
MODULE - V

Templates and Exception Handling, Streams and Files

Templates and Exception Handling, Streams and Files

Module Description

The main goal of this module is to learn the usage of the templates, which is one of the features of C++ programming. This helps the technique of generic programming, where the functions and classes can operate with generic types. This also helps the programmers to utilize the concepts of C++ completely.

At the end of this module, the students will be able to create and implement the usage of the templates, the usage of the function and class templates in the program. Finally at the end of the module students are able to write programs using the concepts like **multiple inheritance and operator overloading**.

Chapter 5.1

Templates and Exceptions

Chapter 5.2

Streams and Files

Chapter Table of Contents

Chapter 5.1

Templates and Exceptions

Aim.....	249
Instructional Objectives.....	249
Learning Outcomes.....	249
5.1.1 Introduction.....	250
5.1.2 Templates.....	250
(i) Function Templates.....	250
(ii) Class Template.....	251
(iii) Template Arguments.....	253
(iv) Member Function Templates.....	254
Self-assessment Questions.....	255
5.1.3 Exceptions.....	255
(i) Handling Exceptions.....	256
(ii) Try, Catch, Throw and Finally Blocks.....	256
(iii) Exceptions with Arguments.....	261
(iv) Exception notes.....	262
Self-assessment Questions.....	263
Summary.....	264
Terminal Questions.....	265
Answer Keys.....	265
Activity.....	266
Bibliography.....	267
e-References.....	267
External Resources.....	267
Video Links.....	267



Aim

To equip the students with the skills to write programs using templates and exceptions in C++ programming



Instructional Objectives

After completing this chapter, you should be able to:

- Explain templates
- Describe class templates and function templates
- Discuss exception handling with examples
- Describe exceptions with arguments along with exception notes



Learning Outcomes

At the end of this chapter, you are expected to:

- Differentiate class templates and function templates
- Elaborate on member function templates and template arguments
- Generate an exception then write a code to handle it
- Explain the role of finally block

5.1.1 Introduction

One extra feature which is added to C++ recently is template. This new concept helps us to define new generic classes and functions which provides for generic programming. In this approach, generic types are used as parameters in algorithms which work for variety of data types and data structures.

A family of classes or functions can be created using these Templates. *For example*, class add() can be created to add different types of data types such as to add integer, float etc.

Templates are also sometimes called as parameterized classes or functions as they are defined with a parameter which can be replaced by a specific data type at the time of actual use of the class or function.

5.1.2 Templates

Template is a feature of C++ programming. It is formed by a process of creating an abstract or generic class from a collection of classes thereby making the abstract (generic) class to behave independently irrespective of data types. This will allow handling all the operations of the class more efficiently and without rewriting the code for each of the operations for the respective data types.

We will look at Function Templates, Class Templates, Member Function Templates and the template arguments in further sections.

(i) Function Templates

Function Templates can be used to create a family of functions with different types of arguments.

The syntax of the function template is as follows:

```
template<class T>
ret_type function_name(parameter-list)
{
    Body of the Function
}
```

In the above syntax, T is a placeholder name for a data type used by the function. This name can be used within the function definition.

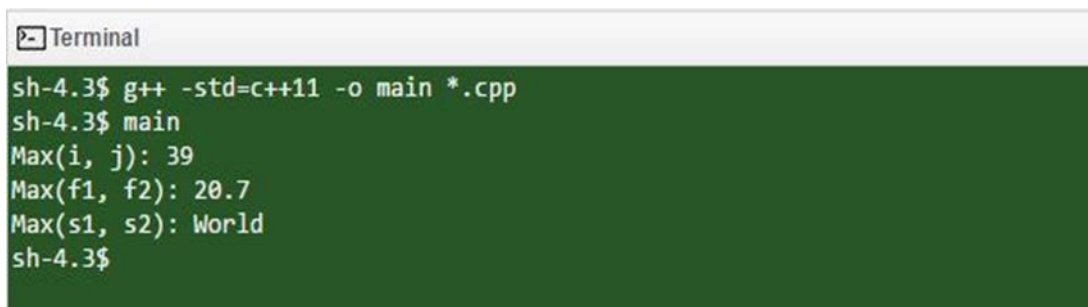
Example Program:

Let us consider the following code which returns the maximum of the two values:

Program:

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  template <typename T>
7  inline T const& Max (T const& a, T const& b)
8  {
9      return a < b ? b:a;
10 }
11 int main ()
12 {
13
14     int i = 39;
15     int j = 20;
16     cout << "Max(i, j): " << Max(i, j) << endl;
17
18     double f1 = 13.5;
19     double f2 = 20.7;
20     cout << "Max(f1, f2): " << Max(f1, f2) << endl;
21
22     string s1 = "Hello";
23     string s2 = "World";
24     cout << "Max(s1, s2): " << Max(s1, s2) << endl;
25
26     return 0;
27 }
```

Output:



```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
sh-4.3$
```

(ii) Class Template

We can also define class templates as we defined function templates,. The general syntax of the generic class is as shown below:

```
Template<class T>
Class class_name
{
    Body of the class
}
```

Here, T is the placeholder of type name, which is specified after the class is instantiated. More than one generic data type can be specified by using a comma-separated list.

Example Program:

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

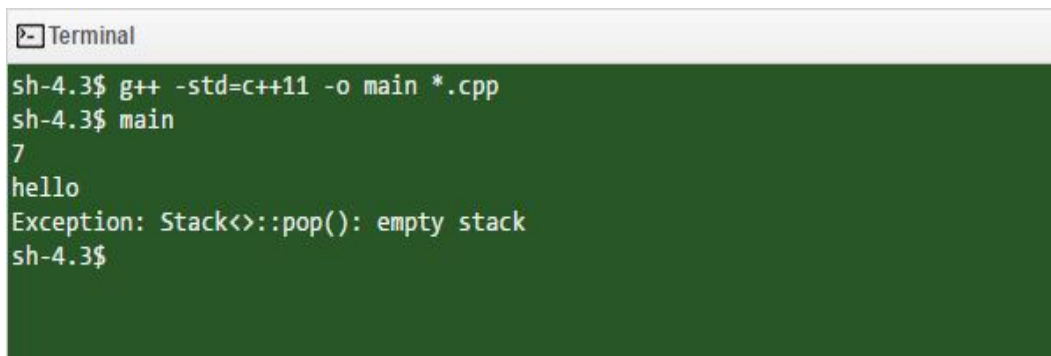
```
1  #include <iostream>
2  #include <vector>
3  #include <cstdlib>
4  #include <string>
5  #include <stdexcept>
6
7  using namespace std;
8
9  template <class T>
10 class Stack {
11     private:
12         vector<T> elems;    // elements
13
14     public:
15         void push(T const&); // push element
16         void pop();          // pop element
17         T top() const;       // return top element
18         bool empty() const{  // return true if empty.
19             return elems.empty();
20         }
21 };
22
23 template <class T>
24 void Stack<T>::push (T const& elem)
25 {
26     // append copy of passed element
27     elems.push_back(elem);
28 }
29
30 template <class T>
31 void Stack<T>::pop ()
32 {
33     if (elems.empty()) {
34         throw out_of_range("Stack<>::pop(): empty stack");
35     }
36     // remove last element
37     elems.pop_back();
38 }
39
40 template <class T>
41 T Stack<T>::top () const
42 {
43     if (elems.empty()) {
44         throw out_of_range("Stack<>::top(): empty stack");
45     }
46     // return copy of last element
47     return elems.back();
48 }
49
50 int main()
```

```

51 {
52     try {
53         Stack<int> intStack; // stack of ints
54         Stack<string> stringStack; // stack of strings
55
56         // manipulate int stack
57         intStack.push(7);
58         cout << intStack.top() << endl;
59
60         // manipulate string stack
61         stringStack.push("hello");
62         cout << stringStack.top() << std::endl;
63         stringStack.pop();
64         stringStack.pop();
65     }
66     catch (exception const& ex) {
67         cerr << "Exception: " << ex.what() << endl;
68         return -1;
69     }
70 }

```

Output:



```

Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
7
hello
Exception: Stack<>::pop(): empty stack
sh-4.3$

```

(iii) Template Arguments

When a template is instantiated, every template parameter (type, non-type, or template) is replaced by a corresponding template argument. For class template the arguments are explicitly given or may be defaulted. For function template the arguments are explicitly provided, defaulted or deduced from the context.

If an argument can be interpreted as both a type-id and an expression, it is always interpreted as a type-id, even if the corresponding template parameter is non-type:

```

template<class T>void f(); // #1
template<int I>void f(); // #2
void g()
{
    f<int()>(); // "int()" is both a type and an expression,
    // calls #1 because it is interpreted as a typ
}

```

(iv) Member Function Templates

Member templates can also be referred as member function templates and nested class templates. These are the template functions which are the members of a class or class template.

If the member functions take their own template arguments, then they are treated as member function templates. The functions of class templates are generic but they are not considered as member templates or member function templates.

Example 1:

```
// member_function_templates.cpp
struct S
{
    template <class Temp>
    void mem_fun(Temp* T) {}
};

int main()
{
    int p;
    S* q = new S();
    q->mem_fun(&p);
}
```

The above example tells a member function templates of non-template or template classes are declared as function templates with their template parameters.

Example 2:

The following example shows a member function template of a template class.

```
// member_function_templates2.cpp
template<typename T>
class X
{
    public:
    template<typename U>
    void mf(const U &u)
    { }
};

int main()
{ }
```



Self-assessment Questions

- 1) Generic Programming is done using
 - a) Structures
 - b) Arrays
 - c) Union
 - d) Templates
- 2) “template” is a keyword.
 - a) True
 - b) False
- 3) For declaring function templates, all the arguments in template declaration must be generic.
 - a) True
 - b) False
- 4) In the template <class T> declaration, T stands for -
 - a) An integer data type
 - b) A generic data type
 - c) An arbitrary class
 - d) A class defined earlier
- 5) After a template is instantiated, every parameter is replaced with its corresponding_____.

5.1.3 Exceptions

We know that the program very often run correctly for the first time. They may have bugs in them. There are two types of bugs in common – **logic errors and syntactic errors**. These errors can be fixed by debugging and testing methods. There are other categories of errors which come across with peculiar problems other than logic and syntactic errors. These are called as **Exceptions**.

Exceptions are the runtime anomalies or unusual conditions that a program may encounter while executing. These anomalies consist of conditions as division by zero, access to an array outside its bound or running out of memory or disk space.

(i) Handling Exceptions

Exceptions are of two types – synchronous exceptions and asynchronous exceptions. Synchronous errors are “out-of-range” and “over-flow”. Asynchronous errors are caused by the events beyond the control of the program. Keyboard interrupt is one such error. The exception handling we discuss here is only for synchronous errors.

Exception handling provides a means to detect and reports an “exceptional circumstances” so that an appropriate action can be taken to handle them. This mechanism provides a separate error handling code which perform the following tasks-

- Find the problem – hit an exception
- Inform an error has occurred – throw the exception
- Receive the error information – catch the exception
- Take corrective actions – handle the exception.

(ii) Try, Catch, Throw and Finally Blocks

The exception handling mechanism in C++ is built on three keywords, namely – **try**, **throw** and **catch**.

The keyword **try** is the prefix to the set of statements which are enclosed with a set of statements which might generate the exceptions. This block is called as **try** block. Once the exception is detected, it is **thrown** using the statement **throw** the **try** block.

The **catch** block catches the expression which is thrown by the **try** block **throw** statement and handles it accordingly. This is illustrated in the following diagram:

The catch block which catches the exception must immediately follow the try block which throws the exception.

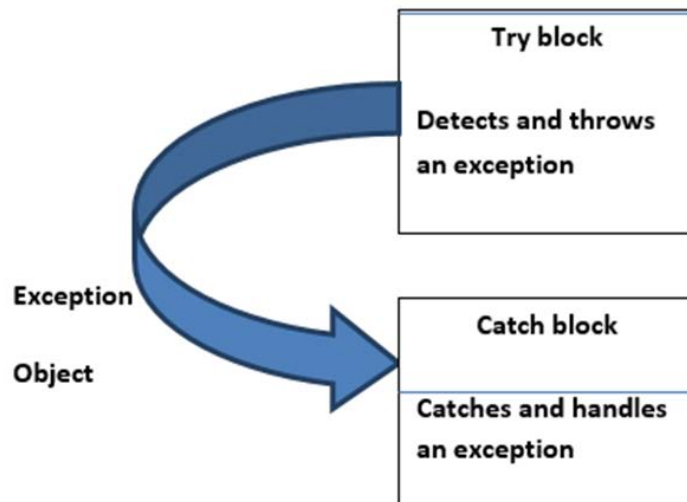


Figure 5.1.1: The Block Throwing Exception

The general form of the above is as shows below:

```

. . . . .
. . . . .
try
{
    . . . . . //block of statements which
detects                                     //block of statements which
    throw exception;                       //detects and throws an exception.
    . . . . .
    . . . . .
}
catch(type arg)                           //Catches Exception
{
    . . . . .
    . . . . . //Block of statements that
handles                                     //the exception.
    . . . . .
    . . . . .
}
. . . . .
. . . . .

```

When a try block is executed, it throws an exception; the control of the program is transferred to the catch statement of the catch block. Here the exceptions are the objects used for the information about the problem. If the object which is thrown matches to the arg type in the catch statement, the block is executed for the handling of the exception. Suppose if they do not match the arg, the program is aborted with the help of abort() function, by default. If there is no exception is detected and thrown, then the control is directly transferred to the next statement of the catch block, by skipping the catch block.

For example,

```
1  #include<iostream>
2
3  int main()
4  {
5      int x,y,z;
6      float p;
7
8      std::cout<<"Enter the value of x:";
9      std::cin>>x;
10     std::cout<<"Enter the value of y:";
11     std::cin>>y;
12     std::cout<<"Enter the value of z:";
13     std::cin>>z;
14     try
15     {
16         if((x-y)!=0)
17         {
18             p=z/(x-y);
19             std::cout<<"Result is:"<<p;
20         }
21         else
22         {
23             throw(x-y);
24         }
25     }
26     catch(int a)
27     {
28         std::cout<<"Answer is infinite because x-y is:"<<a;
29     }
30
31
32 }
33
```

Output:

options	compilation	execution
<pre>Enter the value of x:20 Enter the value of y:20 Enter the value of z:40 Answer is infinite because x-y is:0</pre>		
Exit code: 0 (normal program termination)		

The above program illustrates the exception handling for the divide by zero exception. Program detects and catches an exception divide by zero error.

Throw:

Exceptions are thrown by functions which are invoked from within the **try** blocks. The point at which the throw block is executed is called as **throwpoint**. Once the exception is transferred to the catch block, then the control cannot be transferred back to the throw point.

This is illustrated as follows:

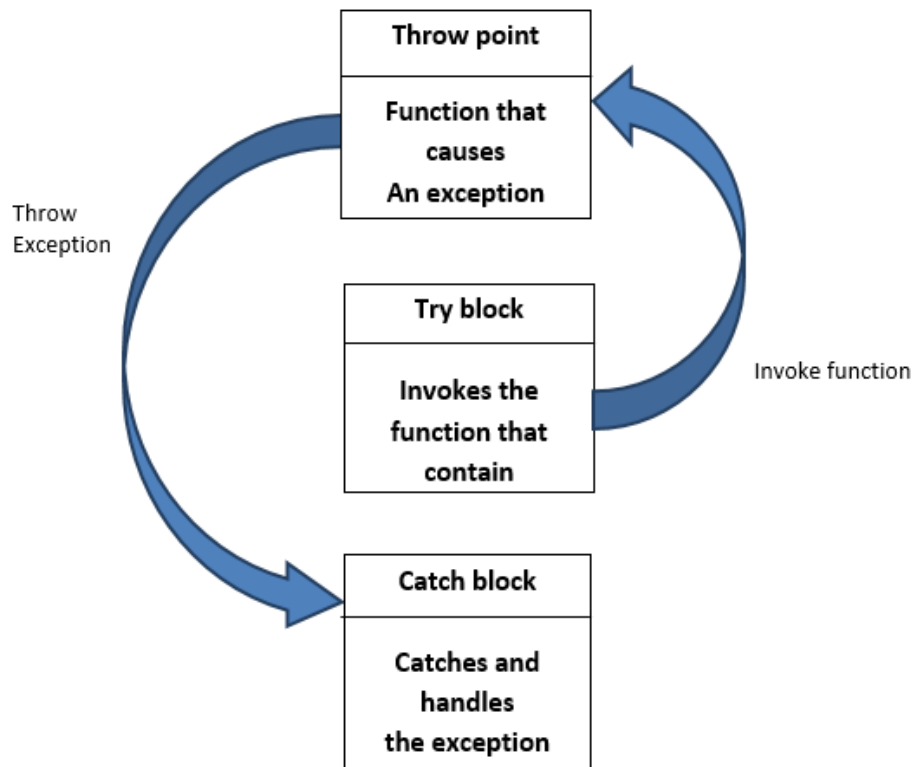


Figure 5.1.2: Function Invoked by Try Block Throwing Exception

The following block gives the general format of the above illustration.

```
type func(arg_list)                                //function with exception
{
    . . . . .
    throw(obj);                                    //throws exception
    . . . . .
}
. . . . .
try
{
    . . . . .
    . . . . .                                //invoke function in this block
    . . . . .
```

```

}
catch(type argument)                //catches exception
{
    . . . . .
    . . . . .                //handles exception in this block
    . . . . .
}
. . . . .
. . . . .
. . . . .

```

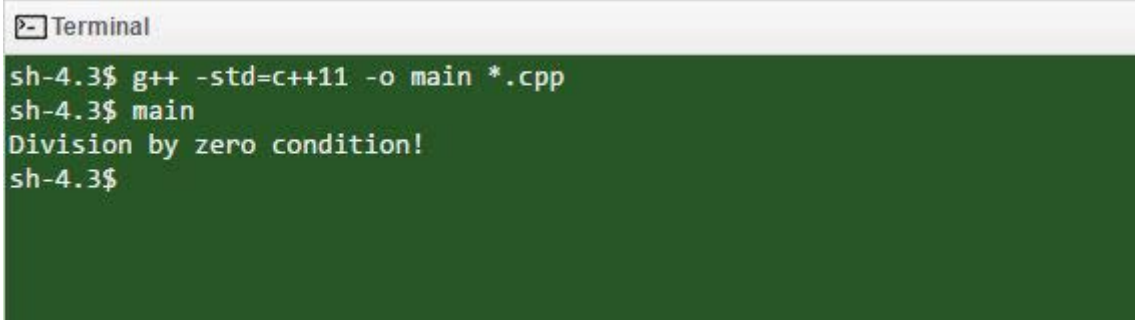
The following code illustrates the above:

```

1  #include <iostream>
2  using namespace std;
3
4  double division(int a, int b)
5  {
6      if( b == 0 )
7      {
8          throw "Division by zero condition!";
9      }
10     return (a/b);
11 }
12
13 int main ()
14 {
15     int x = 50;
16     int y = 0;
17     double z = 0;
18
19     try {
20         z = division(x, y);
21         cout << z << endl;
22     }catch (const char* msg) {
23         cerr << msg << endl;
24     }
25
26     return 0;
27 }

```

Output:



```

Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Division by zero condition!
sh-4.3$

```

When the above code is executed, the function division is executed and the control is transferred to the function definition. Here the difference is checked for zero, if yes then the function throws an exception. If the division is not zero, then the remaining part of the code is executed and the result will be displayed.

(iii) Exceptions with Arguments

A throw expression accepts one parameter, which is passed as an argument to the exception handler. The exception handler is declared with the catch keyword immediately after the closing brace of the try block. The syntax for catch is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it. Only in the case they match, the exception is caught by that handler. Multiple handlers (*i.e.*, catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed. If an ellipsis (...) is used as the parameter of catch, that handler will catch any exception no matter what the type of the exception thrown.

The following code demonstrates the working of exceptions with arguments -

```
class MyIOException : public std::exception
{
public:
    virtual const char* what() const throw()
    {
        return "IO Exception";
    }
};
class MyEmptyImageException : public MyIOException
{
    std::string m_msg;
public:

    MyEmptyImageException(const std::string& bn, const std::string& on)
        : m_msg(std::string("Empty Image : ") + bn + on)
    {}

    virtual const char* what() const throw()
    {
        return m_msg.c_str();
    }
};

int main(int argc, char** argv)
{
    try
```

```
{
    // ... read image
    if (image.empty())
    {
        throw MyEmptyImageException(folderName, imageName);
    }
    // ... remained code
}
catch (MyEmptyImageException& meie)
{
    std::cout << meie.what() << std::endl;
}
catch (std::exception& e)
{
    std::cout << e.what() << std::endl;
}
return 0;
}
```

(iv) Exception Notes

So far we have learnt the simplest and the most common approach of usage of exceptions. Now we'll conclude a few thoughts about exception usage.

Function Nesting: Here the exception which is caused not necessary to be located directly in the try block; but can also be in a function called by a statement in the try block. It's sufficient to install a try block in the upper level of a program. So that the lower levels of the program need not to be overloaded, instead they can directly or indirectly call the try block.

Exceptions and Class Libraries: When a library routine is written by a programmer and it is called in a program, it is required to pass the error to the program which called it. The calling program thus handles the error to fix it. Here the exception mechanism provides the capability because exceptions are nested up through until a catch block is encountered. Later, a throw statement may be in a library routine, but the catch block may be in the program which handles the error caused. If you are writing a class library, you should cause it to throw exceptions for anything that could cause problems to the program using it. If you are writing a program that uses a class library, you should provide try and catch blocks for any exceptions that it throws.

Not for every situation: Exceptions cannot be used at every situation. They oblige a certain overhead in terms of program size and time. *For example*, exceptions may not be used for the user input errors like inputting data which can be easily detectable by the program. Instead the program should use normal decisions and loops to check the user's input and ask the user to try again if necessary.

Destructors Called Automatically: The exception mechanism is very sophisticated. When an exception is thrown, a destructor is automatically called for an object that was created by the code up to the point in the try block. This is required because the application will not be aware of which statement caused the exception, for recovering from the error. The exception mechanism guarantees that the code in the try block will have been “reset,” at least as far as the existence of objects is concerned.

Handling Exceptions: After the exception is caught, we need some time to terminate the application. The exception mechanism gives a way to indicate the source for the error and also the way to perform the clean-up the errors before terminating. This clean-up is done by making a call to the destructor for the objects that are created in the try block. This allows us to release the system resources.



Did you Know?

The try block is immediately followed by catch block, irrespective of the location of the throw point.



Self-assessment Questions

- 6) Exceptions are runtime errors.
 - a) True
 - b) False
- 7) If an exception is not caught, it will be ignored.
 - a) True
 - b) False
- 8) Throwing an exception always causes program termination.
 - a) True
 - b) False
- 9) Exceptions thrown from outside try block will _____.
 - a) Call function return
 - b) Be ignored
 - c) Hang the machine
 - d) Call function terminate



Summary

- C++ supports a mechanism known as templates which implements the concept of generic programming. It also generates a family of classes or a family of functions to handle different data types.
- A specific class will be created from a class template which is called as template class and the procedure is called as instantiation. In the same way the function template is also created.
- Template functions can be overloaded. Member functions should be defined as a function template using the parameters of the functions. Non – type parameter such as derived data types can also be used as the arguments to the templates.
- Exceptions are the runtime errors/anomalies which are caused during the program execution.
- Exceptions can be of two types - **Asynchronous Exceptions** and **Synchronous Exceptions**, as errors beyond the program execution like keyboard error and divide by zero, array out of bound etc.,. Respectively.
- Exceptions can be handled using **try**, **catch**, **throw** and **finally** methods. A **try** block can through the exception by invoking a function which throws an exception. A **catch** block is immediately placed after the try block irrespective of the location.
- One or more **catch** blocks can be placed to handle multiple types of exceptions thrown. Only one catch is also enough to catch all types of exceptions using ellipses as the argument.



Terminal Questions

1. What are templates in C++?
2. Describe class templates and function templates.
3. Discuss exception handling with examples.
4. Describe exceptions with arguments along with exception notes.



Answer Keys

Self-assessment Questions	
Question No.	Answer
1	d
2	a
3	a
4	b
5	Template Argument
6	a
7	b
8	b
9	d



Activity

Activity Type: Online
Minutes

Duration: 45

Description:

1. Develop and execute a program to sort a given set of elements using function template.
2. Develop and execute a program to search a key element in a given set of elements using class template.
3. Develop and execute a program to find average marks of the subjects of a student. Throw multiple exceptions and define multiple catch statements to handle division by zero as well as array index out of bounds exceptions.

Bibliography



e-References

- tutorialspoint.com, (2016). *C++ Exception Handling*. Retrieved 7 April, 2016. from, http://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm
- ntu.edu, (2016). *C++ IO Streams and File Input/output*. Retrieved 7 April, 2016. from, http://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp10_io.html



External Resources

- Balaguruswamy, E. (2008). *Object Oriented Programming with C++*. Tata McGraw Hill Publications.
- Lippman. (2006). *C++ Primer* (3rd ed.). Pearson Education.
- Robert, L. (2006). *Object Oriented Programming in C++* (3rd ed.). Galgotia Publications References.
- Schildt, H., & Kanetkar, Y. (2010). *C++ completer* (1st ed.). Tata McGraw Hill.
- Strousstrup. (2005). *The C++ Programming Language* (3rd ed.). Pearson Publications



Video Links

Topic	Link
Templates	https://www.youtube.com/watch?v=bGIn4ArVpvQ
Function templates, Class templates	https://www.youtube.com/watch?v=3LvZ-RN8kM
Exception Handling	https://www.youtube.com/watch?v=FynvrvLw3Rs



Notes:

