1. Objective

After going through this unit you will understand:

- What is a template and its need in C++?
- What is overloading of template functions?
- How to define a class template?
- Exception handling concept and keywords used in it.
- How to re-throw an exception.
- Usage of multiple catch statements.
- How to handle multiple exceptions in C++.
- Exception and Operator overloading
- Exception in Constructors and Destructors.

1. Introduction

This unit deals with two concept one is template and another is exception handling. The template section contains template definition and other factor associated with it. In this part it is that it allows working of a single class or function with various data types. Normal function template can access the template without being part of any class. Template function can be overloaded.

Second part of unit is about exception handling. It includes the keywords used in exception handling. Re-throwing an exception means forwarding the exception of a catch block to be handled by some other exception handler. Multiple catch statements can also exist under one try block. Multiple exceptions may also be handled by catch.

The rest of the chapter is organized as follows. Section 10.2 describes the need of templates. Section 10.3 explains what a class template is. Section 10.4 discusses normal function template. Section 10.5 is about overloading of templates. Sections 10.6 describes member function templates. Section 10.7 starts the exception handling details. It includes the keywords used in it. Section 10.8 discusses about re-throwing an exception. Section 10.9 handles multiple catch statements and section 10.10 is for catching multiple exceptions. Section 10.11 shows Exception in Constructors and Destructors. Section 10.12 and section 10.13 is of exception and operator overloading and exception and inheritance respectively. And finally section 10.14 concludes the chapter with its summary.

10.2 Need of Templates

A template is a technique that allows a single function or class to work with different data types. Using a template, we can create a single function that can process any type of data; that is, the formal arguments of a template function are of template (generic) type. They can accept data of any type, such as int, float, and long. Thus, a single function can be used to accept values of a different data type. Figures (Working of non-

template function) and (Working of template function) clear the difference between the non-template function and template function. Usually, we overload a function when we need to handle different data types. This approach increases the program size. The disadvantages are that not only the program length is increased but also more local variables are created in the memory. A template safely overcomes all the limitations occurring during the overloading function and allows better flexibility to the program. The portability provided by a template mechanism can be extended to classes. The following sections describe the use of templates in different concepts of C++ programming. Let us understand the process of the declaration of a template.

10.3 Class Templates

In order to declare a class of template type, the following syntax is used:
Template Declaration template < class T> class name_of_class
{
// class data member and function
}

The first statement template < class T> tells the compiler that the following class declaration can use the template data type. The T is a variable of template type that can be used in the class to define a variable of template type. Both template and class are keywords. The <> (angle bracket) is used to declare the variables of template type that can be used inside the class to define the variables of template type. One or more variables can be declared separated by a comma. Templates cannot be declared inside classes or functions. They should be global and should not be local.

T k;

where, k is the variable of template type. Most of the authors use T for defining a template; instead of T, we can use any alphabet.

A program to show values of different data types using overloaded constructor.

#include<iostream.h> #include<conio.h> class data
{
public:
data(char c)
{cout<<"\n"<< " c="<<c <<" Size in bytes:"<<size of(c);} data(int c)
{cout<<"\n"<< " c="<<c <<" Size in bytes:"<<size of(c);} data(double c)
{cout<<"\n"<< " c="<<c <<" Size in bytes:"<<size of(c);}
};

```
int main()
{
clrscr();
data h('A'); // passes character type data  data i(100); // passes integer type data  data j(68.2); // passes
double type data  return 0;
}
```

OUTPUT

```
c = A Size in bytes :1
c = 100 Size in bytes :2  c = 68.2 Size in bytes :8
```

Explanation: In the above program, the class data contains three overloaded one-argument constructors. The constructor is overloaded for char, int, and double type. In function main(), three objects h, i, and j are created, and the values passed are of different types. The values passed are char, int, and double type. The compiler invokes different constructors for various data types. Here, in order to manipulate different data types, we require to overload the constructor; that is, defining a separate function for each non-compatible data type. This approach has the following disadvantages:

1. Re-defining the functions separately for each data type increases the source code and requires more time.
2. The program size is increased. Hence, more disk space is occupied.
3. If the function contains a bug, it should be corrected in every function.
4. Figure 10.1 shows you the working of a program.

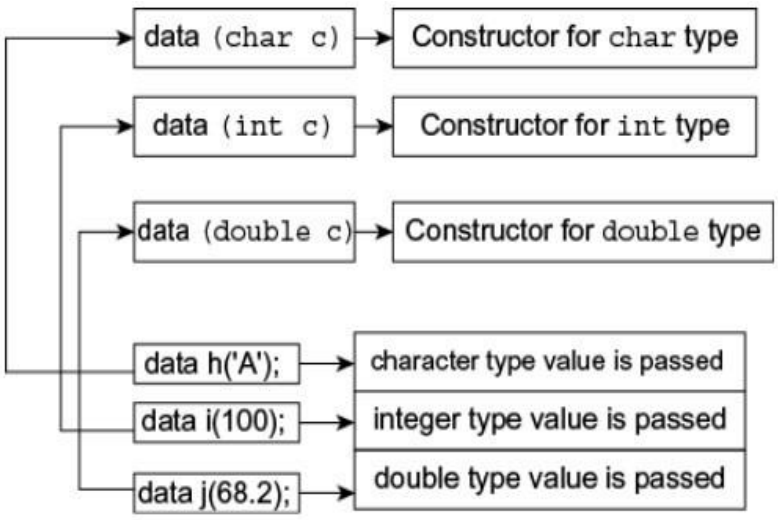

Figure 10.1: Working of non-template function

From the above program, it is clear that for each data type we need to define a separate constructor function.

According to data, type of argument passed respective constructor is invoked. C++ provides templates to overcome such a problem and helps the programmer develop a generic program. The same program is illustrated with the template as follows:

A program to show values of different data types using constructor and template.

```
#include<iostream.h>  #include<conio.h>  template<class T>  class data
{
public:
data (T c)
{
cout<<"\n"<< " c="<<c <<" Size in bytes:"<<size of(c);
}
};
int main()
{
clrscr();
data <char> h('A');  data <int> i(100);  data <float> j(3.12);
return 0;
}
```

OUTPUT

c = A Size in bytes :1
c = 100 Size in bytes :2 c = 3.12 Size in bytes :4

Explanation: In the above program, the constructor contains a variable of template T. The template class variable can hold values of any data type. While declaring an object, the data type name is given before the object. The variable of template type can accept the values of any data type. Thus, the constructor displays the actual values passed. The template variable c can hold the values of any data type. The value and space in bytes required by these variables are displayed at the output. The size of data type changes according to the data types used in the program. Figure 10.2 shows the working of the program.

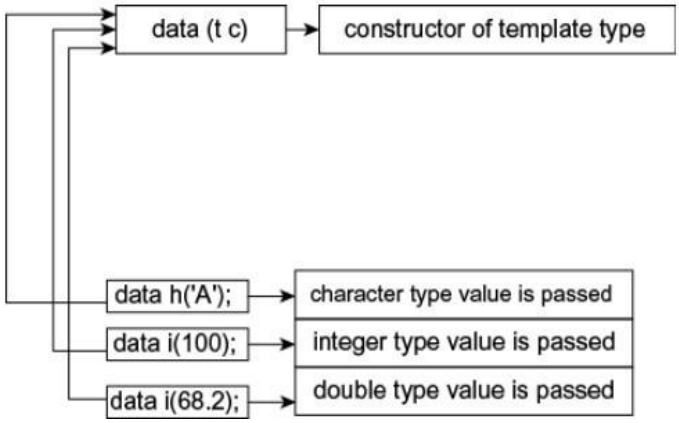

Figure 10.2: Working of template function

In the above program, different values are passed using constructors, but for all data types, the same template function is used

10.4 Normal Function Templates

In the previous section, we observed how to make a template class. In the same manner, a normal function (not a member function) can also use template arguments. The difference between normal and member functions is that normal functions are defined outside the class. They are not members of any class and, hence, can be invoked directly without using the object of a dot operator. The member functions are class members, and they can be invoked using the object of the class they belong to. The declaration of template member functions is described later in this chapter. In C++, normal functions are commonly used as in C. However, the user who wants to use C++ as better C can utilize this concept. The declaration of a normal template function can be done in the following manner:

Normal Template Function Declaration template < class T>

retuntype function_name (arguments)
{
// code
}

The following program shows the practical working of template function:

A program to define normal template function.

#include<iostream.h> #include<conio.h> template<class T> void show (T x)
{cout<<"\n x="<<x ;}

```
void main()
{
clrscr();
char c='A';  int i=65;
double d=65.254;  show(c);
show(i);
show(d);
}
```

OUTPUT

```
x=A
x=65  x=65.254
```

Explanation: Before the body of the function show(), the template argument T is declared. The function show() has one argument x of template type. As explained earlier, the template type variable can accept all types of data. Thus, the normal function show can be used to display values of different data types. In main function, the show() functions are invoked with char, int, and double type of values being passed. The same is displayed in the output.

You are now familiar with utilities of templates. One more point to remember is that when we declare a class template, we can define the class data member of the template type as well as the member function of the class can also use the template member. For making a member function of template type, no separate declaration is needed. The following program explains the above point:

A program to define data members of template type.

```
#include<iostream.h>  #include<conio.h>  template<class T>  class data
{
T x;  public:
data (T u) {x=u;}  void show (T y)
{
cout<<" x="<<x;  cout<<" y=" <<y<<"\n";
}
};
int main()
{
```

```
clrscr();
data <char> c('B');  data <int> i(100);
data <double> d(48.25);  c.show('A');  i.show(65);
d.show(68.25);  return 0;
}
```

OUTPUT

```
x=B y=A  x=100 y=65
x=48.25 y=68.25
```

Explanation: In this program, before the declaration of class data, template <class T> is declared. This declaration allows the entire class, including member function and data member, to use the template-type argument. We have declared data member x of template type. In addition, the one-argument constructor and member function show() also have one formal argument of template type.

A program to create square() function using template.

```
#include<iostream.h>  #include<conio.h>  template <class S>  class sqr
{
public:
sqr (S c)
{
cout<<"\n"<< " c="<<c*c;
}
};
int main()
{
clrscr();
sqr <int> i(25);
sqr <float> f(15.2);  return 0;
}
```

OUTPUT

```
c = 625
c = 231.039993
```

Explanation: In the above program, the class sqr is declared. It contains a constructor with one argument of template type. In main() function, the object i indicates int type, and f indicates float type. The objects i and f invoke the constructor sqr() with values 25 and 15.2, respectively. The constructor displays the squares of these numbers.

Working of Function Templates

In the last few examples, we have learned how to write a function template that works with all data types. After compilation, the compiler cannot guess with which type of data the template function will work. When the template function is called at that moment, from the type of argument passed to the template function, the compiler identifies the data type. Then, every argument of template type is replaced with the identified data type; this process is called instantiating. Thus, according to different data types, respective versions of the template function are created. Though the template function is compatible for all data types, it will not save any memory. When template functions are used, four versions of functions can be used. The data types int, char, float, and double are created. The programmer need not write separate functions for each data type.

10.5 Overloading of Template Functions

A template function also supports the overloading mechanism. It can be overloaded by a normal function or a template function. While invoking these functions, an error occurs if no accurate match is met. No implicit conversion is carried out in the parameters of template functions. The compiler observes the following rules for choosing an appropriate function when the program contains overloaded functions:

1. Searches for an accurate match of functions; if found, it is invoked
2. Searches for a template function through which a function that can be invoked with an accurate match can be generated; if found, it is invoked
3. Attempts a normal overloading declaration for the function
4. In case no match is found, an error will be reported

A program to overload a template function.

```
#include<iostream.h>  #include<conio.h>  template <class A>  void show(A c)
{
cout<<"\n Template variable c="<<c;
}
void show (int f)
{
cout<<"\n Integer variable f="<<f;
}
int main()
{
```

```
clrscr(); show('C'); show(50);
show(50.25); return 0;
}
```

OUTPUT

Template variable c = C Integer variable f = 50

Template variable c = 50.25

Explanation: In the above program, the function show() is overloaded. One version contains template arguments, and the other version contains integer variables. In main(), the show() function is invoked thrice with char, int, and float values that are passed. The first call executes the template version of the function show(), the second call executes the integer version of the function show(), and the third call again invokes the template version of the function show(). Thus, in the manner described above, template functions are overloaded.

10.6 Member Function Templates

In the previous example, the template functions defined were inline, that is, they were defined inside the class. It is also possible to define them outside the class. While defining them outside, the function template should define the function, and the template classes are parameterized by the type argument.

A program to define definition of member function template outside the class and invoke the function.

```
#include<iostream.h> #include<conio.h> template<class T> class data
{
public:
data (T c);
};
template<class T> data<T>::data (T c)
{
cout<<"\n"<< " c="<<c;
}
int main()
{
clrscr();
data <char> h('A');
```

data <int> i(100); data <float> j(3.12); return 0;
}

OUTPUT

c = A
c = 100
c = 3.12

Explanation: In the above program, the constructor is defined outside the class. In such a case, the member function should be preceded by the template name as per the following statements:

Statements of program

The first line defines the template, and the second line indicates the template class type T.

EXCEPTIONAL HANDLING

10.7 The keywords *try, throw and catch*

The exception-handling technique passes the control of a program from a location of exception in a program to an exceptionhandler routine linked with the try block. An exception-handler routine can only be called by the throw statement.

a) try: The try keyword is followed by a series of statements enclosed in curly braces.

Syntax of try statement try
{
statement 1;
statement 2;
}

b) throw: The function of the throw statement is to send the exception found. The declaration of the throw statement is as follows:

Syntax of throw statement throw (excep);

throw excep;
throw // re-throwing of an exception

The argument excep is allowed to be of any type, and it may be a constant. The catch block associated with the try block catches the exception thrown. The control is transferred from the try block to the catch block. The throw statement can be placed in a function or is a nested loop, but it should be in the try block. After throwing the exception, control passes to the catch statement.

c) catch: Similar to the try block, the catch block also contains a series of statements enclosed in curly braces. It also contains an argument of an exception type in parentheses.

Syntax of catch statement try

```
{
Statement 1;
Statement 2;
}
catch ( argument)
{
statement 3; // Action to be taken
}
```

When an exception is found, the catch block is executed. The catch statement contains an argument of exception type, and it is optional. When an argument is declared, the argument can be used in the catch block. After the execution of the catch block, the statements inside the blocks are executed. In case no exception is caught, the catch block is ignored, and if a mismatch is found, the program is terminated.

Guidelines for Exception Handling

The C++ exception-handling mechanism provides three keywords; they are try, throw, and catch. The keyword try is used at the starting of the exception. The throw block is present inside the try block. Immediately after the try block, the catch block is present. Figure 10.3 shows the try, catch, and throw statements.

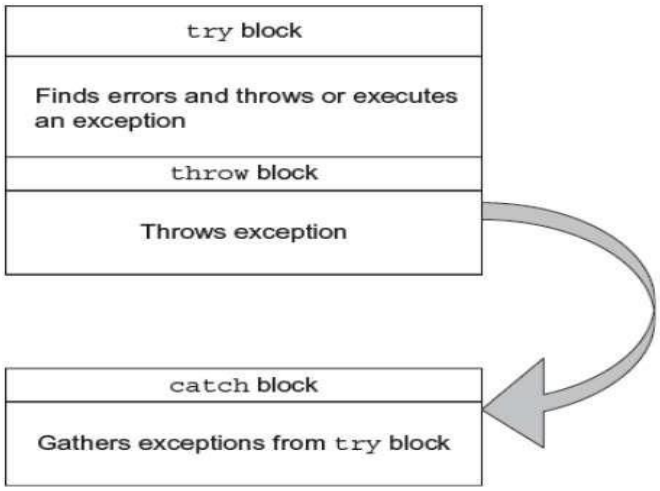

Figure 10.3: Exception Mechanism

As soon as an exception is found, the throw statement inside the try block throws an exception (a message for the catch block that an error has occurred in the try block statements). Only errors occurring inside the try block are used to throw exceptions. The catch block receives the exception that is sent by the throw block. The general form of the statement is as per Figure 10.4.

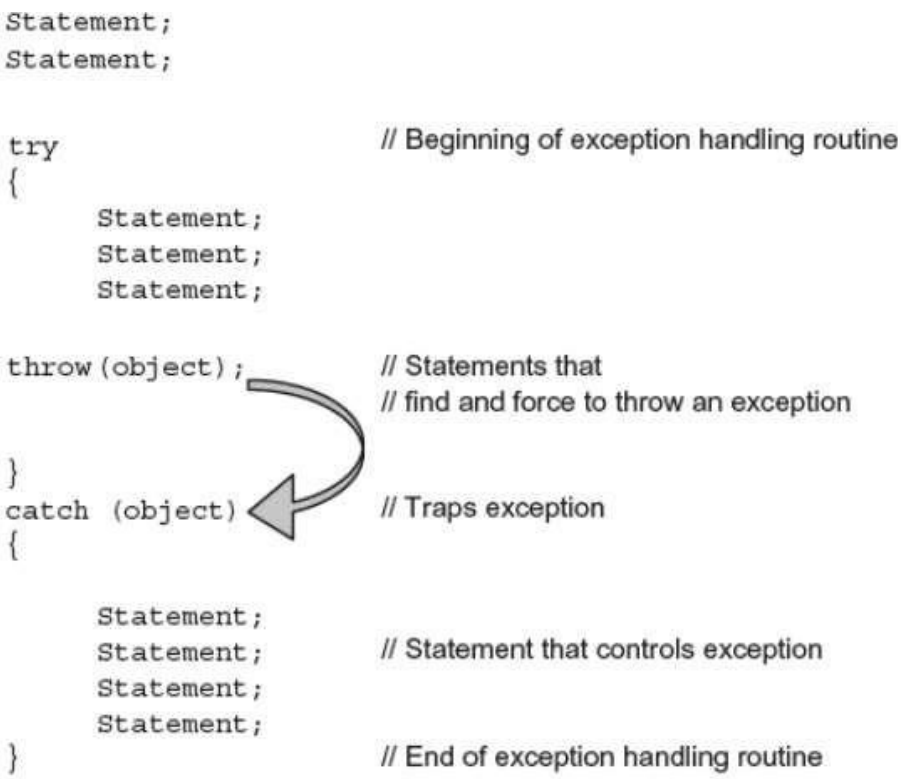

Figure 10.4: try, throw and catch blocks

When the try block passes an exception using the throw statement, the control of the program passes to the catch block. The data type used by throw and catch statements should be same; otherwise, the program is aborted using the abort() function, which is executed implicitly by the compiler. When no error is found and no exception is thrown, in such a situation, the catch block is disregarded, and the statement after the catch block is executed.

A program to throw exception when j=1 otherwise perform the subtraction of x and y.

```
#include<iostream.h>
#include<conio.h>  int main()
{
int x,y;
cout<<"\n Enter values of x and y\n";
```

```
cin>>x>>y;  int j;
j=x>y ? 0 :1;  try
{
if (j==0)
{ cout<<"Subtraction (x-y)="<<x-y<<"\n"; }  else { throw(j); }
}
catch (int k)
{
cout<<"Exception caught : j ="<<j <<"\n";
}
return 0;
}
}
```

OUTPUT

```
Enter values of x and y  7 8
Exception caught : j = 1  Enter values of x and y
4 8
Exception caught : j = 1
```

Explanation: In the above program, the values of x and y are entered. The conditional operator tests the two numbers; if x is greater than y, zero is assigned to j; otherwise, it is one. In the try block, the if condition checks the value of j; the subtraction is carried out when j is zero; otherwise, the else block throws the exception. The catch statement catches the exception thrown. The output of the program is shown above.

A program to define function that generates exception.

```
#include<iostream.h>  void sqr()
{
int s;
cout<<"\n Enter a number:";  cin>>s;
if (s>0)
{
cout<<"Square="<<s*s;
}
else
{
```

```
throw (s);
}
}
int main()
{
try
{
sqr();
sqr();
}
catch (int j)
{
cout<<"\n Caught the exception \n";
}
return 0;
}
```

OUTPUT

```
Enter a number : 5  Square = 25
Enter a number : 0  Caught the exception
```

Explanation: In the above program, the function sqr() is defined. The function sqr() reads an integer through the keyboard and displays its square. Before calculating the square, the if statement checks the number. If it is greater than zero, then the if block calculates the square and displays it; otherwise, the else block throws the exception. In the function main(), in the try block, the function is called twice. In case an exception is thrown from the function sqr(), the catch statement catches it, and the catch block is executed. This happens only when the user enters the number 0 or less than zero. If the user enters 0 in the first call, then the exception is thrown, and the catch block is executed. The compiler ignores the next statement of the try block. Once the control skips from the try block, it never comes back to execute the remaining statements. Thus, in this program if in the first call the user enters 0, then the second call of the function sqr() is not taken into account.

Following points should also be kept in mind at time of exception Handling.

1. It is not essential for the throw statement to appear in the try block as shown in Figure 10.5 in order to throw an exception. The throw statement can be placed in any function, if the function is to be invoked through the try block.

```
try                          Show()
{                            {
 statement;                   statement;
 show (variable)              throw (object);
 statement;                  }
}
catch (object)
{
 statement;
}
```

Figure 10.5: throw statement out of try block

As shown in Figure 19.3, the throw statement is present in the show() function. The show() function is invoked inside the try block. Thus, exception handling can also be defined in the manner described above.

2. When an exception that is not specified is thrown, it is known as an unexpected exception.
3. In case an exception is thrown before the complete execution of a constructor, the destructor for that object will not be executed.
4. As soon as an exception is thrown, the compiler searches nearby handlers (catch blocks). After finding a match, it will be executed.
5. Overuse of exception handling increases the program size. So, apply it whenever most necessary. Incorrect use of exception handling is not consistent and generates bugs in the program. It is hard to debug such bugs.

10.8 Re-throwing Exceptions

It is also possible to again pass the exception received to another exception handler; that is, an exception is thrown from the catch block; this is known as the re-throwing exception. The following statement accomplishes this:

throw;

The above throw statement is used without any argument. This statement throws the exception caught to the next try catch statement. The following program illustrates this:

A program to re-throw an exception. #include<iostream.h>

```
void sub( int j, int k)
{
cout<<“inside function sub()\n”;  try
{
```

```
if (j==0)  throw j;  else
cout<<“Subtraction=”<<j-k<<“\n”;
}
catch (int)
{
cout<<“Caught Null value \n”;  throw;
}
cout<<“** End of sub() ***\n\n”;
}
int main()
{
cout<<“\n inside function main()\n”;  try
sub (8,5);
sub (0,8);
}
catch (int)
{
cout<<“caught null inside main() \n”;
}
cout<<“end of function main() \n”;  return 0;
}
```

OUTPUT

```
inside function main()  inside function sub()  Subtraction = 3
** End of sub() ***  inside function sub()  Caught Null value
caught null inside main()  end of function main()
```

Explanation: In the above program, two try blocks are defined. One is defined in the function sub(), and the other is defined in the function main(). The sub() function has two integer arguments. When the sub() function is invoked, two integer values are sent to this function. Theif statement in the try block of the sub() function checks whether the value of the first variable, that is, j is zero or non-zero. If it is non-zero, subtraction is carried out; otherwise, the throw statement throws an exception. The catch block inside the function sub() collects this exception and again throws the exception using the throw statement. Here, the

throw statement is used without any argument. The catch block of the main function catches there-thrown exception.

10.9 Multiple catch Statements

We can also define multiple catch blocks; in the try block, such programs also contain multiple throw statements based on certain conditions. The format of multiple catch statements is as follows:

```
try
{
 // try section
}
catch (object1)
{
 // catch section1
}
catch (object2)
{
 // catch section2
}
. . . . . .
. . . . . .
catch (type n object)
{
 // catch section-n
}
```

As soon as an exception is thrown, the compiler searches for an appropriate matching catch block. The matching catch block is executed, and control passes to the successive statement after the last catch block. In case no match is found, the program is terminated. In a multiple catch statement, if objects of many catch statements are similar to the type of an exception, in such a situation, the first catch block that matches is executed.

A program to throw multiple exceptions and define multiple catch statement.

```
#include<iostream.h>  void num (int k)
{
try
{
if (k==0) throw k;  else
if (k>0) throw 'P';  else
if (k<0) throw .0;  cout<<"*** try block ***\n";
}
catch(char g)
```

```
{
cout<<“Caught a positive value \n”;
}
catch (int j)
{
cout<<“caught an null value \n”;
}
catch (double f)
{
cout<<“Caught a Negative value \n”;
}
cout<<“*** try catch ***\n \n”;
}
int main()
{
cout<<“Demo of Multiple catches\n”;  num(0);
num(5);
num(-1);  return 0;
}
```

OUTPUT

```
Demo of Multiple catches  caught an null value
*** try catch ***  Caught a positive value
*** try catch ***  Caught a Negative value
*** try catch ***
```

Explanation: In the above program, the function num() contains the try block with multiple catch blocks. In the function main(), the user-defined function num() is invoked with one argument. The if statement within the try block checks the number to see whether it is positive, negative, or zero. According to this, an exception is thrown, and the respective catch block is executed. Here, in the throw statement, objects of different data types such as int, char, and double are used to avoid ambiguity

10.10 Catching Multiple Exceptions

It is also possible to define a single or default catch block from one or more exceptions of different types. In such a situation, a single catch block is used to catch the exceptions thrown by the multiple throw statements.

~~catch~~

```
{
        // Statements for Handling
        // all Exceptions
}
```

A program to catch multiple exceptions.

```
#include<iostream.h>  void num (int k)
{
try
{
if (k==0) throw k;  else
if (k>0) throw 'P';  else
if (k<0) throw .0;  cout<<"*** try block ***\n";
}
catch
{
cout<<"\n Caught an exception\n";
}
}
int main()
{
num(0);
num(5);
num(-1);  return 0;
}
```

OUTPUT

Caught an exception Caught an exception Caught an exception

Explanation: The above program is similar to the previous one. Here, only one difference is observed and that is, instead of multiple catch blocks, a single catch block is defined. For all the exceptions thrown, the same catch block is executed. It is a generic type of catch block. The statement catch catches all the exceptions thrown.

10.11 Exception in Constructors and Destructors

A copy constructor is called in exception handling when an exception is thrown fro the try block using the throw statement. The copy constructor mechanism is applied to initialize the temporary object. In addition, destructors are also executed to destroy the object. If an exception is thrown from the constructor, destructors are called only for those objects that are completely constructed.

A program to use exception handling with constructor and destructor.

```
#include<iostream.h>  #include<process.h>  class number
{
float x;  public :
number (float);  number() {};
~number()
{
cout<<"\n In destructor";
}
void operator ++ (int) // postfix notation
{ x++; }
void operator --() // prefix notation
{
--x; }
void show()
{
cout<<"\n x="<<x;
}
};
number :: number ( float k)
{
if (k==0)
throw number();  else
x=k;
}
void main()
{
try
{
number N(2.4);
cout<<"\n Before Increasing:";  N.show();
cout<<"\n After Increasing:";  N++; // postfix increment
```

```
N.show();
cout<<"\n After Decrementation:";
--N; // prefix decrement  N.show();
number N1(0);
}
catch (number)
{
cout<<"\n invalid number";  exit(1);
}
}
```

OUTPUT

```
Before Increasing:
x=2.4
After Increasing:
x=3.4
After Decrementation:
x=2.4
In destructor  In destructor  invalid number
```

Explanation: In the above program, the operators ++ and – are overloaded. The class number() has two constructors and one destructor. The one-argument constructor is used to initialize the data member x with the value given by the user. The overloaded operators are used to increase and decrease the value of the object. When the user specifies the zero value, an exception is thrown from the constructor. The exception is caught by the catch statement. In function()main(), two objects N and N1 are declared. The exception is thrown when the object N1 is created. When the exception is thrown, the control goes catch block. The catch block terminates the program. The destructors are also invoked.

10.12 Exception and Operator Overloading

An exception handling can be sued with operator-overloaded functions. The following program illustrates the same:

A program to throw exception from overloaded operator function.

```
#include<iostream.h>  #include<process.h>  class number
{
int x;
```

```
public :
number() {};  void operator --();
void show() { cout<<"\n x="<<x; }  number
( int k) { x=k; }
};
void number :: operator --() // prefix notation
{
if (x==0) throw number();  else --x;
}
void main()
{
try
{
number N(4);
cout<<"\n Before Decrementation:";
N.show();
while (1)
{
cout<<"\n After Decrementation";
--N;
N.show();
}
}
catch (number)
{
cout<<"\n Reached to zero";  exit(1);
}
}
```

OUTPUT

```
Before Decrementation  x=4
After Decrementation
x=3
After Decrementation  x=2
After Decrementation  x=1
After Decrementation  x=0
After Decrementation
```

Reached to zero

Explanation: In this program, the operator – is overloaded. When used with class objects, this operator decreases the values of class members. Using the while() loop, the value of the object N is continuously decreased. The object N is initialized with four. The operator –() function checks the value of x (member of object N); if the value of x reaches zero, an exception is thrown, which is caught bythe catch statement.

10.13 Exception & Inheritance

In the last few examples, we have learned how the exception mechanism works with the operator function and with constructors and destructors. The following program explains how exception handling can be done with inheritance:

A program to throw an exception in derived class.

```
#include<iostream.h>  class ABC
{
protected:
char name[15];  int age;
};
class abc : public ABC // public derivation
{
float height;  float weight;  public:
void getdata()
{
cout<<"\n Enter Name and Age:";  cin>>name>>age;
if (age<=0)  throw abc();
cout<<"\n Enter Height and Weight:";  cin>>height >>weight;
}
void show()
{
cout<<"\n Name:"<<name<<"\n Age:"<<age<<" Years";
cout<<"\n Height:"<<height<<"Feets"<<"\n Weight:" <<weight<<"Kg.";
}
};
void main()
{
```

```
try
{
abc x;
x.getdata(); // Reads data through keyboard.  x.show(); // Displays data on the screen.
}
catch (abc) { cout<<"\n Wrong age"; }
}
```

OUTPUT

Enter Name and Age: Amit 0 Wrong age

Explanation: In the above program, the two classes ABC and abc are declared. The class ABC has two protected data members name and age. The class abc has two float data members' height and weight with two member functions getdata() and show(). The class abcis derived from class ABC. The statement class abc: public ABC defines the derived class abc. In the function main(), x is an object of the derived class abc. The object x invokes the member function getdata() and show(). This function reads and displays data, respectively. In the function getdata(), the if statement checks the age entered. If the age entered is zero or less than zero, an exception is thrown. The catch block is executed, and the message "Wrong age" is displayed.

10.14 Summary

A few languages support the exception-handling feature. Without this feature, the programmer needs to detect bugs on their own. The errors may be logical errors or syntactic mistakes (syntax mistakes). The logical error remains in the program due to an unsatisfactory understanding of the program. The syntax mistakes are due to a lack of understanding of the programming language itself.

ANSI C++ is built in language functions for trapping errors and controlling exceptions. All C++ compilers support this newly added facility. C++ provides a type-secure, integrated procedure for coping with the predictable but peculiar conditions that arise in run time. The goal of exception handling is to create a routine that detects and sends an exceptional condition in order to execute suitable actions. An exception is an object. It is sent from the part of the program where an error occurs to the part of the program that is going to control the error. C++ exception method provides three keywords; they are try, throw, and catch. The keyword try is used at the starting of exception. The entire exception statements are enclosed in curly braces. It is known as try block. The catch block receives the exception sent by the throw block in the try block. We can also define multiple catch blocks; in the try block, such programs also contains multiple throw statements based on certain conditions. It is also possible to define a single or default catch block from one or more exceptions of different types. In such a situation, a single catch block is used for catching exceptions thrown by the multiple throw statements. It is also possible to again pass the exception received to another exception handler; that is, an exception is thrown from the catch block; this is known as a re-throwing exception. The specified exceptions are used when we want to bind the function to throw only the specified exception. Specific exception is thrown using condition statement and list of data items in throw.

9.15 Questions

1. Differentiate between template function and non-template function.
2. Write a program to catch multiple exceptions,
3. Why a template is required?
4. What is re-throwing of exception? Write a program to handle this situation.
5. Explain try, throw and catch.
6. When multiple catch statements are required. Give an example.
7. Write a program to throw exception when i=2, otherwise perform multiplication of a and b.
8. What do you mean by Polymorphism?
9. Write a program that throws exception in derived class.
10. Define normal template. Write a program to display data members of template.

9.16 Suggested Readings

1. Object oriented Programming with ANSI and Turbo C++ (Pearson): Ashok N Kamthane
2. Object Oriented Programming with C++, 3/e by E. Balagurusamy, McGraw Hill

Exception handeling

→ Exception are the run time problems or unusual condition that a program may encounter while executing.

eg. division by zero, out of memory space, array is outside of its bounds.

→ Because exception are outside the normal operation of a program, default action is to write out of an error message and terminate the offending process.

Types

Synchronous
↓
Error such as "out of range and over-flow, divide by 0.

Asynchronous
↓
Issue - operating system keyword, memory.
{ It is beyond the control of program }

*. Exception handelling in C++ is designed to handle only Synchronous exceptions.

Exception handling mechanism.

- find the problem
- Inform that an error has occured (Throw an exception) → Key-word } Try.
- Recieve the error information. (catch the exception)
- Take corrective actions. (Handle the exception) } Catch.

⇒ Syntax :

```
----
----
try
{
  ---                    // block of statement which
  throw exception;       detects and throws an exception.
  ---
}
catch (type arg)
  {
    ----
    ---                  // block of statement that
  }                      handles the exceptions.
  ---
```

→ try block: A block of statement which may generates exceptions.

→ When an exception is detected, it is thrown using a throw statement in try block.

→ A Catch block defined by the keyword catch 'catches' the exception 'thrown' by the throw statement in try block and handles it appropriately.

Note:

The Catch block that catches an exception must immediately follow the try block that throws an exception.

Exception handling.

```
main()
{
int a,b,c;
cin>>a>>b;
try {
      if (b==0)
          throw 0;
          c = a/b
          cout<<c;
      }
Catch (int x)
  {
   cout<<"b!=0";
  }
}
```

#

```
try...
main()
{
 int a, b, c;
  cin>>a>>b;
    c = a/b;

 Cout<<c;
   }.
```

Throw point outside the try block

```
void divide (int x, int y)
{
    if (y == 0)
        throw y;
    cout << x/y;
}
main() {
    try {
        divide (3,1);
        divide (4,0);
        divide (40,10);
    }
    Catch (int x) {
        cout << "Exception";
    }
}
```

output = 3 exception.

```
main()
{ int a;
  cout << "Enter a";
      cin >> a;
try {
      if (a==0)
    throw a;
    if (a==1)
```

```
throw 3.1;
if(a == -1)
    throw 'a';
cout<<a;
}
catch(int x)
{
cout<<"Exception";
}
Catch(double x){
cout<<"Exception";
}
catch(char x){
cout<<"Exception";
}
```

* Catch all exceptions:

Suppose situation

Syntax:

```
Catch(...)
{
    //Statements for processing
}   //all exception.
```