LECTURE-34

Exception Handling:

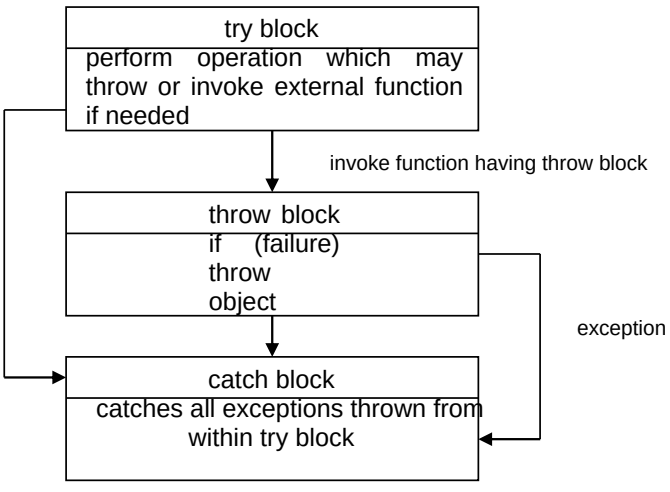
Exception refers to unexpected condition in a program. The unusual conditions could be faults, causing an error which in turn causes the program to fail. The error handling mechanism of c++ is generally referred to as exception handling.

Generally , exceptions are classified into synchronous and asynchronous exceptions.. The exceptions which occur during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data. with in a program is known as synchronous exception.
Example:
errors such as out of range,overflow,underflow and so on.

The exceptions caused by events or faults unrelated to the program and beyond the control of program are asynchronous exceptions.
For example, errors such as keyboard interrupts, hardware malfunctions, disk failure and so on.

exception handling model:

When a program encounters an abnormal situation for which it in not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception. The exception handling mechanism uses three blocks: try, throw and catch. The try block must be followed immediately by a handler, which is a catch block. If an exception is thrown in the try block the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. The relationship of these three exceptions handling constructs called the exception handling model is shown in figure:



throw construct:

The keyword throw is used to raise an exception when an error is generated in the comutation. the throw expression initialize a temporary object of the typeT used in thorw (T arg).
syntax:
throw T;

catch construct:

The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword. The catch handler can also occur immediately after another catch Each handler will only evaluate an exception that matches.
syn:

```
catch(T)
{
// error meassges
}
```

try construct:

The try keyboard defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the prepared to test for the existence of exceptions. If ar exception occurs, the program flow is interrupted.

```
try
{
…
if (failure)
throw T;
}
catch(T)
{
…
}
```

example:

```
#include<iostream.h>
void main()
{
int a,b;
cout<<”enter two numbers:”;
cin>>a>>b;
try
{
if (b= =0)
throw b;
else
cout<a/b;
}
catch(int x)
{
cout<<”2nd operand can’t be 0”;
}
}
```

LECTURE-35

Array reference out of bound:

```
#define max 5
class array
{
private:
int a[max];
public:
int &operator[](int i)
{
if (i<0 || i>=max)
throw i;

else
return a[i];
}
};
void main()
{
array x;
try
{
cout<<"trying to refer a[1]…"
x[1]=3;
cout<<"trying to refer a[13]…"
x[13]=5;
}
catch(int i)
{
cout<<"out of range in array references…";
}
}
```

~~multiple catches in a progra~~m

```
void test(int x)
{
try{
if (x==1)
throw x;
else if (x==-1)
throw 3.4;
else if (x==0)
throw 's';
}
catch (int i)
{
cout<<"caught an integer…";
}
catch (float s)
{
cout<<"caught a float…";
```

```
}
catch (char c)
{
cout<<”caught a
character…”; }}
void main()
{
test(1);
test(-1);
test(0);
}
```

catch all

```
void test(int x)
{
try{
if (x==1)
throw x;
else if (x==-1)
throw 3.4;
else if (x==0)
throw ‘s’;
}
catch (…)
{
cout<<”caught an error…”;
}
```

Module-03:
LECTURE-36

Containership in C++

When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax for the declaration of another class is:

```
Class class_name1
{
_____
_____
};
Class class_name2
{
_____
_____
};
Class class_name3
{
Class_name1 obj1; // object of class_name1
Class_name2 obj2; // object of class_name2
_____-
_____
};
```

```
//Sample Program to demonstrate Containership
#include < iostream.h >
#include < conio.h >
#include < iomanip.h >
#include< stdio.h >
const int len=80;
class employee
{
private:
char name[len];
int number;
public:
void get_data()
{
cout << "\n Enter employee name: ";
cin >> name;
cout << "\n Enter employee number: ";
cin >> number;
}
void put_data()
{
cout << " \n\n Employee name: " << name;
cout << " \n\n Employee number: " << number;
}
};
class manager
{
private:
char dept[len];
int numemp;
employee emp;
public:
void get_data()
{
emp.get_data();
cout << " \n Enter department: ";
cin >> dept;
cout << "\n Enter number of employees: ";
cin >> numemp;
}
void put_data()
{
emp.put_data();
cout << " \n\n Department: " << dept;
cout << " \n\n Number of employees: " << numemp;
}
};
class scientist
{
private:
int pubs,year;
employee emp;
public:
```

```
void get_data()
{
emp.get_data();
cout << " \n Number of publications: ";
cin >> pubs;
cout << " \n Year of publication: ";
cin >> year;
}
void put_data()
{
emp.put_data();
cout << "\n\n Number of publications: " << pubs;
cout << "\n\n Year of publication: "<< year;
}
};
void main()
{
manager m1;
scientist s1;
int ch;
clrscr();
do
{
cout << "\n 1.manager\n 2.scientist\n";
cout << "\n Enter your choice: ";
cin >> ch;
switch(ch)
{
case 1:
cout << "\n Manager data:\n";
m1.get_data();
cout << "\n Manager data:\n";
m1.put_data();
break;
case 2:cout << " \n Scientist data:\n";
s1.get_data();
cout << " \n Scientist data:\n";
s1.put_data();
break;
}
cout << "\n\n To continue Press 1 -> ";
cin >> ch;
}
while(ch==1);
getch();
}
```

Difference between Inheritance and Containership :

Containership: Containership is the phenomenon of using one or more classes within the definition of other class. When a class contains the definition of some other classes, it is referred to as composition, containment or aggregation. The data member of a new class is an object of some other class. Thus the other class is said to be composed of other classes and hence referred to as containership. Composition is often referred to as a “has-a” relationship because the objects of the composite class have objects of the composed class as members.

Inheritance: Inheritance is the phenomenon of deriving a new class from an old one. Inheritance supports code reusability. Additional features can be added to a class by deriving a class from it and then by adding new features to it. Class once written or tested need not be rewritten or redefined. Inheritance is also referred to as specialization or derivation, as one class is inherited or derived from the other. It is also termed as “is-a” relationship because every object of the class being defined is also an object of the inherited class.

LECTURE-37

Template:

Template supports generic programming, which allows developing reusable software components such as functions, classes, etc supporting different data types in a single frame work.
A template in c++ allows the construction of a family of template functions and classes to perform the same operation o different data types. The templates declared for functions are called class templates. They perform appropriate operations depending on the data type of the parameters passed to them.

Function Templates:

A function template specifies how an individual function can be constructed.

```
template <class T>
return type functionnm(T arg1,T arg2)
{
fn body;
}
```

For example:

Input two number and swap their values

```
template <class T>
void swap (T &x,T & y)
{
T z;
z=x;
x=y;
y=z;
}
void main( )
{
char ch1,ch2;
cout<<"enter two characters:";
cin>>ch1>>ch2;
swap(ch1,ch2);
cout<<ch1<<ch2;
int a,b;
cout<<"enter a,b:";
cin>>a>>b;
swap(a,b);
cout<<a<<b;
float p,q;
cout<<"enter p,q:";
cin>>p>>q;
swap(p,q);
cout<<p<<q;
}
```

example 2:

find maxium between two data items.

```
template <class T>
T max(T a,T b)
```

```
{
if (a>b)
return a;
else
return b;
}
void main()
{
char ch1,ch2;
cout<<"enter two characters:";
cin>>ch1>>ch2;
cout<<max(ch1,ch2);
int a,b;
cout<<"enter a,b:";
cin>>a>>b;
cout<<max(a,b);
float p,q;
cout<<"enter p,q:";
cin>>p>>q;
cout<<max(p,q);
}
```

Overloading of function template

```
#include<iostream.h>
template <class T>
void print( T a)
{
cout<<a;
}
template <class T>
void print( T a, int n)
{
int i;
for (i=0;i<n;i++)
cout<<a;
}
void main()
{
print(1);
print(3.4);
print(455,3);
print("hello",3);
}
```

Multiple arguments function template:

find sum of two different numbers

```
template <class T,class U>
T sum(T a,U b)
{
return a+(U)b;
}
void main( )
```

```
{
cout<<sum(4,5.5);
cout<sum(5.4,3);
}
```

LECTURE-38

Class Template

similar to functions, classes can also be declared to operate on different data types. Such classes are class templates. a class template specifies how individual classes can be constructed similar to normal class definition. These classes model a generic class which support similar operations for different data types.

syn:

```
template <class T>
class classnm
{
T member1;
T member2;
…
…
public:
T fun();
…
..
};
```

objects for class template is created like:

```
classnm <datatype> obj;
obj.memberfun();
```

example:

Input n numbers into an array and print the element is ascending order.(array sorting)

```
template <class T>
class array
{
T *a;
int n;
public:
void getdata()
{
int i;
cout<<"enter how many no:";
cin>>n;
a=new T[n];
for (i=0;i<n;i++)
{
cout<<enter a number:";
cin>>a[i];
}
}
void putdata()
{
```

```
for (i=0;i<n;i++)
{
cout<<a[i]<<endl;
}
}
void sort( )
{
T k;
int i,j;
for(i=0;i<n-1;i++)
{
for (j=0;j<n;j++)
{
                    if (a[i]>a[j])
{
                                        k=a[i];
                                        a[i]=a[j];
                                        a[j]=k;
}
}
}
}
};
void main()
{
array <int>x;
x.getdata();
x.sort();
x.putdata();

array <float> y;
y.getdata():
y.sort();
y.putdata();
}
```

LECTURE-39

Virtual destructors:

Just like declaring member functions as virtual, destructors can be declared as virtual, whereas constructors can not be virtual. Virtual Destructors are controlled in the same way as virtual functions. When a derived object pointed to by the base class pointer is deleted, destructor of the derived class as well as destructor of all its base classes are invoked. If destructor is made as non virtual destructor in the base class, only the base class's destructor is invoked when the object is deleted.

```
#icnlude<iostream.h>
#include<string.h>
class father
{
protected:
char *fname;
public:
father(char *name)
{
fname=new char(strlen(name)+1);
strcpy(fname,name);
}
virtual ~father()
{
delete fname;
cout<<"~father is invoked…";
}

virtual void show()

{
cout<<"father name…"<<fname;
}
};

class son: public father

{
protected:
char *s_name;
public:
son(char *fname,char *sname):father(fname)
{
sname=new char[strlen(sname)+1];
strcpy(s_name,sname);
}
~son()
{
delete s_name;
cout<<"~son() is invoked"<<endl;
}
void show()
{
cout<<"father's name"<<fname;
cout<<"son's name:"<<s_name;
```

```
}
};
void main()
{
father *basep;
basep =new father (“mona”);
cout<<”basep points to base object…”
basep->show();
delete basep;
basep=new son(“sona”,”mona”);
cout<<”base points to derived object…”;
basep->show();
delete basep;
}
```

Overloading of >> and << operator

```
#define size 5
class vector
{
int v[size];
public:
vector();
friend vector operator*(int a,vector b);
friend vector operator *(vector b,int a);
friend istream &operator>>(istream &,vector &);
friend ostream &operator<<(ostream &,vector &);
};
vector :: vector()
{
for(int i=0;i<size;i++)
v[i]=0;
}
vector::vector(int *x)
{
for (int i=0;i<size;i++)
v[i]=x[i];
}
vector operator*(int a,vector b)
{
vector c;
for(int i=0;i<size;i++)
c.v[i]=a*b.v[i];
return c;
}
vector operator*(vector b,int a)
{
vector c;
```

```
for(int i=0;i<size;i++)
c.v[i]=a*b.v[i];
return c;
}
istream &operator>>(istream &din,vector &b)
{
for(int i=0;i<size;i++)
din>>b.v[i];
}
ostream &operator<<(ostream &dout,vector &b)
{
for(i=0;i<size;i++)
dout<<a[i];
return dout;
}
int x[size]={2,4,6};
int main()
{
vector m;
vector n=x;
cout<<"enter elements of vector m";
cin>>m;
cout<<m;
vector p,q;
p=2*m;
q=n*2;
cout<<p;
cout<<q;

}
```

LECTURE-40

Managing Console I/O

Introduction

One of the most essential features of interactive programming is its ability to interact with the users through operator console usually comprising keyboard and monitor. Accordingly, every computer language (and compiler) provides standard input/output functions and/or methods to facilitate console operations.

C++ accomplishes input/output operations using concept of stream. A stream is a series of bytes whose value depends on the variable in which it is stored. This way, C++ is able to treat all the input and output operations in a uniform manner. Thus, whether it is reading from a file or from the keyboard, for a C++ program it is simply a stream.

We have used the objects cin and cout (pre-defined in the iostream.h file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types. The >> operator is overloaded in the istream class and « is overloaded in the ostream class. The following is the general format for reading data from the keyboard:

cin >> variable1 >> variable2 >>… …>> variableN;

Where variable1, variable2,.... are valid C++ variable names that have been declared already. This statement will cause the computer to halt the execution and look for input data from the keyboard. The input data for this statement would be:

data1 data2.....dataN

The input data are separated by white spaces and should match the type of variable in the cin list. Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location.

The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type.
For example, consider the following code:

int code;

cin >> code;

Suppose the following data is given as input:

1267E

The operator will read the characters up to 7 and the value 1267 is assigned to code. The character E remains in the input stream and will be input to the next cin statement. The general format of outputting data:
cout << iteml <<item2 << .. ..<< itemN;
The items, item1 through itemN may be variables or constants of any basic types.

The put() and get() Functions

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations. There are two types of get() functions. We can use both get(char*) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char*) version assigns the input character to its argument and the get(void) version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object. For instance, look at the code snippet given below:

```
char c;
cin.get (c); //get a character from keyboard and assign it to c
while (c!= '\n')
{
cout << C; //display the character on screen cin.get (c);
//get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator> >can also be used to read a character but it will skip the white spaces and newline character. The above while loop will not work properly if the statement

```
cin >> c;
```

is used in place of

```
cin.get (c);
```

Try using both of them and compare the results. The get(void) version is used as follows:

```
char c;
c - cin.getl); //cin.get (c) replaced
```

The value returned by the function get() is assigned to the variable c.

The function put(), a member of ostream class, can be used to output a line of text, character by character. For example,

```
cout << put ('x');
```

displays the character x and

```
cout << put (ch) ;
```

displays the value of variable ch.

The variable ch must contain a character value. We can also use a number as an argument to the function put (). For example,

```
cout << put (68) ;
```

displays the character D. This statement will convert the int value 90 to a char value and display the character whose ASCII value is 68,

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c; .
cin.get (c) //read a character
while (c!= '\n')
{
cout<< put(c); //display the character on screen cin.get (c ) ;
}
```

The getline () and write () Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object cin as follows:

cin.getline(line, size);

This function call invokes the function which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size number of characters are read (whichever occurs first). The newline. character is read but not saved. Instead, it is replaced by the null character.

For example; consider the following code:

char name [20] ;
cin.getline(name, 20);

Assume that we have given the following input through the keyboard:

Neeraj good

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

Object Oriented Programming

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

After reading the string/ cin automatically adds the terminating null character to the character array.

Remember, the two blank spaces contained in the string are also taken into account, i.e. between Objects and Oriented and Pro.

We can also read strings using the operator **>>** as follows:

cin >> name;

But remember cin can read strings that do not contain white space. This means that cin can read just one word and not a series of words such as “Neeraj good”.

Formatted Console I/O Operations

C++ supports a number of features that could be used for formatting the output. These features include:

- ios class functions and flags.
- Manipulators.
- User-defined output functions.

The ios class contains a large number of member functions that could be used to format the output in a number of ways. The most important ones among them are listed below.

Table 10.1

Function	Task
width()	To specify the required field size for displaying an output value
Precision()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field.
self()	To specify format flags that can control the form of output display (such as Left-justification and right-justification).
Unself()	To clear the flags specified.

Manipulators are special functions that can be included in these statements to alter the format parameters of a stream. The table given below shows some important! manipulator functions that are frequently used. To access these manipulators, the file iomanip.h should be included in the program.

Table 10.2

Manipulator	Equivalent Ios function
setw()	width()
Setprecision()	Precision()
Setfill()	fill()
setiosflags()	self()
Resetiosflags()	Unself()

In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats.

Student Activity

1. What is a stream?
2. Define put ( ) and get ( ) functions
3. What is the difference between getline ( ) and get ( ) functions?
4. Define write ( ) functions.
5. What are manipulators?

Streams

C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the system supplies an interface to the programmer that is independent of the actual device being accessed, This interface is known as stream.

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the output stream. In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream.

The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. As mentioned earlier, a stream acts as an interface between the program and the input/output device.

Therefore, a C++ program handles data (input or output) independent of the devices used. C++ contains several pre-defined streams that are automatically opened when a program begins its execution. They include cin and cout which have been used very often in our earlier programs. We know that cin represents the input stream connected to the standard input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen). Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, if necessary.

I/O Operations

Input and Output statements of computer languages are used to provide commu-nications between the user and the program. In most of the computer languages, input and output are done

through statements. But in C++, these operations are carried out through its built-in functions. The I/O functions are designed in header files like fstream.h, iostream.h etc.

Through these functions, data can be read from or written to files or standard input/output devices like keyboard and VDU. This execution of a program can be interrupted by input/output calls. Hence the data can be entered or output can be retrieved during execution.

The file, stream classes support a number of member functions for performing the input and output operations on files. One pair of functions, pot( ) and get( ), are designed for handling a single character at a time. Another pair of functions,

write() and readQ, are designed to write and read blocks of binary data.

put( ) and get( )Functions

The function put( ) writes a single character to the associated stream. Similarly, the function get ( ) reads a single character from the, associated stream. The program, requests for a string. On receiving the string, the program writes it, character, by character, to the file using the pot() function in a for loop. Note that the length of the string is used to terminate the for loop.

C++ provides a number of useful predefined stream classes for console input/output operations. Some of the C++ the predefined stream objects are listed below.

cin This is the name of standard input stream, usually keyboard. The corresponding name in C is stdin.

cout This is the name of standard output stream, usually screen of the monitor. The corresponding name in C is stdout.

cerr This is the name of standard error output stream, usually screen of the monitor. The corresponding name in C is stderr.

clog This is another version of cerr. It provides buffer to collect errors. C does not have a keyword equivalent to this.

Keywords

put (): A member of ostream class, could be used to output a line of text, character by character.

Get (): A member of istream class, used to input a single character at a line.

Getline (): The get line ( ) function reads a whole line of -text that ends with a new line character. This function could be invoked by using the object cin.

Manipulators: Special functions that can be included in console I/O statements to alter the format-parameters of a stream

Streams: C++ is designed to work with a wide variety of devices including, disks and take drives.

Although each device is very different the system suppliers an interface to the programmer that is independent of the actual device accessed. This interface is known as stream.

Output stream: The source stream that provides data to the program is called the.output stream.

In their default roles, these streams are tied up with the keyboard and screen of the monitor as describe above. However, you can redirect them from and to other devices and files.

LECTURE-41

Namespaces :

Scopes

Named entities, such as variables, functions, and compound types need to be declared before being used in C++. The point in the program where this declaration happens influences its visibility:

An entity declared outside any block has *global scope*, meaning that its name is valid anywhere in the code. While an entity declared within a block, such as a function or a selective statement, has *block scope*, and is only visible within the specific block in which it is declared, but not outside it.

Variables with block scope are known as *local variables*.

For example, a variable declared in the body of a function is a *local variable* that extends until the end of the the function (i.e., until the brace that closes the function definition), but not outside it.

```
int some_function ()
{
int bar; // local variable
bar = 0;
}
int other_function ()
{
foo = 1; // ok: foo is a global variable
        bar = 2; // wrong: bar is not visible from this function
}
```

In each scope, a name can only represent one entity. For example, there cannot be two variables with the same name in the same scope:

```
int some_function ()
{
int x;
x = 0;
        double x; // wrong: name already used in this scope
x = 0.0;
}
```

The visibility of an entity with *block scope* extends until the end of the block, including inner blocks. Nevertheless, an inner block, because it is a different block, can re-utilize a name existing in an outer scope to refer to a different entity; in this case, the name will refer to a different entity only within the inner block, hiding the entity it names outside. While outside it, it will still refer to the original entity. For example:

```
// inner block scopes
#include <iostream>
```

```
using namespace std;

int main () {
int x = 10;
int y = 20;
{
int x; // ok, inner scope.
x = 50; // sets value to inner x
                    y = 50; // sets value to (outer) y
cout << "inner block:\n";
cout << "x: " << x << '\n';
cout << "y: " << y << '\n';
}
cout << "outer block:\n";
cout << "x: " << x << '\n';
cout << "y: " << y << '\n';
return 0;
}
```

output:

```
inner block: x:
50
y: 50
outer block: x:
10
y: 50
```

Note that is not hidden in the inner block, and thus accessing y still accesses the outer variable.

Variables declared in declarations that introduce a block, such as function parameters and variables declared in loops and conditions (such as those declared on a for or an if) are local to the block they introduce.

Namespaces

Only one entity can exist with a particular name in a particular scope. This is seldom a problem for local names, since blocks tend to be relatively short, and names have particular purposes within them, such as naming a counter variable, an argument, etc...

But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

The syntax to declare a namespaces is:

```
namespace identifier {
named_entities
}
```

Where identifier is any valid identifier and named_entities is the set of variables, types and functions that are included within the namespace. For example:

```
namespace
myNamespace {
int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace.

These variables can be accessed from within their namespace normally, with their identifier (either a or b), but if accessed from outside the myNamespace namespace they have to be properly qualified with the scope operator ::. For example, to access the previous variables from outside myNamespace they should be qualified like:

```
1 myNamespace::a
2
myNamespace::b
```

Namespaces are particularly useful to avoid name collisions. For example:

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

namespace foo
{
int value() { return 5; }
}
namespace bar
{
const double pi = 3.1416;
        double value() { return 2*pi; }
}

int main () {
        cout << foo::value() << '\n';
        cout << bar::value() << '\n';
cout << bar::pi << '\n';
return 0;
}
```

output:

```
5
6.2832
3.1416
```

In this case, there are two functions with the same name: value. One is defined within the namespace foo, and the other one in bar. No redefinition errors happen thanks to namespaces. Notice also how pi is accessed in an unqualified manner from within namespace bar (just as pi), while it is again accessed in main, but here it needs to be qualified as bar::pi.

Namespaces can be split: Two segments of a code can be declared in the same namespace:

```
1 namespace foo { int a; }
```

```
2 namespace bar { int b; } 3
namespace foo { int c; }
```

This declares three variables: a and c are in namespace foo, while b is in namespace bar.

Namespaces can even extend across different translation units (i.e., across different files of source code).

using

The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example:

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

namespace first
{
int x = 5;
int y = 10;
}

namespace second
{
double x = 3.1416;
double y = 2.7183;
}

int main () {
using first::x;
using second::y;
cout << x << '\n';
cout << y << '\n';
cout << first::y << '\n';
          cout << second::x << '\n';
return 0;
}
```

Output:

```
5
2.7183
10
3.1416
```

Notice how in main, the variable x (without any name qualifier) refers to first::x, whereas y refers to second::y, just as specified by the using declarations. The variables first::y and second::x can still be accessed, but require fully qualified names.

using

The keyword can also be used as a directive to introduce an entire namespace:

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

namespace first
{
int x = 5;
int y = 10;
```

```
}

namespace second
{
double x = 3.1416;
double y = 2.7183;
}

int main () {
using namespace first;
cout << x << '\n';
cout << y << '\n';
            cout << second::x << '\n';
           cout << second::y << '\n';
return 0;
}
```

output:

```
5
10
3.1416
2.7183
```

In this case, by declaring that we were using namespace first, all direct uses of x and y without name qualifiers were also looked up in namespace first.

using and using namespace have validity only in the same block in which they are stated or in the entire source code file if they are used directly in the global scope. For example, it would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks:

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

namespace first
{
int x = 5;
}

namespace second
{
double x = 3.1416;
}

int main () {
{
using namespace first;
cout << x << '\n';
}
{
      using namespace second;
cout << x << '\n';
}
return 0;
}
```

output:

```
5
3.1416
```

Namespace aliasing

Existing namespaces can be aliased with new names, with the following syntax:

```
namespace new_name = current_name;
```

The std namespace

All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the namespace. Most examples in these tutorials, in fact, include the following line:

```
using namespace std;
```

This introduces direct visibility of all the names of the std namespace into the code. This is done in these tutorials to facilitate comprehension and shorten the length of the examples, but many programmers prefer to qualify each of the elements of the standard library used in their programs. For example, instead of:

```
cout << "Hello world!";
```

It is common to instead see:

```
std::cout << "Hello world!";
```

Whether the elements in the std namespace are introduced with using declarations or are fully qualified on every use does not change the behavior or efficiency of the resulting program in any way. It is mostly a matter of style preference, although for projects mixing libraries, explicit qualification tends to be preferred.

Storage classes

The storage for variables with *global* or *namespace scope* is allocated for the entire duration of the program. This is known as *static storage*, and it contrasts with the storage for *local variables* (those declared within a block). These use what is known as automatic storage. The storage for local variables is only available during the block in which they are declared; after that, that same storage may be used for a local variable of some other function, or used otherwise.

But there is another substantial difference between variables with *static storage* and variables with *automatic storage*:
- Variables with *static storage* (such as global variables) that are not explicitly initialized are automatically initialized to zeroes.
- Variables with *automatic storage* (such as local variables) that are not explicitly initialized are left uninitialized, and thus have an undetermined value.

For example:

```
// static vs automatic storage
#include <iostream>
```

```
using namespace std;

int x;

int main ()
{
int y;
        cout << x << '\n';
        cout << y << '\n';
return 0;
}
```

Output:

```
0
4285838
```

The actual output may vary, but only the value of x is guaranteed to be zero; y can actually contain just about any value (including zero).

Lecture-42:

New & Delete Operators

Dynamic memory allocation means creating memory at runtime. For example, when we declare an array, we must provide size of array in our source code to allocate memory at compile time.
But if we need to allocate memory at runtime me must use new operator followed by data type. If we need to allocate memory for more than one element, we must provide total number of elements required in square bracket[ ]. It will return the address of first byte of memory.

Syntax of new operator

```
ptr = new data-type;
//allocte memory for one element

ptr = new data-type [ size ];
//allocte memory for fixed number of element
```

Delete operator is used to deallocate the memory created by new operator at run-time. Once the memory is no longer needed it should by freed so that the memory becomes available again for other request of dynamic memory.

Syntax of delete operator

```
delete ptr;
//deallocte memory for one element

delete[] ptr;
//deallocte memory for array
```

Example of c++ new and delete operator

```
#include<iostream.h>
#include<conio.h>
```

```
void
main() {

        int size,i;
        int *ptr;

        cout<<"\n\tEnter size of Array :
        "; cin>>size;

 ptr = new int[size];
     //Creating memory at run-time and return first byte of address to ptr.
for(i=0;i<5;i++) //Input arrray from user.
{
cout<<"\nEnter any number : ";
cin>>ptr[i];
}
for(i=0;i<5;i++) //Output arrray to console.
cout<<ptr[i]<<", ";
delete[] ptr;
//deallocating all the memory created by new operator

}
```

Output :

```
Enter size of Array :
5 Enter any number
: 78 Enter any
number : 45 Enter
any number : 12
Enter any number :
89 Enter any
number : 56
78, 45, 12, 89, 56,
```