  5
    a = b;                 //4   5  5
    b = temp;              //4   5  4
    // return a;
}

int main(){
    int x =4, y=5;
    // cout<<"The sum of 4 and 5 is "<<sum(a, b);
```

```
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    // swap(x, y); // This will not swap a and b
    // swapPointer(&x, &y); //This will swap a and b using pointer reference
    swapReferenceVar(x, y); //This will swap a and b using reference variables
    // swapReferenceVar(x, y) = 766; //This will swap a and b using reference
variables
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Part: 17

Inline Functions, Default Arguments & Constant Arguments in C

In this tutorial, we will discuss inline functions, default arguments, and constant arguments in C++

Inline Functions in C++

Inline functions are used to reduce the function call. When one function is being called multiply times in the program it increases the execution time, so inline function is used to reduce time and increase program efficiency. If the inline function is being used when the function is called, the inline function expands the whole function code at the point of a function call, instead of running the function. Inline functions are considered to be used when the function is small otherwise it will not perform well. Inline is not recommended when static variables are being used in the function. An example of an inline function is shown in Code Snippet 1.

```
inline int product(int a, int b){
    return a*b;
}
```

Code Snippet 1: Inline function

As shown in Code Snippet 1, 1st inline keyword is used to make the function inline. 2nd a product function is created which has two arguments and returns the product of them. Now we will call the product function multiple times in our main program which is shown in Code Snippet 2.

```
int main(){
    int a, b;
    cout<<"Enter the value of a and b"<<endl;
    cin>>a>>b;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    return 0;
}
```

Code Snippet 2: Calling Inline Product Function

As shown in Code Snippet 2, we called the product function multiple times. The main thing to note here is that the function will not run instead of it the function code will be copied at the place where the function is being called. This will increase the execution time of the program because the compiler doesn't have to copy the values and get the return value again and again from the compiler. The output of the following program is shown in figure 1.

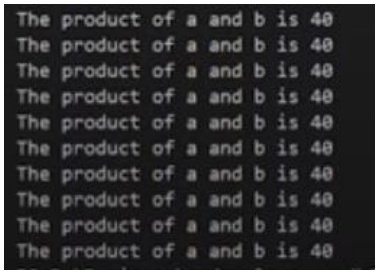


Figure 1: Inline Function Output

Static Keyword in C++

Static keyword has different meanings when used with different types. We can use static keyword with:

Static Variables: Variables in a function, Variables in a class

Static Members of Class: Class objects and Functions in a class

Let us now look at each one of these use of static in details:

Static Variables

- **Static variables in a Function**: When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call. This is useful for implementing coroutines in C/C++ or any other application where previous state of function needs to be stored.

```
// C++ program to
demonstrate
// the use of static Static
// variables in a Function
#include <iostream>
#include <string>
using namespace std;

void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";
        // value is updated and
        // will be carried to
    next
        // function calls
        count++;
    }

int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

Output:

```
0 1 2 3 4
```

- You can see in the above program that the variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.

  As a side note, Java doesn't allow static local variables in functions.

- **Static variables in a class**: As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables **in a class are shared by the objects.** There cannot be multiple copies of same static variables for different objects. Also because of this reason static variables cannot be initialized using constructors.

```
// C++ program to demonstrate
static
// variables inside a class

#include<iostream>
using namespace std;

class GfG
{
   public:
     static int i;
     GfG()
     {
        // Do nothing
     };
};

int main()
{
  GfG obj1;
  GfG obj2;
```

```
    obj1.i =2;
    obj2.i = 3;
    // prints value of i
    cout << obj1.i<<" "<<obj2.i;
}
```

- You can see in the above program that we have tried to create multiple copies of the static variable i for multiple objects. But this didn't happen. So, a static variable inside a class should be initialized explicitly by the user using the class name and scope resolution operator outside the class as shown below:

```
// C++ program to demonstrate
static
// variables inside a class

#include<iostream>
using namespace std;

class GfG
{
public:
    static int i;

    GfG()
    {
        // Do nothing
    };
};

int GfG::i = 1;

int main()
{
    GfG obj;
    // prints value of i
    cout << obj.i;
}
```

Output:

```
1
```

Static Members of Class

- **Class objects as static:** Just like variables, objects also when declared as static have a scope till the lifetime of program.

  Consider the below program where the object is non-static.

```
// CPP program to illustrate
// when not using static keyword
#include<iostream>
using namespace std;

class GfG
{
    int i;
    public:
        GfG()
        {
            i = 0;
            cout << "Inside
Constructor\n";
        }
        ~GfG()
        {
            cout << "Inside
Destructor\n";
        }
};

int main()
{
    int x = 0;
    if (x==0)
    {
        GfG obj;
    }
    cout << "End of main\n";
}
```

Output:

```
Inside Constructor
Inside Destructor
End of main
```

- In the above program the object is declared inside the if block as non-static. So, the scope of variable is inside the if block only. So, when the object is created the constructor is invoked and soon as the control of if block gets over the destructor is invoked as the scope of object is inside the if block only where it is declared.

  Let us now see the change in output if we declare the object as static.

```
// CPP program to illustrate
// class objects as static
#include<iostream>
using namespace std;

class GfG
{
    int i = 0;

    public:
    GfG()
    {
        i = 0;
        cout << "Inside
Constructor\n";
    }
    ~GfG()
    {
        cout << "Inside
Destructor\n";
    }
};

int main()
{
    int x = 0;
    if (x==0)
    {
        static GfG obj;
    }
    cout << "End of main\n";
}
```

Output:

```
Inside Constructor
End of main
Inside Destructor
```

- You can clearly see the change in output. Now the destructor is invoked after the end of main. This happened because the scope of static object is throughout the life time of program.
- **Static functions in a class**: Just like the static data members or static variables inside the class, static member functions also don't depend on object of class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator.

  Static member functions are allowed to access only the static data members or other static member functions, they cannot access the non-static data members or member functions of the class.

```
// C++ program to demonstrate static
// member function in a class
#include<iostream>
using namespace std;

class GfG
{
   public:

    // static member function
    static void printMsg()
    {
        cout<<"Welcome to GfG!";
    }
};

// main function
int main()
{
    // invoking a static member
function
    GfG::printMsg();
}
```

Default Arguments in C++

Default arguments are those values which are used by the function if we don't input our value. It is recommended to write default arguments after the other arguments. An example program for default arguments is shown in Code Snippet 3.

```
float moneyReceived(int currentMoney, float factor=1.04){
    return currentMoney * factor;
}

int main(){
    int money = 100000;
    cout<<"If you have "<<money<<" Rs in your bank account, you will recive
"<<moneyReceived(money)<< "Rs after 1 year"<<endl;
    cout<<"For VIP: If you have "<<money<<" Rs in your bank account, you will recive
"<<moneyReceived(money, 1.1)<< " Rs after 1 year";
    return 0;
}
```

Code Snippet 3: Default Argument Example Program

As shown in Code Snippet 3, we created a **"moneyReceived"** function which has two arguments **"int currentMoney"** and **"float factor=1.04"**. This function returns the product of **"currentMoney"** and **"factor"**. In our main function, we called **"moneyReceived"** function and passed one argument **"money"**. Again we called **"moneyReceived"** function and passed two arguments **"money"** and **"1.1"**. The main thing to note here is that when we passed only one argument **"money"** to the function at that time the default value of the argument **"factor"** will be used. But when we passed both arguments then the default value will not be used. The output for the following program is shown in figure 2.

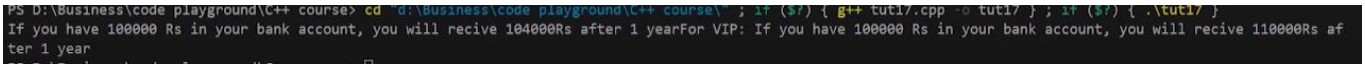


Figure 2: Default Argument Example Program Output

Constant Arguments in C++

Constant arguments are used when you don't want your values to be changed or modified by the function. An example of constant arguments is shown in Code Snippet 4.

```
int strlen(const char *p){

}
```

Code Snippet 4: Constant Arguments Example

As shown in Code Snippet 4, we created a **"strlen"** function which takes a constant argument **"p"**. As the argument is constant so its value won't be modified.

Code as described/written in the video

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

inline int product(int a, int b){
    // Not recommended to use below lines with inline functions
    // static int c=0; // This executes only once
    // c = c + 1; // Next time this function is run, the value of c will be retained
    return a*b;
}
```

```
float moneyReceived(int currentMoney, float factor=1.04){
    return currentMoney * factor;
}

// int strlen(const char *p){

// }
int main(){
    int a, b;
    // cout<<"Enter the value of a and b"<<endl;
    // cin>>a>>b;
    // cout<<"The product of a and b is "<<product(a,b)<<endl;
    int money = 100000;
    cout<<"If you have "<<money<<" Rs in your bank account, you will recive
"<<moneyReceived(money)<< "Rs after 1 year"<<endl;
    cout<<"For VIP: If you have "<<money<<" Rs in your bank account, you will recive
"<<moneyReceived(money, 1.1)<< " Rs after 1 year";
    return 0;
}
```

Part: 18

Recursions & Recursive Functions in C++

In this tutorial, we will discuss recursion and recursive functions in C++

Recursion and Recursive Function

When a function calls itself. It is called recursion and the function which is calling itself is called a recursive function. The recursive function consists of a base case and recursive condition. It is very important to add a base case in recursive function otherwise recursive function will never stop executing. An example of the recursive function is shown in Code Snippet 1.

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

Code Snippet 1: Factorial Recursive Function

As shown in Code Snippet 1, we created a **"factorial"** function which takes one argument. In the function body, there is a base case which checks that if the value of variable **"n"** is smaller or equal to **"1"** if the condition is **"true"** return **"1"**. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a factorial. These are the steps which will be performed by recursive condition:

- 4 * factorial(4-1)
- 4 * 3 * factorial(3-1)
- 4* 3 * 2 * factorial(2-1)
- 4 * 3 * 2 * 1

An example to pass the value to the recursive factorial function is shown in Code Snippet 2.

```
int main(){
    int a;
    cout<<"Enter a number"<<endl;
    cin>>a;
    cout<<"The factorial of "<<a<< " is "<<factorial(a)<<endl;
    return 0;
}
```

Code Snippet 2: Factorial Recursive Function Call

As shown in Code Snippet 2, we created an integer variable "a", which takes input at the runtime and that value is passed to the factorial function. The output for the following program is shown in figure 1.

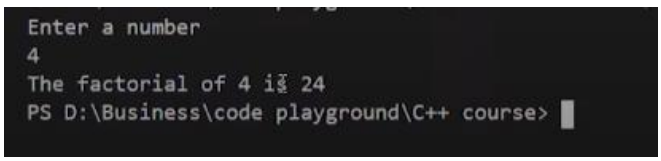


Figure 1: Factorial Recursive Function Output

As shown in figure 1, we input the value **"4"** and it gives us the factorial of it which is **"24"**. Another example of a recursive function for the Fibonacci series is shown in Code Snippet 3.

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

Code Snippet 3: Fibonacci Recursive Function

As shown in Code Snippet 3, we created a **"fib"** function which takes one argument. In the function body, there is a base case which checks that if the value of variable **"n"** is smaller than **"2"**, if the condition is **"true"** return **"1"**. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a Fibonacci number. An example to pass the value to the Fibonacci function is shown in Code Snippet 4.

```
int main(){
    int a;
    cout<<"Enter a number"<<endl;
    cin>>a;
    cout<<"The term in fibonacci sequence at position "<<a<< " is "<<fib(a)<<endl;
    return 0;
}
```

Code Snippet 4: Fibonacci Recursive Function Call

As shown in Code Snippet 4, we created an integer variable **"a"**, which takes input at the runtime and that value is passed to the Fibonacci function. The output for the following program is shown in figure 2.

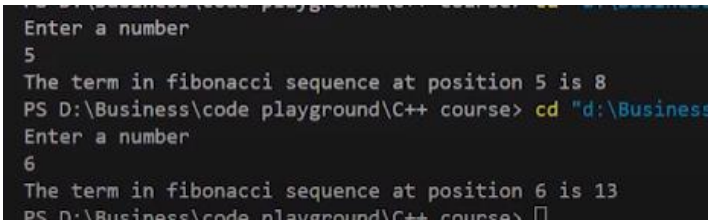


Figure 2: Fibonacci Recursive Function Output

As shown in figure 2, 1st we input the value **"5"** and it gives us the Fibonacci number at that place which is **"8"**. 2nd we input the value **"6"** and it gives us the Fibonacci number at that place which is **"13"**.

One thing to note here is that recursive functions are not always the best option. They perform well in some problems but not in every problem.

Code as described/written in the video

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

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

// fib(5)
// fib(4) + fib(3)
// fib(2) + fib(3) + fib(2) + fib(3)

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

// Step by step calculation of factorial(4)
// 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(){
    // Factorial of a number:
    // 6! = 6*5*4*3*2*1 = 720
    // 0! = 1 by definition
    // 1! = 1 by definition
    // n! = n * (n-1)!
    int a;
    cout<<"Enter a number"<<endl;
    cin>>a;
    // cout<<"The factorial of "<<a<< " is "<<factorial(a)<<endl;
    cout<<"The term in fibonacci sequence at position "<<a<< " is "<<fib(a)<<endl;
    return 0;
}
```

Part: 19

Function Overloading with Examples in C++

In this tutorial, we will discuss function overloading in C++

Function Overloading in C++

Function overloading is a process to make more than one function with the same name but different parameters, numbers, or sequence. An example program to explain function overloading is shown in Code Snippet 1.

```
int sum(float a, int b){
    cout<<"Using function with 2 arguments"<<endl;
    return a+b;
}

int sum(int a, int b, int c){
    cout<<"Using function with 3 arguments"<<endl;
    return a+b+c;
}
```

Code Snippet 1: Sum Function Overloading Example

As shown in Code Snippet 1, we have created two **"sum"** functions, the 1st **"sum"** function takes two arguments **"int a"**, **"int b"** and return the sum of those two variables; and the 2nd sum function is taking three arguments **"int a"**, **"int b"**, **"int c"** and return the sum of those three variables. Function call for these **"sum"** function is shown in Code Snippet 2.

```
int main(){
    cout<<"The sum of 3 and 6 is "<<sum(3,6)<<endl;
    cout<<"The sum of 3, 7 and 6 is "<<sum(3, 7, 6)<<endl;
    return 0;
}
```

Code Snippet 2: Sum Function Call

As shown in Code Snippet 2, we passed two arguments in the first function call and three arguments in the second function call. The output of the following program is shown in figure 1.

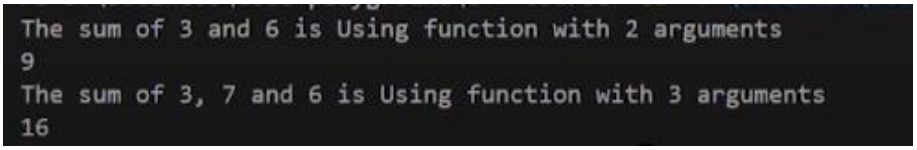


Figure 1: Sum Function Output

As shown in Code Snippet 3, both the **"sum"** function runs fine and gives us the required output. The main thing to note here is that the name of the function can be the same but the data type and the sequence of arguments need to be different as shown in the example program otherwise program will not run.

Another example of function overloading is shown in Code Snippet 3.

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

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

```
}

// Rectangular box
int volume (int l, int b, int h){
    return (l*b*h);
}
```

Code Snippet 3: Volume Function Overloading Example

As shown in Code Snippet 3, we have created three **"volume"** functions, the 1st **"volume"** function calculates the volume of the cylinder and has two arguments **"double r"** and **"int h"**; the 2nd **"volume"** function calculates the volume of the cube and has one argument **"int a"**; the 3rd **"volume"** function calculates the volume of the rectangular box and has three arguments **"int l"**, **"int b"** and **"int h"**. The function call for these **"volumes"** function is shown in Code Snippet 4.

```
int main(){
    cout<<"The volume of cuboid of 3, 7 and 6 is "<<volume(3, 7, 6)<<endl;
    cout<<"The volume of cylinder of radius 3 and height 6 is "<<volume(3, 6)<<endl;
    cout<<"The volume of cube of side 3 is "<<volume(3)<<endl;
    return 0;
}
```

Code Snippet 4: Volume Function Call

As shown in Code Snippet 4, we passed three arguments in the first function call, two arguments in the second function call, and one argument in the third function call. The output of the following program is shown in figure 2.

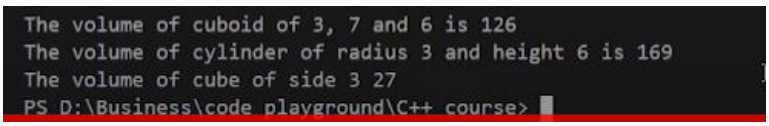


Figure 2: Volume Function Output

As shown in figure 2, all three **"volume"** functions run fine and give us the required output.

Code as described/written in the video

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

int sum(float a, int b){
    cout<<"Using function with 2 arguments"<<endl;
    return a+b;
}

int sum(int a, int b, int c){
    cout<<"Using function with 3 arguments"<<endl;
    return a+b+c;
}

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

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

```
// Rectangular box
int volume (int l, int b, int h){
    return (l*b*h);
}

int main(){
    cout<<"The sum of 3 and 6 is "<<sum(3,6)<<endl;
    cout<<"The sum of 3, 7 and 6 is "<<sum(3, 7, 6)<<endl;
    cout<<"The volume of cuboid of 3, 7 and 6 is "<<volume(3, 7, 6)<<endl;
    cout<<"The volume of cylinder of radius 3 and height 6 is "<<volume(3, 6)<<endl;
    cout<<"The volume of cube of side 3 is "<<volume(3)<<endl;
    return 0;
}
```

Part: 20

Object Oriented Programming in C++

In this series of our C++ tutorials, we will visualize object-oriented programming in the C++ language. In our last lecture, we discussed function overloading in C++.

Why Object-Oriented Programming?

Before we discuss object-oriented programming, we need to learn why we need object-oriented programming?

- C++ language was designed with the main intention of adding object-oriented programming to C language
- As the size of the program increases readability, maintainability, and bug-free nature of the program decrease.
- This was the major problem with languages like C which relied upon functions or procedure (hence the name procedural programming language)
- As a result, the possibility of not addressing the problem adequately was high
- Also, data was almost neglected, data security was easily compromised
- Using classes solves this problem by modeling program as a real-world scenario

Difference between Procedure Oriented Programming and Object-Oriented Programming

Procedure Oriented Programming

- Consists of writing a set of instruction for the computer to follow
- The main focus is on functions and not on the flow of data
- Functions can either use local or global data
- Data moves openly from function to function

Object-Oriented Programming

- Works on the concept of classes and object
- A class is a template to create objects
- Treats data as a critical element
- Decomposes the problem in objects and builds data and functions around the objects

Basic Concepts in Object-Oriented Programming

- **Classes** - Basic template for creating objects
- **Objects –** Basic run-time entities
- **Data Abstraction & Encapsulation –** Wrapping data and functions into a 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 –** message (Information) call format

Benefits of Object-Oriented Programming

- Better code reusability using objects and inheritance
- Principle of data hiding helps build secure systems
- Multiple Objects can co-exist without any interference
- Software complexity can be easily managed

```
No Source Code Associated With This Video
```