Unit 14 Exception Handling

Structure:

14.1 Introduction

In the previous unit you have studied standard template library and its components. In this unit you will study the concept of exception handling in C++. The program which we write might have bugs. The common types of errors which occur in our programs are – logical and syntactic errors. The reason for logical errors is the poor understanding of the problem and solution procedure. Syntactic errors occur due to the poor understanding of the programming language. These errors can be detected by debugging and testing procedures. Sometimes we come across errors other than the above mentioned ones. These errors are known as exceptions. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside its bounds or running out of memory or disk space. When such an exceptional condition occurs in a program, it should be identified and handled effectively. Some built in features are provided by C++ to detect and handle the exceptions which are basically run-time errors.

Objectives:

After studying this unit you should be able to:

- describe exception, exception handling, and the necessity to have exception handling.
- explain the usage of try and catch blocks.
- explain the models of exception-handling, and the necessity of exception specifications.
- discuss the special functions unexpected(), terminate(), and more.
- describe the situations when the exceptions should be avoided and used.

14.2 Basics of Exception Handling

There are two kinds of exceptions i.e. synchronous and asynchronous exceptions. Errors like “out of range index” and “overflow” belong to the category of synchronous exceptions. The errors which occur beyond the control of the program (like keyboard interrupts) belong to the category of asynchronous interrupts. The proposed exception handling mechanism in C++ is designed to handle only asynchronous exceptions.

The aim of exception handling mechanism is to identify an exception and take necessary action to handle it. The mechanism suggests a separate error handling code that performs the following tasks:

1. Find the problem (hit the exception).
2. Inform that an error has occurred (throw the exception).
3. Receive the error information (catch the exception).
4. Take corrective actions (handle the exception).

There are two parts in error handling code: one is to detect errors and to throw exceptions; the other is to catch the exceptions and to take appropriate actions.

14.3 Exception Handling Mechanism

The exception handling mechanism in C++ is built upon the three keywords named as **try, throw** and **catch.** The block of statements which generates exceptions are prefaced by the “try” keyword. These blocks of statements are called “try block”. When, an exception is detected, it is thrown using a **throw** statement in the try block. A “catch block” defined by the keyword “catch” catches the exception thrown by the throw statement in the try block, and handles it appropriately. The relationship is shown in figure 14.1:

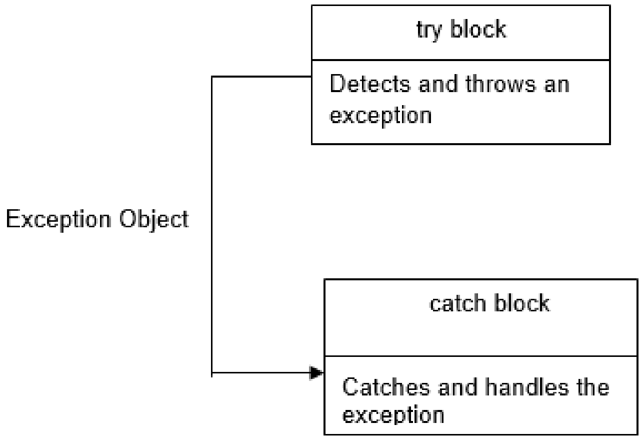
try block
Detects and throws an exception
Exception Object
catch block
Catches and handles the exception

Figure 14.1: The *block throwing* exception

When an exception is thrown by a try block, the control is transferred to the catch block and program leaves the try block. It should be noted that the exceptions are objects that transmit the information about a problem. If a type of object thrown matches the arg type in the catch statement, then the catch block is executed for handling the exception. If it doesn’t match, the program is aborted with the help of abort() function which is invoked by default. When there is no exception then the control is transferred to the statement immediately after the catch block i.e. the catch block is skipped. The simple try catch mechanism is shown in the program below.

```
#include<iostream.h>
int main()
 {
   int a,b;
   cout << “ Enter the values of a and b:”;
   cin >> a;
```

```
    cin >> b;
    int x= a-b;
  try
  {
    if(x!=0)
     {
       cout << “result (a/x) =” << a/x << “\n”;
     }
   else            //there is an exception
    {
      throw (x);   //throws int object
    }
}
catch(int i)       //catches the exception
 {
   cout << “exception caught : divide by zero error\n”;
 }
 cout << “end”;
 return 0;
}
```

The output of the above program will be:

First run

```
Enter the values of a and b: 10 5
result (a/x) = 2
end
```

Second run

```
Enter the values of a and b: 5 5
exception caught: divide by zero
end
```

You can see that the first run has shown a successful execution. In the first run, there is no exception and hence, catch block is skipped and the execution resumes with the first line after the catch. You can observe that in second run the ‘divide by zero’ error occurs as the denominator x becomes zero. Using the object x the exception is thrown. As the data type of the exception object is int the catch statement containing int type argument catches the exception and the necessary message is displayed.

Most often, exceptions are thrown by functions that are invoked from within the try blocks. The point at which throw is executed is called throw point. The control cannot be returned to the throw point once the exception is thrown to the catch block. This relationship is shown in figure 14.2.

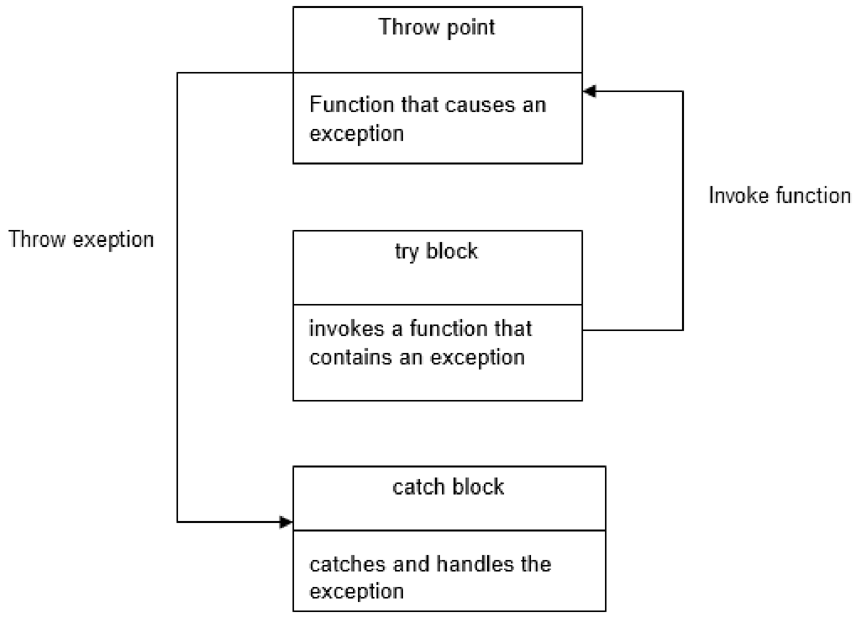
Throw point
Function that causes an exception
Invoke function
Throw exeption
try block
invokes a function that contains an exception
catch block
catches and handles the exception

Figure 14.2: Function invoked by try block throwing exception

The code below shown is the general format of this kind of code

```
type function (arg list)          // function with exception
{
  ……….
………..
throw (object);                   //throws exception
………
………
}
……….
……….
try
 {
   ………
   ……… invoke function here
   ……..
}
catch(type arg)                    // catches exception
 {
   ……….
   ………. Handles exception here
   ……….
}
……….
```

The program below illustrates how a try block invokes a function that generates an exception.

```
// throw point outside the try block
#include<iostream.h>
void divide (int x, int y, int z)
  {
    cout << "\n we are inside the function \n";
    if ((x-y)! =0)
      {
        int R = z/ (x-y)
        cout << "Result =" << R << "\n";
      }
   else
     {
       throw(x-y)              //throw point
     }
}
void main()
  {
    try
    {
      cout << " we are inside the try block \n";
       divide (10,20,30)             //invoke divide()
       divide (10,10,10)             // invoke divide()
    }
catch(int i)
 {
  cout << “caught the exception\n”;
 }
}
```

The output of the above program will be as follows:

```
we are inside the try block
we are inside the function
Result = – 3
We are inside the function
caught the exception
```

Self Assessment Questions

1. _______________ are run time unusual conditions that a program may encounter while executing.
2. The two kinds of exceptions are ________ and _________ exceptions.
3. The block of statements which generates exceptions is prefaced by the ________ keyword
4. When an exception is thrown by a try block, then the control is transferred to the ________.

14.4 Throwing Mechanism

When an exception that is desired to be handled is detected, then it is thrown using the “throw” statement. It can take one of the following forms:

```
throw (exception);
throw exception;
throw;          //it is used for rethrowing an exception
```

The operand object exception may be of any type including constants. It is also possible to throw objects not intended for error handling.

The catch statement associated with the try block catches the exception when the exception is thrown. That is the control exits the current try block and is transferred to the catch block after the try block.

14.5 Catching Mechanism

As you have already studied, we include code for handling exceptions in catch blocks. A catch block looks like a function definition and it has the following form:

```
catch (type arg)
```

```
 {
   //statements to manage exceptions
}
```

Here the type shows the type of the exceptions to be handled by the catch block. The parameter arg is an optional parameter name. Between the two braces the exception handling code is placed. The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed.

The parameter can be used in the exception handling code if the parameter in the catch statement is named. After the handler is executed, the control is transferred to the statements immediately following the catch blocks.

If an exception is not caught due to mismatch, then the program will terminate abnormally. If an exception is not caught by catch statements then the catch block is simply skipped.

14.5.1 Multiple catch statements

Sometimes program segment has more than one condition to throw an exception. In such cases you can associate more than one catch blocks with a try block. The syntax to achieve this task is shown below:

```
 try
  {
    //try block
   }
catch (type1 arg)
 {
   //catch block1
}
catch (type2 arg)
 {
   //catch block2
 }
……..
……..
catch(typeN arg)
  {
    //catch blockN
  }
```

When an exception occurs, there is a search for an appropriate exception handler. The handler which is matched is executed. After the handler is executed, the control is transferred to the statement immediately after the catch block for that try. The program is terminated if there is no match.

If arguments of more than one catch statements match the type of exception, the first handler that matches with the exception is executed.

The example below shows how the various types of exceptions are handled by multiple catch statements.

```
#include<iostream.h>
void test(int x)
  {
        try
         {
           if( x==1) throw x;              //int
           else
              if (x==0) throw 'x';         //char
           else
              if (x==-1) throw 1..0;       //double
           cout << "end of try block\n";
         }
        catch (char c)                     //catch1
                {
                        cout << " caught a character \n";
                }
        catch (int m)                      //catch2
                {
                        cout << "caught an integer\n";
                }
        catch (double d)                   //catch3
                {
                        cout << "caught a double\n";
                }
        cout << "end of try catch system\n";
 }
```

```
void main()
        {
                cout << “testing multiple catches\n”;
                cout << “x ==1 \n”;
                test (1);
                cout << “x==0 \n”;
                test(0);
                cout << “x==-1 \n”;
                test (-1);
                cout << “x==2 \n”;
                test (2);
        }
```

The output of the above program will be:

```
testing multiple catches
x==1
caught an integer
end of try cacth system
x==0
caught a character
end of try catch system
x==-2
end of try block
end of try catch system
```

As you can observe from the above program, when it is executed first the function test() with x=1 is invoked and hence throws x an int exception. There is match of type of parameter m in catch2 and hence the catch2 handler is executed. The function test() with x=0 is invoked immediately after the execution of catch2 handler. This type the function throws ‘x’, a character type exception and hence the first handler is executed. Finally, the handler catch3 is executed when a double exception is thrown. You should note that each time only the handler which catches the exception is executed and all the rest handlers are skipped.

When no exception is thrown by the try block and if there is a normal execution, then the control is transferred to the first statement after the last catch handler associated with that try block.

Self Assessment Questions

5. When an exception is detected, it is thrown using the ______________ statement.
6. A catch block looks like a function definition and it has the form _________.

14.6 User Defined Exception Class

It is possible to create your own exception class and use them as exception types in your exception handling code instead of using predefined data types in C++ as exception. You can achieve further control over exception handling by defining your own exception classes. This can be explained with the following example.

Let us assume that your program requires an input number in the range of 0 to 100. So instead of using built-in data types of C++, we will write our own exception classes to validate the input. An out of range exception will be thrown if user inputs a number greater than 100. And a negative number exception is thrown if user inputs a number less than zero. To achieve this we need to create the two exception classes of our own.

First we will create an exception for out of range condition. We define a class called OutofRange as follows:

```
class OutofRange
        {
                public:
                        OutofRange()
                                {
                                  cout << “exception :out of range” << endl;
                                }
        };
```

As you can see above that a constructor is defined by the class that prints a message to the user whenever an object of this type is constructed. Similarly, we will create another class for negative number input. The class definition is as follows:

```
class NegativeNumber
        {
                public:
                    NegativeNumber()
                          {
                            cout << "exception: Negative input" << endl;
                          }
};
```

Again, the class constructor prints an appropriate message on the user console. Once you create the two user-defined cl for classes for exception, you need to use these data types in the catch block of your error handler code. To catch OutofRange, we the catch block will be as shown below:

```
      catch(OutofRange)
       {
            //error handler code
      }
```

Similarly to catch negative number exception, the catch block will be as follows:

```
      catch(NegativeNumber)
       {
            //error handler code
      }
```

Next you need to throw exceptions of these types in your try block. For this you need to construct an object of the user-defined exception and then use it as a parameter in the throw statement. For example: to throw OutofRange exception, the following code will be used:

```
      OutofRange e;
      throw e;
```

14.7 Termination vs. Resumption

There are two basic models in exception-handling theory. In *termination* (which is what C++ supports) you assume the error is so critical that there's no way to get back to where the exception occurred. Whoever threw the exception decided there was no way to salvage the situation, and they don't *want* to come back. The alternative is called *resumption*. It means the exception handler is expected to do something to rectify the situation, and then the faulting function is retried, presuming success the second time. If you want resumption, you still hope to continue execution after the exception is handled, so your exception is more like a function call – which is how you should set up situations in C++ where you want resumption-like behavior (that is, don't throw an exception; call a function that fixes the problem). Alternatively, place your *try* block inside a *while* loop that keeps reentering the *try* block until the result is satisfactory.

Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. So although resumption sounds attractive at first, it isn't quite so useful in practice. One reason may be the distance that can occur between the exception and its handler; it's one thing to terminate to a handler that's far away, but to jump to that handler and then back again may be too conceptually difficult for large systems where the exception can be generated from many points.

Self Assessment Questions

7. We can achieve further control over exception handling by __________
8. In __________ we assume the error is so critical that there's no way to get back to where the exception occurred.

14.8 Exception Specifications

It is not necessary to inform the person using your function what exceptions you might throw. However, this is not considered as a good practice as he cannot be sure what code to write to catch all potential exceptions. Of course, if he has your source code, he can hunt through and look for *throw* statements, but very often a library doesn't come with sources. C++ provides us the facility to inform the user what exceptions are thrown by the function, so it can be handled by the user. This is the *exception specification* and it's part of the function declaration, appearing after the argument list.

The exception specification reuses the keyword *throw*, followed by a parenthesized list of all the potential exception types. So your function declaration may look like

```
void f() throw(toobig, toosmall, divzero);
```

With exceptions, the traditional function declaration

```
void f();
```

means any type of exception may be thrown from the function. If you say

```
void f() throw();
```

it illustrates that a function doesn't throw any exceptions.

For good coding policy, good documentation, and ease-of-use for the function caller, you should always use an exception specification when you write a function that throws exceptions.

14.8.1 unexpected()

If your exception specification claims you're going to throw a certain set of exceptions and then you throw something that isn't in that set, what's the penalty? When something other than what appears in the exception specification is thrown, a special function *unexpected()* is called.

14.8.2 set_unexpected()

unexpected() is implemented with a pointer to a function, and you can change its behavior. You do so with a function called *set_unexpected()* which, like *set_new_handler()*, takes the address of a function with no arguments and void return value. Also, it returns the previous value of the *unexpected()* pointer so that you can save it and restore it later. To use *set_unexpected()*, you must include the header file <*exception*>.

14.8.3 Catching any exception

If your function has no exception specification, any type of exception can be thrown. One solution to this problem is to create a handler that *catch*es any type of exception. You do this using the ellipses in the argument list:

```
catch(...) {
   cout << "an exception was thrown" << endl;
}
```

This will catch any exception, so you'll want to put it at the *end* of your list of handlers to avoid pre-empting any that follow it. The ellipses give you no possibility to have an argument or to know anything about the type of the exception. It's a catch-all.

Self Assessment Questions

9. The exception specification reuses the keyword ________, followed by a parenthesized list of all the potential exception types.
10. ___________ is implemented with a pointer to a function, so you can change its behavior.

14.9 Rethrowing an Exception

Sometimes you may want to rethrow the exception that you just caught, particularly when you use the ellipses to catch any exception because there's no information available about the exception. This is accomplished by saying *throw* with no argument:

```
catch(...) {
   cout << "an exception was thrown" << endl;
   throw;
}
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

The program below illustrates how an exception is rethrown and caught.

```
#include<iosream.h>
void divide (double x, double y)
         {
                   cout << “inside function\n”;
         try
           {
              if(y==0.0)
                   throw y;        //throwing double
             else
                   cout << “division = “ << x/y << “\n”;
         }
     catch (double)              //catch a double
         {
                   cout << “caught double inside function \n”;
                   throw;
         }
         cout << “end of function \n\n”;
 }
void main()
         {
                   cout << “inside main \n”;
                   try
                    {
                           divide(10.5, 2.0);
                           divide(20.0, 0.0);
                   }
                   catch(double)
                   {
                           cout << “caught double inside main \n”;
                   }
cout << “end of main \n”;
}
```

The output of the above program will be:

```
inside main
inside function
division = 5.25
end of function
inside function
caught double inside function
caught double inside main
end of main
```

When an exception is rethrown, it will not be caught by the same catch statement or any other catch in the group. Instead, it will be caught by an appropriate catch in outer try/catch sequence only.

A catch handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any catch statements in that group. It will be passed on to the next outer try/catch sequence for processing.

14.10 Uncaught Exceptions

If none of the exception handlers following a particular *try* block matches an exception, that exception moves to the next-higher context, that is, the function or *try* block surrounding the *try* block that failed to catch the exception. (The location of this higher-context *try* block is not always obvious at first glance.) This process continues until, at some level, a handler matches the exception. The exception is considered to be caught at that point and no further searching takes place.

The exception is uncaught if no handler catches the exception at any level. An uncaught exception also occurs if a new exception is thrown before an existing exception reaches its handler – the most common reason for this is that the constructor for the exception object itself causes a new exception.

14.10.1 terminate()

This function is called automatically if an exception is uncaught. Like *unexpected()*, terminate is actually a pointer to a function. Its default value is the Standard C library function *abort()*, which immediately exits the program with no calls to the normal termination functions (which means that destructors for global and static objects might not be called).

No destructors are called for an uncaught exception. If you don't wrap your code (including, if necessary, all the code in *main()*) in a try block followed by handlers and ending with a default handler (*catch(...)*) to catch all exceptions, then you will take your lumps. An uncaught exception should be thought of as a programming error.

14.10.2 set_terminate()

Using this function, it is possible to install your own terminate () function, which returns a pointer to the terminate() function you are replacing, so you can restore it later if you want. Your custom *terminate()* must take no arguments and have a *void* return value. In addition, any *terminate()* handler you install must not return or throw an exception, but instead must call some sort of program-termination function. If *terminate()* is called, it means the problem is unrecoverable. Like *unexpected()*, the *terminate()* function pointer should never be null.

Self Assessment Questions

11. Rethrowing is achieved by using throw with ________.
12. Using __________ function it is possible to install your own terminate() function.

14.11 Standard Exceptions

The set of exceptions used with the Standard C++ library are also available for your own use. Generally it's easier and faster to start with a standard exception class than try to define your own. If the task which we want to perform cannot be done by the standard then we should derive from it.

The following table describes the standard exceptions:

Table 14.1: Standard Exceptions

Exception	The base class for all the exceptions thrown by the C++ standard library. You can ask *what()* and get a result that can be displayed as a character representation.
logic_error	Derived from *exception*. Reports program logic errors, which could presumably be detected before the program executes.
runtime_error	Derived from *exception*. Reports runtime errors, which can presumably be detected only when the program executes.

The iostream exception class *ios::failure* is also derived from *exception*, but it has no further subclasses.

The classes in both of the following tables 14.2 and 14.3 can be used as they are, or they can act as base classes to derive your own more specific types of exceptions.

Table 14.2: Exceptions derived from logic_error

domain_error	Reports violations of a precondition.
invalid_argument	Indicates an invalid argument to the function it's thrown from.
length_error	Indicates an attempt to produce an object whose length is greater than or equal to NPOS (the largest representable value of type *size_t*).
out_of_range	Reports an out-of-range argument.
Bad_cast	Thrown for executing an invalid *dynamic_cast* expression in run-time type identification
Bad_typeid	Reports a null pointer *p* in an expression *typeid(*p)*.

Table 14.3: Exceptions derived from runtime_error

range_error	Reports violation of a postcondition.
overflow_error	Reports an arithmetic overflow.
bad_alloc	Reports a failure to allocate storage.

14.12 Programming with Exceptions

For most programmers, especially C programmers, exceptions are not available in their existing language and take a bit of adjustment. Here are some guidelines for programming with exceptions.

14.12.1 When to avoid exceptions

Exceptions aren't the answer to all problems. The following sections point out situations where exceptions are not required:

- **Not for asynchronous events** – The Standard C *signal()* system, and any similar system, handles asynchronous events: events that happen outside the scope of the program, and thus events the program cannot anticipate. The asynchronous events cannot be handled by C++ exceptions the reason is that the exception and its handler are on the same call stack. That is, exceptions rely on scoping, whereas asynchronous events must be handled by completely separate code that is not part of the normal program flow (typically, interrupt service routines or event loops). This is not to say that asynchronous events cannot be *associated* with exceptions. But the interrupt handler should do its job as quickly as possible and then return. Later, at some well-defined point in the program, an exception might be thrown *based on* the interrupt.
- **Not for ordinary error conditions** – If you have enough information to handle an error, it's not an exception. It should be taken care of in the current context rather than throwing an exception to a larger context. Also in C++ exceptions are not thrown for the machine level events. They should be handled by some other mechanisms like the operating system or hardware. That way, C++ exceptions can be reasonably efficient, and their use is isolated to program-level exceptional conditions.
- **Not for flow-of-control** – We should not use the exceptions other than their original intent. For example some programmers may think of using exceptions instead of switch statement and as an alternate return mechanism. This should not be done because the exception-handling system is significantly less efficient than normal program execution; exceptions are a rare event, so the normal program shouldn't pay for them. Also, exceptions from anything other than error conditions are quite confusing to the user of your class or function.
- **You're not forced to use exceptions** – Many programs are very simple in their implementations. Only thing required in the program is to take input and perform some processing. Some problems may arise while

executing these programs like you might attempt to allocate memory and fail, or try to open a file and fail, and so on. It is acceptable in these programs to use *assert()* or to print a message and *abort()* the program, allowing the system to clean up the mess, rather than to work very hard to catch all exceptions and recover all the resources yourself. So you should not use exceptions if there is not a need to use it.

- **New exceptions, old code –** Sometimes a situation may arise in which we want to modify an existing program that is not using any exception. You may introduce a library that *does* use exceptions and wonder if you need to modify all your code throughout the program. Assuming you have an acceptable error handling scheme already in place, the most sensible thing to do here is surround the largest block that uses the new library (this may be all the code in *main()*) with a *try* block, followed by a *catch(...)* and basic error message. You can refine this to whatever degree necessary by adding more specific handlers, but, in any case, the code you're forced to add can be minimal. You can also isolate your exception-generating code in a try block and write handlers to convert the exceptions into your existing error-handling scheme. It's truly important to think about exceptions when you're creating a library for someone else to use, and you can't know how they need to respond to critical error conditions.

14.12.2 Using exceptions

Use exceptions to:

- Fix the problem and call the function (which caused the exception) again.
- Patch things up and continue without retrying the function.
- Calculate some alternative result instead of what the function was supposed to produce.
- Do whatever you can in the current context and rethrow the same exception to a higher context.
- Do whatever you can in the current context and throw a different exception to a higher context.
- Terminate the program.
- Make your library and program safer. This is a short-term investment (for debugging) and a long-term investment (for application robustness).

- Always use exception specifications – The exception specification is like a function prototype: It tells the user to write exception handling code and what exceptions to handle. It tells the compiler the exceptions that may come out of this function. It is not possible to find out what exceptions will arise from a particular function by looking at the code. Sometimes unexpected exception is generated by the function it calls. And sometimes an old function that didn't throw an exception is replaced with a new one that does, and you'll get a call to *unexpected()*. Anytime you use exception specifications or call functions that do, you should create your own *unexpected()* function that logs a message and rethrows the same exception.
- Start with standard exceptions – Before you throw any exception you should check the Standard C++ library. If you find a standard exception that does what we need then that will be easier for user to understand and handle. You should try to derive the exception from an existing standard exception if the desired exception type is not part of the standard library. It's nice for your users if they can always write their code to expect the *what()* function defined in the *exception()* class interface.
- Nest your own exceptions – If you create exceptions for your particular class, it is a very good idea to nest the exception classes inside your class to provide a clear message to the reader that this exception is used only for your class. In addition, it prevents the pollution of the namespace. It is possible to nest your exceptions even if it is derived from C++ standard exceptions.
- Use exception hierarchies – Many types of critical errors that can be encountered with our class or library are classified by the exception hierarchies. This gives helpful information to users, assists them in organizing their code, and gives them the option of ignoring all the specific types of exceptions and just catching the base-class type. Also, any exceptions added later by inheriting from the same base class will not force all existing code to be rewritten – the base class handler will catch the new exception. Of course, the Standard C++ exceptions are a good example of an exception hierarchy, and one that you can use to build upon.

- Catch by reference, not by value – If you throw an object of a derived class and it is caught *by value* in a handler for an object of the base class, that object is "sliced" – that is, the derived-class elements are cut off and you'll end up with the base-class object being passed. The possibility is that the result is not what we want because the object will behave like a base class object and not the derived class object it really is (or rather, was – before it was sliced). Here is an example:

```
//: C07:Catchref.cpp
// Why catch by reference?
#include <iostream>
using namespace std;
class Base {
  public:
    virtual void what() {
      cout << "Base" << endl;
    }
};
class Derived : public Base {
  public:
    void what() {
      cout << "Derived" << endl;
    }
};
void f() { throw Derived(); }
int main() {
  try {
    f();
  } catch(Base b) {
    b.what();
  }
  try {
    f();
  } catch(Base& b) {
    b.what();
  }
} ///:~
```

Output of the above program is as follows:

```
Base
Derived
```

The output of the program is as shown above because, when the object is caught by value, it is turned into a *Base* object (by the copy constructor) and must behave that way in all situations, whereas when it's caught by reference, only the address is passed and the object isn't truncated, so it behaves like what it really is, a *Derived* in this case. Although you can also throw and catch pointers, by doing so you introduce more coupling – the thrower and the catcher must agree on how the exception object is allocated and cleaned up. This is a problem because the exception itself may have occurred from heap exhaustion. If you throw exception objects, the exception-handling system takes care of all storage.

- Don't cause exceptions in destructors – Because destructors are called in the process of throwing other exceptions, you'll never want to throw an exception in a destructor or cause another exception to be thrown by some action you perform in the destructor. If this happens, it means that a new exception may be thrown *before* the catch-clause for an existing exception is reached, which will cause a call to *terminate()*. This means that if you call any functions inside a destructor that may throw exceptions, those calls should be within a *try* block in the destructor, and the destructor must handle all exceptions itself. None must escape from the destructor.

Self Assessment Questions

13. The asynchronous events can be handled by C++ exceptions. (True/False)
14. We should check the standard library before throwing an exception. (True/False)

14.13 Summary

- Exceptions are run time anomalies or unusual conditions that a program may encounter while executing.
- There are two kinds of exceptions i.e. synchronous and asynchronous exceptions

- The exception handling mechanism in C++ is built upon the three keywords named as try, throw and catch.
- When an exception that is desired to be handled is detected, it is thrown using the “throw” statement.
- We include code for handling exceptions in catch blocks.
- Sometimes program segment has more than one condition to throw an exception. In such cases you can associate more than one catch blocks with a try block.
- It is possible to create your own exception class and use them as exception types in your exception handling code instead of using pre-defined data types in C++ as exception.
- There are two basic models in exception-handling theory. In termination (which is what C++ supports) you assume the error is so critical there’s no way to get back to where the exception occurred.
- The alternative is called resumption. It means the exception handler is expected to do something to rectify the situation, and then the faulting function is retried, presuming success the second time.
- C++ provides us the facility to inform the user what exceptions are thrown by the function, so it can be handled by the user. This is the exception specification and it is part of the function declaration, appearing after the argument list.
- Sometimes you may want to rethrow the exception that you just caught. This is accomplished by using throw with no argument.
- The exception is uncaught if no handler catches the exception at any level
- An uncaught exception also occurs if a new exception is thrown before an existing exception reaches its handler.
- The programmers should follow certain guidelines while dealing with the exceptions.

14.14 Terminal Questions

1. Define exception. Explain exception handling mechanism.
2. Explain the implementation of multiple catch statements with the help of an example.

3. Discuss user-defined exception classes.
4. Compare termination and resumption
5. What is rethrowing of exception?
6. Describe the standard exceptions in C++ language.
7. Describe all the situations when we should avoid or use the exceptions.

14.15 Answers

Self Assessment Questions

1. Exceptions
2. Synchronous, Asynchronous
3. try
4. catch block
5. throw
6. catch(type arg)
   {
   //statements to manage exceptions
   }
7. User defined exception classes
8. termination
9. throw
10. unexpected()
11. no argument
12. set_terminate()
13. False
14. True

Terminal Questions

1. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. The exception handling mechanism in C++ is built upon the three keywords named as try, throw and catch. For more details refer section 14.3.
2. Sometimes program segment has more than one condition to throw an exception. In such cases you can associate more than one catch blocks with a try block. For more details refer section 14.5.
3. It is possible to create your own exception class and use it as exception type in your exception handling code instead of using pre-defined data

types in C++ as exception. You can achieve further control over exception handling by defining your own exception classes. For more details refer section 14.6.

4. There are two basic models in exception-handling theory. In termination (which is what C++ supports) you assume the error is so critical there’s no way to get back to where the exception occurred. Whoever threw the exception decided that there was no way to salvage the situation, and they don’t want to come back. The alternative is called resumption. For more details refer section 14.7.
5. Sometimes you may want to rethrow the exception that you just caught, particularly when you use the ellipses to catch any exception because there is no information available about the exception. This is accomplished by saying throw with no argument. For more details refer section 14.9.
6. The set of exceptions used with the Standard C++ library are also available for your own use. For more details refer section 14.11.
7. The exceptions should not be used for the following: Not for asynchronous events, not for ordinary error conditions, not for flow-of-control, you are not forced to use exceptions New exceptions, old code. For more details refer section 14.12.

References:

- Object Oriented Programming with C++ - Sixth Edition,
  by E Balagurusamy. Tata McGraw-Hill Education.
- The C++ Standard Library: A Tutorial and Handbook, By Nicolai M. Josuttis, Addison-Wesley Professional.
- Object-Oriented Programming with C++ 2Nd Ed. By Sarang, Sarang Poornachandra, PHI Learning Pvt. Ltd.