Unit – 3
C++ Language Constructs
Lecture 4
OOP
BCT / BEI –II Semester

Prepared By:

Rama Bastola

Contents

Dynamic Memory Allocation

- In some situations, memory required to store particular information in a defined variable is not known in advance
- Allocating the required memory for the variable at run time is known as dynamic memory allocation.
- **new** operator is used to allocate memory at run time for the variable of a given type which returns the address of the space allocated
- Dynamically allocated memory anymore can be deallocated using **delete** operator, which “clean-up” memory that was previously allocated by new operator.

new operator

- The new operator denotes a request for memory allocation on the Heap.
- If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.
- **Syntax:**

```
pointer-variable = new data-type;
```

- **Example :**

```
int *p = NULL;
p = new int;
```

```
int *p = new int;
```

new operator: Initialize Memory

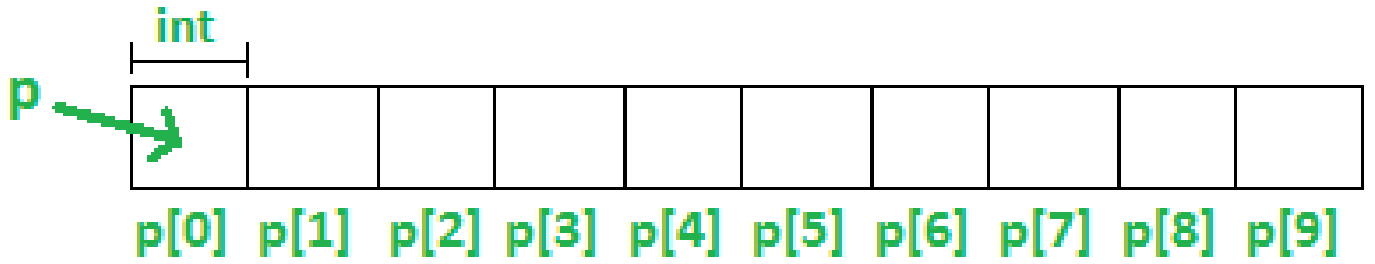
```
pointer-variable = new data-type(value);
Example:
int *p = new int(25);
float *q = new float(75.25);
```

Allocate block of memory:

- new operator is also used to allocate a block(an array) of memory of type *data-type*.

```
pointer-variable = new data-type[size];
```

```
Example:
        int *p = new int[10]
```



delete operator

- **delete** operator is used to deallocate dynamically allocated memory by programmers.
- Syntax:

```
delete pointer-variable;
```

  - Where, pointer-variable is the pointer that points to the data object created by ***new***.
  - Example: **delete p;**
- To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
delete[] pointer-variable;
```

- Example:

```
// It will free the entire array
// pointed by p.
delete[] p;
```

Program : Dynamic Memory Allocation using new / delete operator

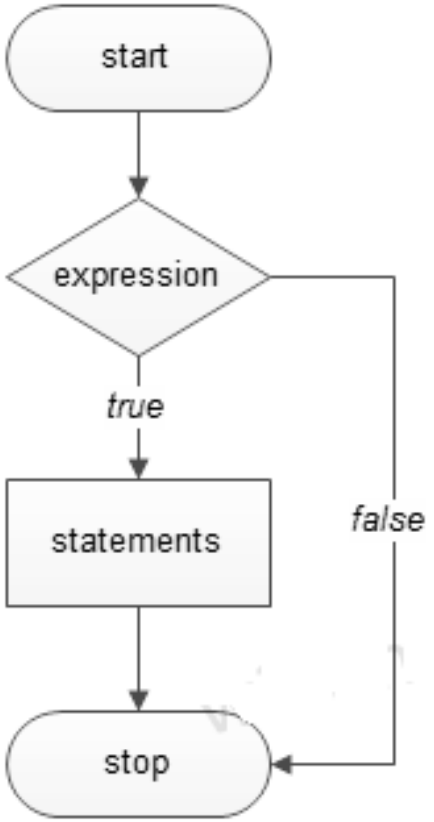
```
// Dynamic Memory allocation using new and delete
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;
    // memory allocation of num number of floats
    ptr = new float[num];
    cout << "Enter marks of students." << endl;
    for (int i = 0; i < num; i++)
    {
        cout << "Student" << i + 1 << ": ";
        cin >> *(ptr + i);
    }
    cout << "\nDisplaying marks of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << " :" << *(ptr + i) << endl;
    }
    delete [] ptr;  // ptr memory is released
    return 0;
}
```

```
Enter total number of students: 4
Enter marks of students.
Student1: 60.5
Student2: 70.5
Student3: 65.5
Student4: 85

Displaying marks of students.
Student1 :60.5
Student2 :70.5
Student3 :65.5
Student4 :85
```

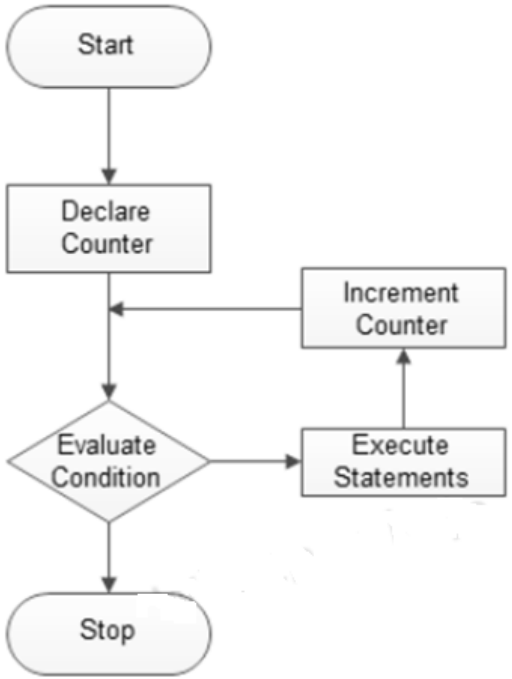
C++ Decision Making or Conditional Statements

- allow to make a decision, based upon the result of a condition.
- Decision making statements in C++
  - if statement
  - if-else statement
  - else-if statement
  - goto statement
  - switch statement
  - Conditional Operator



C++ Loops

- In general, statements get executed sequentially with a C++ program, one statement followed by another.
- C++ provides statements for several control structures along with iteration/repetition capability that allows programmers to execute a statement or group of statements multiple times
- C++ supports the following loops:
    - while loops
    - do while loops
    - for loops



C++ Loop Control Statements

- Loop control statements are used to change the normal sequence of execution of the loop

Statement	Syntax	Description
break statement	break;	Is used to terminate loop or switch statements.
continue statement	continue;	Is used to suspend the execution of current loop iteration and transfer control to the loop for the next iteration.
goto statement	goto labelName;labelName: statement;	It transfers current program execution sequence to some other part of the program.

Functions

function is a self-contained block of statements that can be executed repeatedly whenever we need it.

Function Prototype

(function declaration)

Syntax:

```
dataType functionName (Parameter List)
```

Example:

```
int addition();
```

Function Definition

Syntax:

```
returnType functionName(Function arguments){
  //body of the function
}
```

Example:

```
int addition()
{

}
```

Function Call

A function can be called or accessed by specifying its name followed by a list of arguments enclosed in parenthesis and separated by commas.

Function Overloading

- Multiple functions having the same name but different parameters (with a change in type, sequence or number), which can be use to perform a similar form of operations is known as function overloading.
- It is the ability to create multiple functions with same name
  - slightly different implementation
  - and depending on the context (type, sequence, and a number of the value passed), the appropriate function gets invoked.

Function Overloading : Example

- Example:

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

- Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.
- ***Overloaded functions may or may not have different return type but it should have different argument(s).***
- The number and type of arguments passed to these two functions are same even though the return type is different.

Hence, the compiler will throw error

```
// Error code
int test(int a) { }
double test(int b){ }
```

Function Overloading : Program

```
#include <iostream>
using namespace std;
void display(int);
void display(float);
void display(int, float);
int main() {
    int a = 20;
    float b = 50.5;
    display(a);
    display(b);
    display(a, b);
    return 0;
}
void display(int var) {
    cout << "Integer number: " << var << endl;
}
void display(float var) {
    cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
    cout << "Integer number: " << var1 << " and float number:" << var2;
}
```

```
Integer number: 20
Float number: 50.5
Integer number: 20 and float number:50.5
```

Inline Function [1]

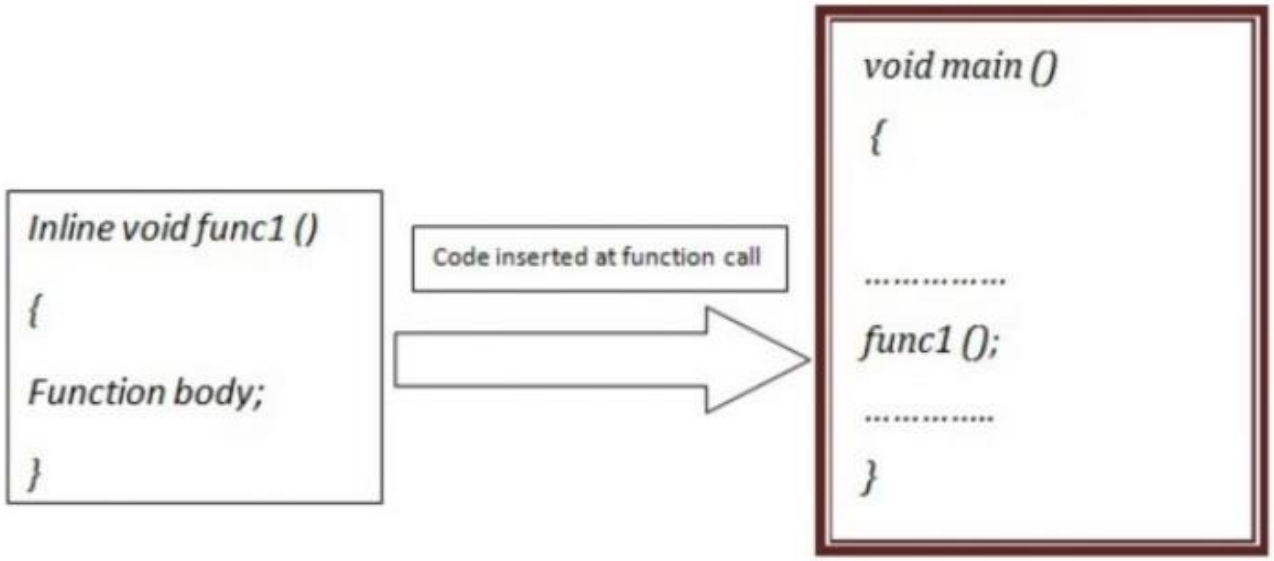
- Normally, a function call transfers the control from the calling program to the called function.
- After the execution of the program, the called function returns the control to the calling program with a return value.
- This concept of function saves program space because instead of writing same code multiple times the function stored in a place can be simply used by calling it at a desired place in the program.
- This might be handy to **reduce the program size** but it **definitely increases the execution time** of the program as the function is invoked every time the control is passed to the function and returns a value after execution.
- In large functions, this is very helpful but in a small function in order to save execution time a user may wish to put the code of function definition directly in the line of called location.
- For this C++ provides inline function to reduce function call overhead. That is every time a function is called the compiler generate a copy of the function's code in place to avoid function call.
- This type of function whose code is copied to the called location is called **inline function**.

Inline Function [2]

- Syntax:

```
inline data_type function_name(arguments_list);
```

- **Note:**
  - Inline function should be used only with small and frequently used function because it makes the program to take more memory.
  - Definition of inline function is written before it is used in the program because the compiler knows how to place the function into its inlined code.



Inline Function [3]

```
// inlinefunction.cpp
 #include <iostream>
using namespace std;
 inline int square(int n)    //inline function
  {
    n*= n;
  }
 int main()
 {
   int num;
   cout << "enter number: ";
   cin >> num;
   cout<<"Square of the Number:"<<square(num)<<endl;
   return 0;
 }
```

```
enter number: 5
Square of the Number:25
```

Default Argument

- In C, when a function is called, the number of argument and parameters must be same.
- In C++, there is a provision of supplying less number of arguments than the actual number of parameters. This mechanism is supported by default argument.
- If we do not supply any argument, the default value is used for the argument that is absent in the function call.
- The default values are specified when function is declared.

Default Argument : working

Case1: No argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp( );
}

void temp(int i, float f ) {
... ... ...
}
```

Case2: First argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp(6);
}

void temp(int i, float f ) {
... ... ...
}
```

Case3: All arguments Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp(6, -2.3 );
}

void temp(int i, float f ) {
... ... ...
}
```

Case4: Second argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
    temp( 3.4);
}

void temp(int i, float f ) {
... ... ...
}
```

i = 3, f = 8.8
Because, only the second argument cannot be passed
The parameter will be passed as the first argument.

```cpp
// C++ Program to demonstrate working of default argument
#include <iostream>
using namespace std;
void display(char = '#', int = 1);
int main()
{
    cout << "No argument passed:\n";
    display();
    cout << "\nFirst argument passed:\n";
    display('*');
    cout << "\nBoth argument passed:\n";
    display('@', 10);
    return 0;
}

void display(char c, int n)
{
    for(int i = 1; i <= n; ++i)
    {
        cout << c;
    }
    cout << endl;
}
```

```
No argument passed:
#

First argument passed:
*

Both argument passed:
@@@@@@@@@@
```

Can we implement function with default argument by overloaded function?

Reference Variable

- A reference variable is **an alternative name or alias** for a variable
- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- A variable can be declared as reference by putting ‘&’ in the declaration.
- Syntax:

  data_type &reference_name = variable_name

  Example: int num=20;

  int &rnum = num;

- A reference variable cannot be initialized with constant value as:

  int &ptr= 5; // error

  char &ch = ‘\n’; // error

References vs Pointers

- Major differences between references and pointers are −
  - You cannot have NULL references.
    - You can have NULL pointers
  - Once a reference is initialized to an object, it cannot be changed to refer to another object.
    - Pointers can be pointed to another object at any time.
  - A reference must be initialized when it is created.
    - Pointers can be initialized at any time.

Reference Variable : Program

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

int main()
{
    int num = 20;

    int &rnum = num;

    rnum = 50;
    cout << "num = " << num << endl ;
    num = 100;
    cout << "rnum = " << rnum << endl ;

return 0;
}
```

```
num = 50
rnum = 100
```

Pass by Reference

- When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function i.e.,
  - When function is working with its own arguments, it is actually working on the original data

Pass by Reference : Program

```
// Swapping using reference variable
#include<iostream>
using namespace std;

void swap (int &, int &);

int main()
{
    int a = 10, b = 20;
    cout<<"Before Swapping:"<<"a="<<a<<"  b="<<b<<endl;
    swap( a, b );
    cout<<"After Swapping:"<<"a="<<a<<"  b="<<b<<endl;
    return 0;
}
void swap (int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}
```

```
Before Swapping:a=10  b=20
After Swapping:a=20  b=10
```

Return by Reference

- A function can return a variable by reference.
- Like the variable alias in passing arguments as reference, the return by reference returns the alias
  - actually the reference which does not need the dereferencing operator
- ***Allows the function to be written on the left hand side of the equality expression***
- For example:

  max(a,b) = 50;

  This returns the reference of the variable that are passed as reference to the function

Return by Reference : Program

```
// Return by Reference
#include<iostream>
using namespace std;

int& max (int &, int &);

int main()
{
    int a = 10, b = 20;
    cout<<"Before Calling:"<<"a="<<a<<"  b="<<b<<endl;
    max( a, b )=50;
    cout<<"After Calling:"<<"a="<<a<<"  b="<<b<<endl;
    return 0;
}
int& max (int& n1, int& n2)
{
    if(n1>n2)
        return n1;
    else
        return n2;
}
```

```
Before Calling:a=10  b=20
After Calling:a=10  b=50
```

Return by Reference : Address

```
// Reference Variable Address
#include<iostream>
using namespace std;
int main()
{
    int num = 20;
    int &rnum = num;
    cout<<"num="<<num<<endl<<"rnum="<<rnum<<endl;
    rnum = 100;
    cout<<"num="<<num<<endl<<"rnum="<<rnum<<endl;
    cout<<"Address of num="<<&num<<endl<<"Address of rnum="<<&rnum<<endl;
    return 0;
}
```

```
num=20
rnum=20
num=100
rnum=100
Address of num=0x28fef8
Address of rnum=0x28fef8
```

```
/** C++ Program to demonstrate working of default argument & Function overloading */
    #include <iostream>
    using namespace std;
    void display(float = 20.5, int = 10);
    void display(float = 30.5);
    int main()
    {
        cout << "No argument passed:\n";
        display();
        cout << "\nFirst argument passed:\n";
        display(50.5);
        cout << "\nBoth argument passed:\n";
        display(10.5, 10);
        return 0;
    }

    void display(float f, int n)
    {
        cout<<endl<<"Float Value="<<f<<"\tInteger Value="<<n<<endl;
    }
     void display(float f)
    {
        cout<<endl<<"Float Value="<<f<<"No integer Value"<<endl;
    }
```

Does this code work??
Discussion session!!

Revision work

Self Study

3.14 Array, Pointer and String

3.15 Structure, Union and Enumeration