

```

    outfile.close();
    return 0; // successful copy
}
void main( int argc, char *argv[] )
{
    cout<< "cp - Copy file, Copyright (C) 1996, RAJ, C-DAC, Bangalore.\n";
    if( argc < 3 )
    {
        cout << "Usage: cp <source file> <destination file>";
        exit( 1 );
    }
    if( CopyFile( argv[1], argv[2] ) != 0 )
        cout << "\nfile copy operation failed.";
}

```

Run1

cp - Copy file, Copyright (C) 1996, RAJ, C-DAC, Bangalore.
Usage: cp <source file> <destination file>

Run2

cp - Copy file, Copyright (C) 1996, RAJ, C-DAC, Bangalore.
Error: noname.cpp non-existent
file copy operation failed.

Run3

cp - Copy file, Copyright (C) 1996, RAJ, C-DAC, Bangalore.

The arguments passed at the command line for the above three executions are as follows:

Run1: cp

Run2: cp noname.exe name.exe

Run3: cp cp.cpp temp.cpp

In main(), the statements

```

    fstream infile; // source file
    fstream outfile; // destination file

```

create two objects infile and outfile of the class fstream. They can be used either to read or write to the disk. The statement

```

    infile.open( SourceFile, ios::in | ios::binary );

```

opens SourceFile in binary read mode and assigns the handle to the object infile. Whereas, the statement

```

    outfile.open( DestinationFile, ios::out | ios::binary );

```

opens DestinationFile in binary write mode and assigns the handle to the object outfile.

The statement

```

    infile.read( (char *) buff, BUFSIZE );

```

reads the BUFSIZE number of characters from the infile into the variable buff, and the statement

```

    outfile.write( (char *) buff, infile.gcount() );

```

writes the number of characters that are read (gcount() returns the count of the number of characters read successfully) from the input file into the destination disk file.

The statement

```
if( infile.gcount() < BUFFSIZE )
```

checks whether the number of characters read from the input file is less than the requested number. If yes, it indicates that the input file has no more characters to be read and terminates the reading process.

The statements

```
infile.close();
outfile.close();
```

close both the input and output files from further processing.

Review Questions

- 18.1 What is a file ? What are the steps involved in manipulating a file in a C++ program ?
- 18.2 Explain the various file stream classes needed for file manipulations ?
- 18.3 Describe different methods of opening a file. Write a program to open a file named "xxx.bio" and write your name and other details into that file.
- 18.4 What are the different types of errors that might pop-up while processing files ?
- 18.5 Write an interactive program that accepts student's score and prints the result to a file.
- 18.6 Explain how `while (input_file)` expression detects the end of a file ?
- 18.7 What are file modes ? Describe various file mode options available ?
- 18.8 The file open modes `ios::app` and `ios::ate` set file pointer to end-of-file. What then, is the difference between them ?
- 18.9 What are file pointers ? Describe get-pointers and put-pointers.
- 18.10 What are the differences between sequential and random files ?
- 18.11 What are the differences between ASCII and binary files ?
- 18.12 Write a program which copies the contents of one file to a new file by removing unnecessary spaces between words.
- 18.13 Create a class called `student`. This class should have overloaded stream operator functions to save or retrieve objects of the `student` class from a file. Write an interactive program to manipulate objects of the `student` class with a file.
- 18.14 What are filter-utilities ? Write a program to display files on the screen page-wise. The output must pause after every page and continue until carriage return (enter) key is pressed. Accept name of a file to be processed from the command-line.
- 18.15 Explain how memory buffers can be connected to stream objects.
- 18.16 Write an interactive program to maintain an employee database. It has to maintain information such as employee id, name, qualification, designation, salary, etc. The user must be able to access all details about a person either by entering employee name or by employee id. Note that request for information may come randomly. It has to support an option for creating, updating, and deleting a database (in addition to query).

Exception Handling

19.1 Introduction

The increase in complexity and size of the software systems and the increase in society's dependence on the computer systems have been accompanied by an increase in the costs associated with their failure. The rising cost of failure in a computer system has stimulated interest in improving software reliability.

Software does not degrade physically as a function of time or environmental stress. It was assumed earlier that the concepts such as reliability or failure rate were not applicable to computer programs. It is true that a program that has once performed a given task as specified will continue to do so provided that none of the following change: the input, the computing environment, or user requirements. However, it is not reasonable to expect a program to be constantly operating on the same input data, because changes in computing environment and user requirements must be accommodated in most of the applications. Past and current failure free operation cannot be taken as a dependable indication that there will be no failure in the future.

The two main techniques for building reliable software (for dependable computing) are fault avoidance and fault tolerance. *Fault avoidance* deals with the prevention of fault occurrence by *construction*. It emphasizes on techniques to be applied during system development to ensure that the running system satisfies all reliability criteria *a priori*. It emphasizes that a sound way to deal with design faults is to stop them from getting into the system in the first place. *Fault tolerance* deals with the method of providing services complying with the specification in spite of faults having occurred (or occurring) by *redundancy*. In C++, exception handling allows to build fault tolerant systems.

Fault tolerance approach attempts to increase reliability by designing the system to continue to provide service in spite of the presence of faults. It begins with error detection. It must be possible to detect the occurrence of a latent error before it leads to failure. Once an error has been detected, the goal is error recovery. The goal of fault tolerant design is to improve dependability by enabling the system to perform its intended function in the presence of a given number of faults.

The Annotated C++ Reference Manual (ARM) by Ellis and Stroustrup states *Exception handling provides a way of transferring control and information to an unspecified caller that has expressed willingness to handle exceptions of a given type. Exceptions of arbitrary types can be 'thrown and caught' and the set of exceptions a function may throw can be specified. The termination model of exception handling is provided. Exception handling can be used to support notions of error handling and fault tolerant computing.*

19.2 Error Handling

In traditional programming techniques, validation of input data and some runtime errors were handled explicitly by the module in which the error occurred. Although, the users of these modules know how to

cope with such errors, there is no means to detect the errors and handle them in the user's code instead of the library. The notion of exceptions is supported in C++ to deal with such problems. Here, *exception* refers to unexpected condition in a program. The unusual conditions could be faults, causing an error which in turn causes the program to fail. The error-handling mechanism of C++ is generally referred to as *exception handling*. It provides a straightforward mechanism for adding reliable error handling mechanism in a program.

Generally, exceptions are classified into *synchronous* and *asynchronous exceptions*. The exceptions which occur during the program execution, due to some fault in the input-data or technique that is not suitable to handle the current class of data, within the program, are known as *synchronous exceptions*. For instance, errors such as out-of-range, overflow, underflow, and so on belong to the class of synchronous exceptions. The exceptions caused by events or faults unrelated (external) to the program and beyond the control of program are called *asynchronous exceptions*. For instance, errors such as keyboard interrupts, hardware malfunctions, disk failure, and so on belong to the class of asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions caused within a program.

Exception handling is an integral part of the ANSI/ISO C++ language standard. This standardization ensures that the power of object-oriented design is supported throughout the program. An especially strong feature of the standard is the availability of virtual functions and the use of objects to define exceptions. Virtual functions guarantee a minimum runtime overhead—zero additional program overhead if no exceptions are thrown. When used properly, C++ exception handling solves many problems with alternative error handling techniques (such as returning error values from methods or using global error handlers).

In accordance with ANSI specifications, recent implementation of most C++ compilers are supporting the exception-handling model. When an abnormal situation arises at runtime, the program should terminate. However, throwing an exception allows the user to gather information at the throw point that could be useful in diagnosing the causes which led to failure. An user can also specify in the exception handler the actions to be taken before the program terminates. Only synchronous exceptions are handled (the cause of failure is generated from within the program). An event such as Control-C (which is generated from outside the program) is not considered to be an exception.

19.3 Exception Handling Model

When a program encounters an abnormal situation for which it is not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception. The exception-handling mechanism uses three blocks: *try*, *throw*, and *catch*. The relationship of these three exception handling constructs called the *exception handling model* is shown in Figure 19.1.

The *try-block* must be followed immediately by a handler, which is a *catch block*. If an exception is *thrown* in the *try-block*, the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so could result in abnormal termination of the program. Though C++ allows an exception to be of any type, it is useful to make exceptions as objects. The exception object is treated exactly the same way as other normal objects. An exception carries information from the point where the exception is thrown to the point where the exception is caught. This information allows the program user to know as to when the program encounters an anomaly at runtime.

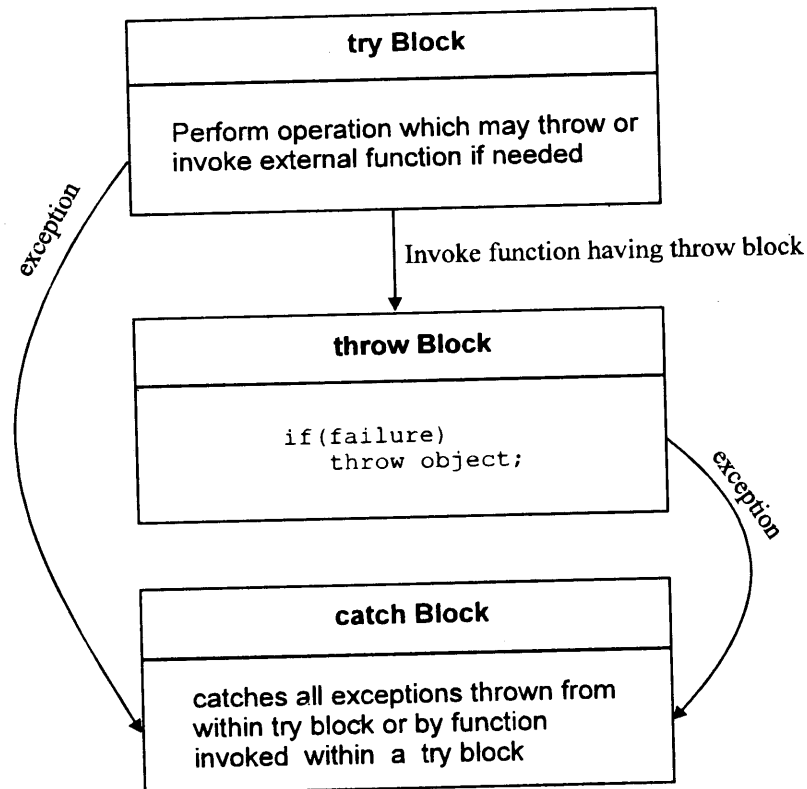


Figure 19.1: Exception handling model

19.4 Exception Handling Constructs

Exception handling mechanism transfers control and information from a point of exception in a program to an exception handler associated with the *try-block*. An exception handler will be invoked only by a *thrown expression* in the code executed by the handler's try-block or by functions called from the handler's try-block. C++ offers the following three constructs for defining these blocks.

- ◆ try
- ◆ throw
- ◆ catch

The exception handler is indicated by the `catch` keyword. The handler must be used immediately after the `try-block`. The keyword `catch` can also occur immediately after another `catch`. Each handler will only evaluate an exception that matches, or can be converted to the type specified in its argument list. Every exception thrown by the program must be caught and processed by the exception handler. If the program fails to provide an exception handler for a thrown exception, the program will call the `terminate()` function.

Exception handlers are evaluated in the order they are encountered. An exception is said to be caught when its type matches the type in the `catch` statement. Once a type match is made, program

control is transferred to the handler. The handler specifies what actions should be taken to deal with the program anomaly. The *stack-unwinding* (catch-cleanup) operation is initiated immediately after processing the *catch block* that matches with the exception type. In normal sequence (no exceptions are raised) stack-unwinding is performed immediately after the *try-block* and program execution continues. (A `goto` statement can be used to transfer program control out of a handler but such a statement can never be used to enter a handler.) After the handler has been executed, the program continues its execution from the point after the last handler for the current try-block and no other handlers are evaluated for the current exception.

throw Construct

The keyword `throw` is used to raise an exception when an error is generated in the computation. The throw-expression initializes a temporary object of the type `T` (to match the type of argument `arg`) used in `throw(T arg)`. The syntax of the throw construct is shown in Figure 19.2.

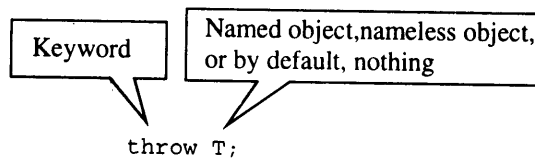


Figure 19.2: Syntax of throw construct

catch Construct

The exception handler is indicated by the `catch` keyword. It must be used immediately after the statements marked by the `try` keyword. The `catch` handler can also occur immediately after another `catch`. Each handler will only evaluate an exception that matches, or can be converted to the type specified in its argument list. The syntax of the `catch` construct is shown in Figure 19.3.

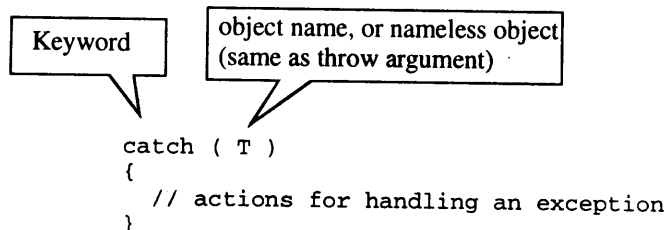
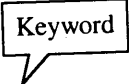


Figure 19.3: Syntax of catch construct

try Construct

The `try` keyword defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword `try`. Following the `try` keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The syntax of the `try` construct is shown in Figure 19.4.



```

try
{
    // code raising exception or referring to
    // a function raising exception
}
catch( type_id1 )
{
    // actions for handling an exception
}
...
...
catch( type_idn )
{
    // action for handling an exception
}

```

Figure 19.4: Syntax of try construct

A block of code in which an exception can occur must be prefixed by the keyword `try`. The `try` keyword is followed by a block of code enclosed within braces. It indicates that the program is prepared for testing the existence of exceptions. If an exception occurs, the program flow is interrupted and the exception handler is invoked.

The mechanism suggests that error handling code must perform the following tasks.

1. Detect the problem causing exception (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 exceptions)

Exception handling code resembles the following pattern:

```

my_function()
{
    .....
    if( operation_fail )
        throw Object1; // throw-point
    .....
}
....
try
{
    // begin of try block
    .....
    my_function(); // call the function my_function
    .....
    if( overflow )
        throw Object2; // throw-point
    .....
}
    // end of try block

```

```

catch( Object1 )
{
    .....
    // take corrective action for operation_fail
    .....
}
catch( Object2 )
{
    .....
    // take corrective action for overflow
    .....
}
....

```

The following sequence of steps are performed when an exception is raised:

- ◆ The program searches for a matching handler.
- ◆ If a handler is found, the stack is unwound to that point.
- ◆ Program control is transferred to the handler.
- ◆ If no handler is found, the program will invoke the `terminate()` function (explained later). If no exceptions are thrown, the program executes in the normal fashion.

The program `divzero.cpp` illustrates the mechanism for detecting errors, raising exceptions, and handling such exceptions. It has the class `number` to store an integer number and the member function `read()` to read a number from the console and the member function `div()` to perform division operations. It raises exception if an attempt is made to perform *divide-by-zero* operation. It has an empty class named `DIVIDE` used as the throw's expression-id.

```

// divzero.cpp: Divide Operation Validation, (divide-by-zero)
#include <iostream.h>
class number
{
    private:
        int num;
    public:
        void read()    // read number from keyboard
        {
            cin >> num;
        }
        class DIVIDE {};    // abstract class used in exceptions
        int div( number num2 )
        {
            if( num2.num == 0 )    // check for zero division if yes
                throw DIVIDE();    // raise exception
            else
                return num / num2.num;    // compute and return the result
        }
};

```



```

int main()
{
    number num1, num2;
    int result;
    cout << "Enter Number 1: ";
    num1.read();
    cout << "Enter Number 2: ";
    num2.read();
    // statements must be enclosed in try block if you intend to handle
    // exceptions raised by them
    try
    {
        cout << "trying division operation...";
        result = num1.div( num2 );
        cout << "succeeded" << endl;
    }
    catch( number::DIVIDE )    // exception handler block
    {
        // actions taken in response to exception
        cout << "failed" << endl;
        cout << "Exception: Divide-By-Zero";
        return 1;
    }
    // no exceptions, display result
    cout << "num1/num2 = " << result;
    return 0;
}

```

Run1

```

Enter Number 1: 10
Enter Number 2: 2
trying division operation...succeeded
num1/num2 = 5

```

Run2

```

Enter Number 1: 10
Enter Number 2: 0
trying division operation...failed
Exception: Divide-By-Zero

```

In main(), the try-block

```

    try
    { ...; result = num1.div( num2 ); ...; }

```

invokes the member function `div()` to perform the division operation. If any attempt is made to divide by zero, the following statement in `div()`:

```

    if( num2.num == 0 ) // check for zero division if yes
        throw DIVIDE(); // raise exception

```

detects the same and raises the exception by passing a nameless object of type class `DIVIDE`. All the statements following the one which raised the exception are skipped (see output of **Run2** above) and search for an exception handler begins. The runtime system searches catch-block to detect the handler.

The block of code in `main()` following the `try`-block:

```
catch( number::DIVIDE )
{
    cout << "Exception: Divide-By-Zero";
    return 1;
}
```

will catch the exception raised due to the call to the function in the *try*-block and executes its body (see Figure 19.5). If no exception is raised, the exception handling *catch*-block will not be executed and execution proceeds to the next statement, which displays the result.

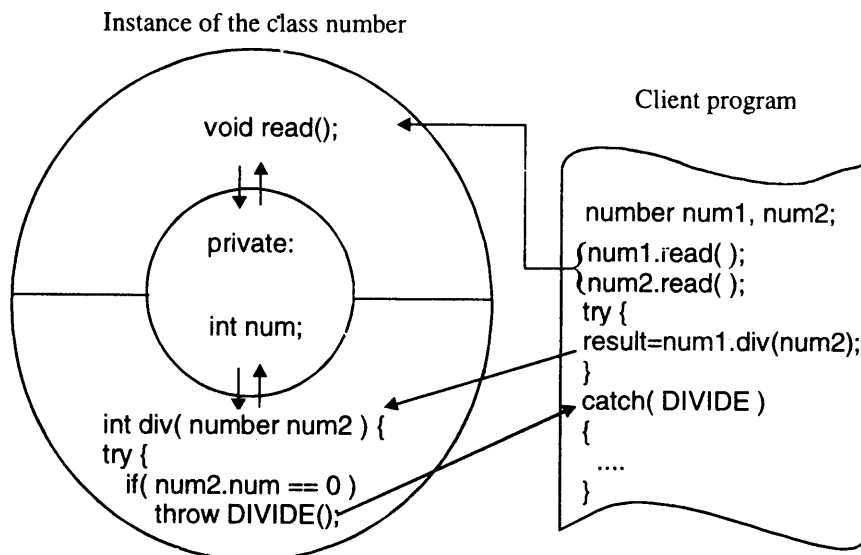


Figure 19.5: Exception handling in the number class

Array Reference Out of Bound

The program `arrbound.cpp` illustrates the mechanism of validating array element references. If any attempt is made to refer to an element whose index is beyond the array size, an exception is raised.

```
// arrbound.cpp: Array Reference Bound Validation
#include <iostream.h>
const int ARR_SIZE = 10; // maximum array size
class array
{
    private:
        int arr[ARR_SIZE];
    public:
        class RANGE {}; // Range abstract class
        int & operator[]( int i )
        {
            if( i < 0 || i >= ARR_SIZE )
                throw RANGE(); // throw abstract object
```

```

        return arr[i]; // valid reference
    }
};
void main()
{
    array a; // create array
    cout << "Maximum array size allowed = " << ARR_SIZE << endl;
    try
    {
        cout << "Trying to refer a[1]...";
        a[1] = 10;
        cout << "succeeded" << endl;
        cout << "Trying to refer a[15]...";
        a[15] = 10; // refer 15th element from array a, causes exception
        cout << "succeeded" << endl;
    }
    catch( array::RANGE ) // true if throw is executed in try scope
    {
        // action for exception
        cout << "Out of Range in Array Reference";
    }
}

```

Run

```

Maximum array size allowed = 10
Trying to refer a[1]...succeeded
Trying to refer a[15]...Out of Range in Array Reference

```

The statement in try-block of main():

```
a[1] = 10;
```

updates the first element of the array. However, another statement

```
a[15] = 10;
```

in the same block, tries to update the fifteenth element. It leads to an exception since the array size is only 10. This exception is caught by the statement

```
catch( array::RANGE )
```

which issues a warning message on the standard output.

19.5 Handler Throwing the Same Exception Again

There are several good reasons to allow an exception to be implicitly propagated from a function (callee) to its caller. Of course, it follows the *democracy* principle: a client (caller) is the better candidate to decide what actions are to be taken when something goes wrong. If a function does not want to take any corrective action in response to an exception, it can pass the same to the caller of a function. The `throw` construct without an explicit exception parameter raises the previous exception. An exception must currently exist otherwise, `terminate()` is invoked. The program `pass.cpp` illustrates the method of passing the same exception to the caller if the current handler is unable to handle it.

712 **Mastering C++**

```
// pass.cpp: passing all exceptions that occur in parent to child
#include <iostream.h>
#include <process.h>
const int ARR_SIZE = 10;    // maximum array size
class array
{
private:
    int arr[ARR_SIZE];
public:
    array();
    class RANGE {};        // Range abstract class
    int & operator[]( int i )
    {
        if( i < 0 || i >= ARR_SIZE )
            throw RANGE(); // throw abstract object
        return arr[i];     // valid reference
    }
};
array::array()
{
    for( int i = 0; i < ARR_SIZE; i++ )
        arr[i] = i;
}
// read an element from the array, if any exception pass the same to caller
int read( array & a, int index )
{
    int element;
    try
    {
        element = a[ index ];
    }
    catch(array::RANGE) // catch the exceptions raised in class
    {
        cout << endl<< "Parent passing exception to child to handle"<<endl;
        throw;        // pass all exceptions to the caller
    }
    return element;
}
void main()
{
    array a; // create array object
    int index, element;
    cout << "Maximum vector size allowed = " << ARR_SIZE << endl;
    while( 1 )
    {
        cout << "Enter element to referenced: ";
        cin >> index;
        try
        {
            cout << "Trying to access object array 'a' for index = "<<index;
            element = read( a, index );
        }
    }
}
```

```

        cout << endl << "Element in Array = " << element << endl;
    }
    catch( array::RANGE ) // true if throw is executed in try scope
    {
        // action for exception
        cout << "Child: Out of Range in Array Reference";
        exit( 1 );
    }
}
}

```

Run

```

Maximum vector size allowed = 10
Enter element to referenced: 1
Trying to access object array 'a' for index = 1
Element in Array = 1
Enter element to referenced: 5
Trying to access object array 'a' for index = 5
Element in Array = 5
Enter element to referenced: 10
Trying to access object array 'a' for index = 10
Parent passing exception to child to handle
Child: Out of Range in Array Reference

```

The *catch-block* in the function `read()` does not take any corrective action for the exception `array::RANGE`. It throws the exception to the caller and the *catch-block* in `main()` terminates the program after displaying the message:

```

Child: Out of Range in Array Reference

```

on the standard output device.

19.6 List of Exceptions

Raising or catching an exception affects the way a function relates to other functions. C++ language makes it possible for the user to specify a list of exceptions that a function can throw. This exception specification can be used as a suffix to the function declaration specifying the list of exceptions that a function may directly or indirectly throw as a part of a function declaration. The syntax for exception specification is shown in Figure 19.6.

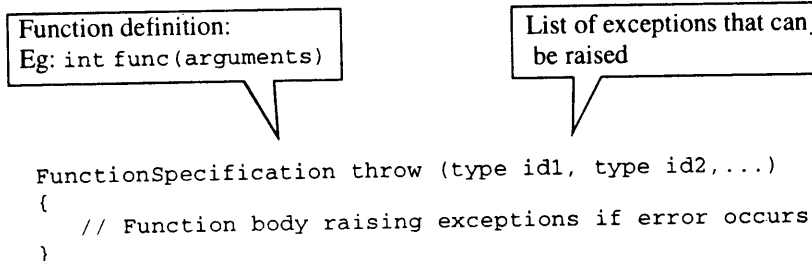


Figure 19.6: Syntax of specifying a list of exceptions

The *exception-list*, which is the function suffix is not considered to be a part of the specification of a function. Consequently, a pointer to a function is not affected by the function's exception specification. Such a pointer checks only the function's return value and argument types. Therefore, the following is legal:

```
void f1(void) throw();            // cannot throw exceptions
void f2(void) throw (BETA); // can throw BETA objects
int func() throw( X, Y )        // can throw only X and Y exceptions
{...
;
;
```

C++ allows to have pointers to a function raising exception, for instance,

```
void (* fptr)();                // Pointer to a function returning void
fptr = f1;
fptr = f2;
```

However, extreme care should be taken when overriding virtual functions; the exception specification is not considered as a part of the function type, it is possible to violate the program design. If an exception which is not listed in the exception specification is *thrown*, the function `unexpected()` will be called (discussed later in this chapter).

In the following example, the derived class `BETA::vfunc` is defined so that it should not throw any exceptions—a departure from the original function declaration.

```
class ALPHA
{
public:
struct ALPHA_ERR {};
virtual void vfunc(void) throw (ALPHA_ERR) {};
// Exception specification
};
class BETA : public ALPHA
{
void vfunc(void) throw() {}; // Exception specification is changed
};
```

The following are examples of functions with exception specifications.

```
void f1();                        // The function can throw any exception
void f2() throw();               // Should not throw any exceptions
void f3() throw( A, B* );        // Can throw exceptions publicly derived
// from A, or a pointer to publicly derived B
```

Raising an Unspecified Exception

The definition and all declarations of such a function must have an exception specification containing the same set of type-ids. If a function throws an exception not listed in its specification, the program will call the function `unexpected()`. This is a runtime issue and it will not be flagged at compile time. Therefore, care must be taken to handle any exception which can be thrown by statements/functions invoked within a function.

```
void my_func1() throw (A, B)
{
// Body of function.
}
```

This example specifies a list of exceptions that `my_func1()` can throw. No other exception will propagate out of `my_func1`. If an exception other than A or B is generated within `my_func1`, it is considered to be an unexpected exception and program control will be transferred to the predefined unexpected function. The program `sign1.cpp` illustrates raising of an exception other than that specified in the exception list..

```
// sign1.cpp:determine whether the input is +ve or -ve through exceptions
#include <iostream.h>
class positive {};
class negative {};
class zero {};
// this function can raise only positive and negative exceptions
void what_sign( int num ) throw( positive, negative )
{
    if( num > 0 )
        throw positive();
    else
        if( num < 0 )
            throw negative();
        else
            throw zero(); // unspecified exception
}
void main()
{
    int num;
    cout << "Enter any number: ";
    cin >> num;
    try
    {
        what_sign( num );
    }
    catch( positive )
    { cout << "+ve Exception"; }
    catch( negative )
    { cout << "-ve Exception"; }
    catch( zero )
    { cout << "0 Exception"; }
}
```

Run1

Enter any number: 10
+ve Exception

Run2

Enter any number: -10
-ve Exception

Run3

Enter any number: 0
Abnormal program termination

The prototype of the function `what_sign()` is specified as

```
void what_sign( int num ) throw( positive, negative )
```

It indicates that, this function can raise exceptions `positive` and `negative`, but the statement

```
throw zero(); // unspecified exception
```

raises the exception `zero`, which is not in the exception list of this function. It calls the default exception handler, which aborts the execution of the program (see **Run3**) although there exists an explicit exception handler in the caller of this function.

Exceptions in a No-Exception Function

The following function and exception specification indicates that it will not generate any exception:

```
void my_func2() throw ()
{
    // Body of this function.
}
```

If any statement in the body of `my_func2()` throws an exception, the control is transferred to library function `abort()`, which terminates the program by issuing an error message. The program `sign2.cpp` illustrates the effect of raising an exception in a function which is not supposed to raise any exception.

```
// sign2.cpp: determine whether the input is positive or negative
#include <iostream.h>
class zero {};
// this function cannot raise exception
void what_sign( int num ) throw()
{
    if( num > 0 )
        cout << "+ve number";
    else
        if( num < 0 )
            cout << "-ve number";
        else
            throw zero(); // unspecified exception
}
void main()
{
    int num;
    cout << "Enter any number: ";
    cin >> num;
    try
    {
        what_sign( num );
    }
    catch( zero )
    { cout << "0 Exception"; }
}
```

Run1

```
Enter any number: 10
+ve number
```


Run2

Enter any number: -10
-ve number

Run3

Enter any number: 0
Abnormal program termination

The prototype of the function `what_sign()`:

```
void what_sign( int num ) throw()
```

indicates that it does not raise any exception, but the statement

```
throw zero(); // unspecified exception
```

raises the exception. It invokes the default exception handler which aborts the execution of the program (see *Run3*) though there exists an explicit exception handler in the caller of this function.

19.7 Catch All Exceptions

C++ supports a feature to catch all the exceptions raised in the try-block. The syntax of the catch construct to handle all the exceptions raised in the try block is shown in Figure 19.7.

three dots: indicate catch
all exceptions

```
catch ( . . . )
{
    // actions for handling an exception
}
```

Figure 19.7: Syntax of catch all construct

The three dots in the `catch(...)` indicates that it catches all types of exceptions raised in its preceding try-block. The program `catall1.cpp` illustrates the mechanism of handling all the exceptions raised by a single handler.

```
// catall1.cpp: All exceptions are caught
#include <iostream.h>
class excep2 {};
void main()
{
    try
    {
        cout << "Throwing uncaught exception" << endl;
        throw excep2();
    }
    catch( ... ) // catch all the exceptions
    {
        // action for exception
        cout << "Caught all exceptions" << endl;
    }
}
```

718 Mastering C++

```
    }  
    cout << "I am displayed";  
}
```

Run

Throwing uncaught exception
Caught all exceptions
I am displayed

The statement in the try-block of main():

```
    throw excep2();
```

raises the exception excep2(). It is caught by the statement,

```
    catch( ... ) // catch all the exceptions
```

The program having multiple catch-all exceptions is illustrated in `catall2.cpp`. It has multiple functions calling one another.

```
// catall2.cpp: making exception-specifications and handle all exceptions  
#include <iostream.h>  
class ALPHA{}; // Exception declaration  
ALPHA _a; // object of ALPHA  
void f3(void) throw (ALPHA)  
{  
    // Will throw only type-alpha objects  
    cout << "f3() was called" << endl;  
    throw( _a ); // throw exception explicit object  
}  
void f2(void) throw()  
{  
    // should not throw exceptions  
    try  
    {  
        // wrap all code in a try-block  
        cout << "f2() was called" << endl;  
        f3();  
    }  
    catch ( ... )  
    {  
        // trap all exceptions  
        cout << "f2() has elements with exceptions!" << endl;  
    }  
}  
int main()  
{  
    try  
    {  
        f2();  
        return 0; // f2 succeeds, terminate  
    }  
    catch( ... )  
    {  
        cout << "Need more handlers!";  
    }  
}
```

```

    cout << endl << "continued after handling exceptions";
    return 1;
}

```

Run

```

f2() was called
f3() was called
f2() has elements with exceptions!

```

In `f3()`, the statement

```

    throw( _a ); // throw exception explicit object

```

throws the exception using named object `_a`, which is the instance of the class `ALPHA`. It is caught by the handler in the caller function `f2()`. There is a handler to catch all exceptions in `main()`, but is not activated; all the exceptions are caught in `f2()` and no exceptions are passed to its caller.

19.8 Exceptions in Constructors and Destructors

When an exception is thrown, the copy constructor is invoked as a part of the exception handling. The copy constructor is used to initialize a temporary object at the throw point. Other copies may be generated by the program. When the program flow is interrupted by an exception, destructors are invoked for all automatic objects which were constructed from the entry point of the try-block. If the exception was thrown during construction of some object, destructors will be called only for those objects which were fully constructed. For example, if an array of objects was under construction when an exception was thrown, destructors will be called only for the array elements which were fully constructed.

As a building block of design patterns for proper handling of exceptions, there is a need for *secure operations* that allow transfer of resource responsibilities without throwing exceptions. In C++, it is a bad idea to leave a destructor by throwing an exception. This is because a destructor may be invoked during runtime stack unwinding when another exception was thrown; a second throw that aborts one of these destructors will immediately invoke `terminate()`, which aborts the program by default. In other words, all destructors in a C++ program should have an empty specification `throw()`. This is called *secure operations*.

Those objects which are created from a try-block to any statement raising an exception serve no purpose if any exception is raised. Hence, they must be destroyed by releasing the allocated resources. The process of calling destructor for automatic objects constructed on the path from a try-block to a thrown expression is called *stack unwinding*. The program `twoexcep.cpp` illustrates the concept of having multiple types of exceptions in a program.

```

// twoexcep.cpp: Array Creation and Reference Bound Validation
#include <iostream.h>
const int ARR_SIZE = 10; // maximum array size, that can be allocated
class array
{
private:
    int *arr; // pointer to array
    int size; // maximum array size

```

```

public:
class SIZE {};    // Size abstract class
class RANGE {};    // Range abstract class
array( int SizeRequest )    // constructor
{
    if( SizeRequest < 0 || SizeRequest > ARR_SIZE )
        throw SIZE();
    // allocate resources
    size = SizeRequest;
    arr = new int[ size ];
}
~array()    // destructor
{
    // deallocate resources
    delete arr;
}
int & operator[]( int i )    // subscript operator overloading
{
    if( i < 0 || i > size )
        throw RANGE(); // throw abstract object
    return arr[i]; // valid reference
}
};
void main()
{
    cout << "Maximum array size allowed = " << ARR_SIZE << endl;
    try
    {
        cout << "Trying to create object a1(5)...";
        array a1(5);    // create array
        cout << "succeeded" << endl;
        cout << "Trying to refer a1[5]...";
        a1[5] = 10;
        cout << "succeeded..";
        cout << "a1[5] = " << a1[5] << endl;
        cout << "Trying to refer a1[15]...";
        a1[15] = 10;    // causes exception
        cout << "succeeded" << endl;
    }
    catch( array::SIZE )
    {
        // action for exception
        cout << "...Size exceeds allowable Limit" << endl;
    }
    catch( array::RANGE ) // true if throw is executed in try scope
    {
        // action for exception
        cout << "...Array Reference Out of Range" << endl;
    }
}

```

```

// Array creation unsuccessful, Request > ARR_SIZE
try
{
    cout << "Trying to create object a2(15)...";
    array a2(15); // create array, causes exception
    cout << "succeeded" << endl;
    a2[3] = 3; // valid access
}
catch( array::SIZE )
{
    // action for exception
    cout << "....Size exceeds allowable Limit" << endl;
}
catch( array::RANGE ) // true if throw is executed in try scope
{
    // action for exception
    cout << "....Array Reference Out of Range" << endl;
}
}

```

Run

```

Maximum array size allowed = 10
Trying to create object a1(5)...succeeded
Trying to refer a1[5]...succeeded..a1[5] = 10
Trying to refer a1[15]....Array Reference Out of Range
Trying to create object a2(15).....Size exceeds allowable Limit

```

The one-argument constructor of the class array,
`array(int SizeRequest) // constructor`
throws an exception,
`throw SIZE();`

if an attempt is made to create an array beyond the allowable range. The statement

```

if( i < 0 || i > size )
    throw RANGE(); // throw abstract object

```

throws an exception if an attempt is made to access an array element by using invalid index (lower than minimum bound or higher than the maximum bound).

19.9 Handling Uncaught Exceptions

The uncaught exception handling mechanism relies on two library functions, `terminate()` and `unexpected()`, for coping with exceptions unhandled explicitly. C++ supports the following special functions to handle uncaught exceptions in a systematic manner:

- ◆ `terminate()`
- ◆ `set_terminate()`
- ◆ `unexpected()`
- ◆ `set_unexpected()`

terminate()

The function `terminate()` is invoked when an exception is raised and the handler is not found. The

default action for `terminate` is to invoke `abort()`. Such a default action causes immediate termination of the program execution. The program `uncaught.cpp` illustrates the series of events that can occur when the program encounters an exception for which no handler can be found.

```
// uncaught.cpp: Uncaught exception invokes abort() automatically
#include <iostream.h>
class except1 {};
class except2 {};
void main()
{
    try
    {
        cout << "Throwing uncaught exception" << endl;
        throw except2();
    }
    catch( except1 ) // true if throw except1 is executed in try scope
    {
        // action for exception
        cout << "Exception 1";
    }
    // except2 is not caught hence, program aborts
    // here without proceeding further
    cout << "I am not displayed";
}

```

Run

```
Throwing uncaught exception
Abnormal program termination
```

The statement in `main()`'s try-block:

```
    throw except2();
```

raises an exception `except2` for which no handler exists. Here, `terminate()` comes to rescue this condition. When `terminate()` function is called, the program aborts by displaying the message,

```
Abnormal program termination
```

and does not proceed further.

The programmer can modify the way the program will terminate when an exception is generated. The `terminate()` function can call user defined function instead of `abort()` if the user defined function is registered with `set_terminate()` function.

set_terminate()

The `set_terminate` function allows the user to install a function that defines the program's actions to be taken to terminate the program when a handler for the exception cannot be found. The actions are defined in `t_func`, which is declared to be a function of type `terminate_function`. A `terminate_function` type defined in `except.h`, is a function that takes no arguments, and returns nothing. By default, an exception for which no handler can be found results in the program calling the `terminate` function. This will normally result in a call to `abort` function. The program then ends with the message, *Abnormal program termination*. If some function other than `abort()` is to be invoked by

the `terminate()`, the user should define `t_func` function. This `t_func` function can be installed by `set_terminate` as the termination function. The installation of `t_func` allows the user to implement any action that is not taken by `abort()`. The syntax of the `set_terminate` function declared in the header file `except.h` is as follows:

```
typedef void (*terminate_function)();
terminate_function set_terminate( terminate_function t_func );
// Define your termination scheme
terminate_function my_terminate( void )
{
    // Take actions before terminating
    // should not throw exceptions
    exit(1); // must end somehow
}
// Register your termination function
set_terminate( my_terminate );
```

The program `myhand.cpp` handles uncaught exceptions with the user specified terminate function.

```
// myhand.cpp: All exceptions are not caught, executes MyTerminate()
#include <iostream.h>
#include <except.h>
class except1 {};
class except2 {};
void MyTerminate()
{
    cout << "My Terminate is invoked";
    exit( 1 );
}
void main()
{
    set_terminate( MyTerminate ); // sets to our own terminate function
    try
    {
        cout << "Throwing uncaught exception\n";
        throw except2();
    }
    catch( except1 )
    {
        // action for exception
        cout << "Caught exception, except1\n";
    }
    // program abort() here; MyTerminate() will be called
    cout << "I am not displayed";
}
```

Run

```
Throwing uncaught exception
My Terminate is invoked
```

In `main()`, the statement

```
set_terminate( MyTerminate );
```

sets the function `MyTerminate` as a termination function to be invoked when there exists no exception handler for the exception raised. The statement in the try-block ,

```
throw excep2();
```

raises the exception `excep2`, which is uncaught. The system automatically invokes the function `MyTerminate` as a part of unhandled exceptions.

unexpected()

The `unexpected` function is called when a function throws an exception not listed in its exception specification. The program calls `unexpected()` which calls any user-defined function registered by `set_unexpected`. If no function is registered with `set_unexpected`, the `unexpected()` function then invokes the `terminate()` function. The prototype of the `unexpected()` call is

```
void unexpected();
```

The function `unexpected` returns nothing (void) but *it can throw an exception* through the execution of a function registered by the `set_unexpected` function.

```
// sign3.cpp: unexpected exceptions
#include <iostream.h>
#include <process.h> // has prototype for exit()
#include <except.h>
class zero {};
// this function cannot raise exception
void what_sign( int num ) throw()
{
    if( num > 0 )
        cout << "+ve number";
    else
        if( num < 0 )
            cout << "-ve number";
        else
            throw zero(); // unspecified exception
}
void main()
{
    int num;
    cout << "Enter any number: ";
    cin >> num;
    try
    {
        what_sign( num );
    }
    catch(...)
    {
        cout << "catch all exceptions";
    }
    cout << endl << "end of main()";
}
```


Run1

```
Enter any number: 10
+ve number
end of main()
```

Run2

```
Enter any number: -3
-ve number
end of main()
```

Run3

```
Enter any number: 0
Abnormal program termination
```

The function

```
void what_sign( int num ) throw()
```

raises an unspecified exception

```
throw zero(); // unspecified exception
```

leading to the invocation of the `unexpected()` function automatically (see *Run3*).

`set_unexpected()`

The function `set_unexpected()` lets the user to install a function that defines the program's actions to be taken when a function throws an exception not listed in its exception specification. The actions are defined in `unexpected_func()` library function. By default, an unexpected exception causes `unexpected()` to be called, which in turn calls `unexpected_func`.

Program behavior when a function is registered with `set_unexpected()`:

```
// Define your unexpected handler
unexpected_function my_unexpected( void )
{
    // Define actions to take
    // possibly make adjustments
}
// register your handler
set_unexpected( my_unexpected );
```

The program `sign4.cpp` illustrates the mechanism of defining the user defined unexpected exception handler. The user defined `unexpected_func` must not return to its caller. An attempt to return to the caller results in an undefined program behavior. The `unexpected_func()` can invoke `abort()`, `exit()`, or `terminate()` functions.

```
// sign4.cpp: unexpected exceptions through user-defined function.
#include <iostream.h>
#include <process.h> // has prototype for exit()
#include <except.h>

class zero {}; // empty class
// this function cannot raise exception
```

726 **Mastering C++**

```
void what_sign( int num ) throw()
{
    if( num > 0 )
        cout << "+ve number";
    else
        if( num < 0 )
            cout << "-ve number";
        else
            throw zero(); // unspecified exception
}
// this is automatically called whenever an unexpected exception occurs
void MyUnexpected()
{
    cout << "My unexpected handler is invoked";
    exit( 1 );
}
void main()
{
    int num;
    cout << "Enter any number: ";
    cin >> num;
    set_unexpected( MyUnexpected ); // user defined handler
    try
    {
        what_sign( num );
    }
    catch(...) // catch all exceptions
    {
        cout << "catch all exceptions";
    }
    cout << endl << "end of main()";
}
```

Run1

```
Enter any number: 10
+ve number
end of main()
```

Run2

```
Enter any number: -3
-ve number
end of main()
```

Run3

```
Enter any number: 0
My unexpected handler is invoked
```

The function `what_sign()` raises an unspecified exception,
`throw zero(); // unspecified exception`
leading to the invocation of the user defined `MyUnexpected()` automatically (see **Run3**).

19.10 Exceptions in Operator Overloaded Functions

The program `interact.cpp` illustrates the mechanism for handling exceptions in the vector class, while creating its objects and accessing its elements either for a read or write operation. It overloads the operator `[]` to simulate the array operations on the user defined data type.

```
// interact.cpp: interactive program raises exception for improper data
#include <iostream.h>
#include <process.h>
const int VEC_SIZE = 10; // maximum vector size, that can be allocated
class vector
{
private:
    int *vec; // pointer to array for vector elements
    int size; // maximum vector size
public:
    class SIZE {}; // Size abstract class
    class RANGE {}; // Range abstract class
    vector( int SizeRequest )
    {
        if( SizeRequest <= 0 || SizeRequest > VEC_SIZE )
            throw SIZE();
        size = SizeRequest;
        vec = new int[ size ];
    }
    ~vector() // destructor
    {
        delete vec;
    }
    // subscripted operator overloading
    int & operator[]( int i );
};
// subscripted operator overloading
int & vector::operator[]( int i )
{
    if( i < 0 || i >= size )
        throw RANGE(); // throw abstract object
    return vec[i]; // valid reference
}
void main()
{
    int size, data, index;
    cout << "Maximum vector size allowed = " << VEC_SIZE << endl;
    try
    {
        cout << "What is the size of vector you want to create: ";
        cin >> size;
        cout << "Trying to create object vector v1 of size = " << size;
        vector v1(size); // create vector
        cout << "...succeeded" << endl;
    }
}
```

```

    cout << "Which vector element you want to access (index): ";
    cin >> index;
    cout << "What is the new value for v1[ " << index << " ]: ";
    cin >> data;
    cout << "Trying to modify a1[ " << index << " ]...";
    v1[index] = data;
    cout << "succeeded" << endl;
    cout << "New Value of a1[ " << index << " ] = " << v1[index];
}
catch( vector::SIZE )
{
    // action for exception
    cout << "failed" << endl;
    cout << "Vector creation size exceeds allowable limit";
    exit( 1 );
}
catch( vector::RANGE ) // true if throw is executed in try scope
{
    // action for exception
    cout << "...failed" << endl;
    cout << "Vector reference out-of-range";
    exit( 1 );
}
}
}

```

Run1

```

Maximum vector size allowed = 10
What is the size of vector you want to create: 5
Trying to create object vector v1 of size = 5...succeeded
Which vector element you want to access (index): 2
What is the new value for v1[ 2 ]: 7
Trying to modify a1[ 2 ]...succeeded
New Value of a1[ 2 ] = 7

```

Run2

```

Maximum vector size allowed = 10
What is the size of vector you want to create: 5
Trying to create object vector v1 of size = 5...succeeded
Which vector element you want to access (index): 10
What is the new value for v1[ 10 ]: 2
Trying to modify a1[ 10 ]...failed
Vector reference out-of-range

```

Run3

```

Maximum vector size allowed = 10
What is the size of vector you want to create: 15
Trying to create object vector v1 of size = 15
Vector creation size exceeds allowable limit

```

Note:

Run1: All operations are valid, no exception is generated

Run2: Invalid vector reference, exception generated

Run3: Invalid size for vector creation, exception generated

In *Run2*, an attempt is made to refer to the 11th element (but index is 10) of the vector whose size is 10. It raises an exception, which is caught by the statement,

```
catch( vector::RANGE )
```

In *Run3*, an attempt is made to create the vector of size 15, but the allowable limit is 10 as restricted by the value of `VEC_SIZE` constant. The statement

```
catch( vector::SIZE )
```

catches the exception raised while creating objects of the vector class.

19.11 Exceptions in Inheritance Tree

The mechanism of handling exceptions in the base and derived classes is illustrated in `virtual.cpp`.

```
// virtual.cpp: Binding a pointer to base class' object to base or derived
// objects at runtime and invoking respective members if they are virtual
#include <iostream.h>
#include <process.h>
// empty class for Father and Son inheritance
class WRONG_AGE
{
};
class Father
{
protected:
    int f_age;
public:
    Father( int n )
    {
        if( n < 0 )
            throw WRONG_AGE();
        f_age = n;
    }
    virtual int GetAge(void)
    {
        return f_age;
    }
};
// Son inherits all the properties of father
class Son : public Father
{
protected:
    int s_age;
public:
    Son( int n, int m ):Father(n)
    {
        // if son's age is greater or equal to father, throw exception
        if( m >= n )
            throw WRONG_AGE();
    }
};
```

730 Mastering C++

```
        s_age = m;
    }
    virtual int GetAge(void)
    {
        return s_age;
    }
};
void main()
{
    int father_age;
    int son_age;
    Father *basep;    // pointer to father objects
    cout << "Enter Age of Father: ";
    cin >> father_age;
    try
    {
        basep = new Father( father_age );    // pointer to father
    }
    catch( WRONG_AGE )
    {
        cout << "Error: Father's Age is < 0";
        exit( 1 );
    }
    cout << "Father's Age: ";
    cout << basep->GetAge() << endl; // calls father::GetAge
    delete basep; // remove Father class object
    cout << "Enter Age of Son: ";
    cin >> son_age;
    try
    {
        basep = new Son( father_age, son_age );    // pointer to son
    }
    catch( WRONG_AGE )
    {
        cout << "Error: Father age cannot be less than son age!!!";
        exit( 1 );
    }
    cout << "Son's Age: ";
    cout << basep->GetAge() << endl; // calls son::GetAge()
    delete basep; // remove Son class object
}
```

Run1

Enter Age of Father: 45
Father's Age: 45
Enter Age of Son: 20
Son's Age: 20

Run2

Enter Age of Father: 20
Father's Age: 20

```
Enter Age of Son: 45
Error: Father age cannot be less than son age!!!
```

Run3

```
Enter Age of Father: -2
Error: Father's Age is < 0
```

The first try-block in the `main()` will check for the validity of the father's age. As in *Run3*, if the fathers' age is less than the zero, the exception `WRONG_AGE` is raised.

The second try-block in the `main()` will check for the validity of son's age in accordance with father's age. As in *Run2*, if son's age is greater than the age of father, the exception `WRONG_AGE` is raised.

19.12 Exceptions in Class Templates

The program `matrix.cpp` illustrates exception handling mechanism along with other features of OOPs such as class templates, operator overloading including friend functions, binary operators, assignment through object copy, etc. The specification of the template class `matrix` with exceptions is similar to that without exceptions, but, errors are handled using exceptions instead of returning an error code as a function return value.

```
// matrix.cpp: Matrix manipulation class template and exception handling
#include <iostream.h>
#include <process.h>
const int TRUE = 1;
const int FALSE = 0;
// empty class for matrix exception
class MatError
{
};
// template matrix class
template <class T>
class matrix
{
private:
    int MaxRow;    // number of rows
    int MaxCol;    // number of columns
    T MatPtr[5][5]; // if T is int, int MatrPtr[5][5];
public:
    matrix()
    {
        MaxRow = 0; MaxCol = 0;
    }
    matrix::matrix( int row, int col )
    {
        MaxRow = row;
        MaxCol = col;
    }
    friend istream & operator >> ( istream & cin, matrix <T> &dm );
    friend ostream & operator << ( ostream & cout, matrix <T> &sm );
    matrix <T> operator + ( matrix <T> b );
};
```

732 Mastering C++

```
        matrix <T> operator - ( matrix <T> b );
        matrix <T> operator * ( matrix <T> b );
        void operator = ( matrix <T> b );
        int operator == ( matrix <T> b );
};
template <class T>
matrix<T> matrix<T>::operator + ( matrix <T> b )
{
    matrix <T> c( MaxRow, MaxCol );
    int i, j;
    if( MaxRow != b.MaxRow || MaxCol != b.MaxCol )
        throw MatError();
    for( i = 0; i < MaxRow; i++ )
        for( j = 0; j < MaxCol; j++ )
            c.MatPtr[i][j] = MatPtr[i][j] + b.MatPtr[i][j];
    return( c );
}
template <class T>
matrix <T> matrix<T>::operator - ( matrix <T> b )
{
    matrix <T> c( MaxRow, MaxCol );
    int i, j;
    if( MaxRow != b.MaxRow || MaxCol != b.MaxCol )
        throw MatError();
    for( i = 0; i < MaxRow; i++ )
        for( j = 0; j < MaxCol; j++ )
            c.MatPtr[i][j] = MatPtr[i][j] - b.MatPtr[i][j];
    return( c );
}
template <class T>
matrix <T> matrix<T>::operator * ( matrix <T> b )
{
    matrix <T> c( MaxRow, b.MaxCol );
    int i, j, k;
    if( MaxCol != b.MaxRow )
        throw MatError();
    for( i = 0; i < c.MaxRow; i++ )
        for( j = 0; j < c.MaxCol; j++ )
        {
            c.MatPtr[i][j] = 0;
            for( k = 0; k < MaxCol; k++ )
                c.MatPtr[i][j] += MatPtr[i][k] * b.MatPtr[k][j];
        }
    return( c );
}
template <class T>
int matrix<T>::operator == ( matrix <T> b )
{
    int i, j;
    if( MaxRow != b.MaxRow || MaxCol != b.MaxCol )
        return( FALSE );
}
```



```

for( i = 0; i < MaxRow; i++ )
{
    for( j = 0; j < MaxCol; j++ )
        if( MatPtr[i][j] != b.MatPtr[i][j] )
            return( FALSE );
    }
return( TRUE );
}
// function invoked when statement of type matrix a = matrix b is used
template <class T>
void matrix<T>::operator = ( matrix <T> b )
{
    int i, j;
    MaxRow = b.MaxRow;
    MaxCol = b.MaxCol;
    for( i = 0; i < MaxRow; i++ )
        for( j = 0; j < MaxCol; j++ )
            MatPtr[i][j] = b.MatPtr[i][j];
}
template <class T>
istream & operator >> ( istream & cin, matrix <T> &dm )
{
    int i, j;
    cout << "How many rows ? ";
    cin >> dm.MaxRow;
    cout << "How many columns ? ";
    cin >> dm.MaxCol;
    for( i = 0; i < dm.MaxRow; i++ )
        for( j = 0; j < dm.MaxCol; j++ )
            {
                cout << "Matrix[" << i << ", " << j << "] = ? ";
                cin >> dm.MatPtr[i][j];
            }
    return( cin );
}
template <class T>
ostream & operator << ( ostream & cout, matrix <T> &sm )
{
    int i, j;
    for( i = 0; i < sm.MaxRow; i++ )
        {
            cout << endl;
            for( j = 0; j < sm.MaxCol; j++ )
                cout << sm.MatPtr[i][j] << " ";
        }
    return( cout );
}
void main()
{
    matrix <int> a;    // to store float elements
    matrix <int> b;    // matrix <float> a; matrix <float> b;
}

```

```

cout << "Enter Matrix A details..." << endl;
cin >> a;
cout << "Enter Matrix B details..." << endl;
cin >> b;
cout << "Matrix A is ...";
cout << a << endl;
cout << "Matrix B is ...";
cout << b;
matrix <int> c;
try
{
    c = a + b;
    cout << endl << "C = A + B...";
    cout << c;
}
catch( MatError )
{
    cout << endl << "Error: Invalid matrix order for addition";
}
matrix <int> d;
try
{
    d = a - b;
    cout << endl << "D = A - B...";
    cout << d;
}
catch( MatError )
{
    cout << endl << "Error: Invalid matrix order for subtraction";
}
matrix <int> e( 3, 3 );
try
{
    e = a * b;
    cout << endl << "E = A * B...";
    cout << e;
}
catch( MatError )
{
    cout << endl << "Error: Invalid matrix order for multiplication";
}
cout << endl << "(Is matrix A equal to matrix B) ? ";
if( a == b )
    cout << "Yes";
else
    cout << "No";
}

```

Run

```

Enter Matrix A details...
How many rows ? 1

```

```

How many columns ? 2
Matrix[0,0] = ? 1
Matrix[0,1] = ? 2
Enter Matrix B details...
How many rows ? 2
How many columns ? 1
Matrix[0,0] = ? 1
Matrix[1,0] = ? 2
Matrix A is ...
1 2
Matrix B is ...
1
2
Error: Invalid matrix order for addition
Error: Invalid matrix order for subtraction
E = A * B...
5
(Is matrix A equal to matrix B) ? No

```

In the definition of `matrix` class's member functions, it can be observed that the validity of a matrix operation is handled by exceptions. For instance, in the overloaded member function operator `+`, the statement

```

if( MaxRow != b.MaxRow || MaxCol != b.MaxCol )
{
    cout << "Error: Invalid matrix order for addition";
    throw MatError;
}

```

raises an exception `MatError` if there is a mismatch in the row and column count of the two matrices involved in the addition operation. Note that, function templates can also raise exceptions.

19.13 Fault Tolerant Design Techniques

Fault tolerant software design techniques can be classified into the following:

- (1) N-version programming
- (2) Recovery block

These schemes correspond to the hardware fault tolerance methods, *static redundancy* (fault masking or voting) and *dynamic redundancy* respectively.

N-Version Programming

In this technique N-programmers develop N algorithms for the same problem without interacting with each other. All these algorithms are executed simultaneously on a multiprocessor system and the majority solution is taken as the correct answer.

Recovery Block

The recovery block structure represents the dynamic redundancy approach to software fault tolerance. It consists of three software elements: (1) *primary routine*, which executes critical software functions; (2) *acceptance test*, which tests the output of the primary routine after each execution; and (3) *alternate routine*, which performs the same function as the primary routine (but may be less capable or slower), and is invoked by an acceptance test after detection of a fault.

In fault tolerance, once the error has been detected, the next goal is error recovery. The erroneous state must be replaced by an acceptable valid state from which processing may proceed. Forward error recovery attempts to identify any damage to the system state and to repair it in some way, so that failure may be avoided. It simply restores previously saved values of the system state and proceeds from there, possibly using a different program than the one that led to the error. Backward error recovery can be used with unanticipated faults and unlike forward error recovery, it can be used to recover from design faults. Figure 19.8, demonstrates the model of a recovery block and its requirements.

The simplest structure of the recovery block is:

```

Ensure T
  By P
Else
  By Q
Else
  Error
    
```

where T is the *acceptance-test* condition that is expected to be met by successful execution of either primary routine P or the alternate routine Q. The structure is easily expanded to accommodate several alternatives Q1, Q2, ..., Qn.

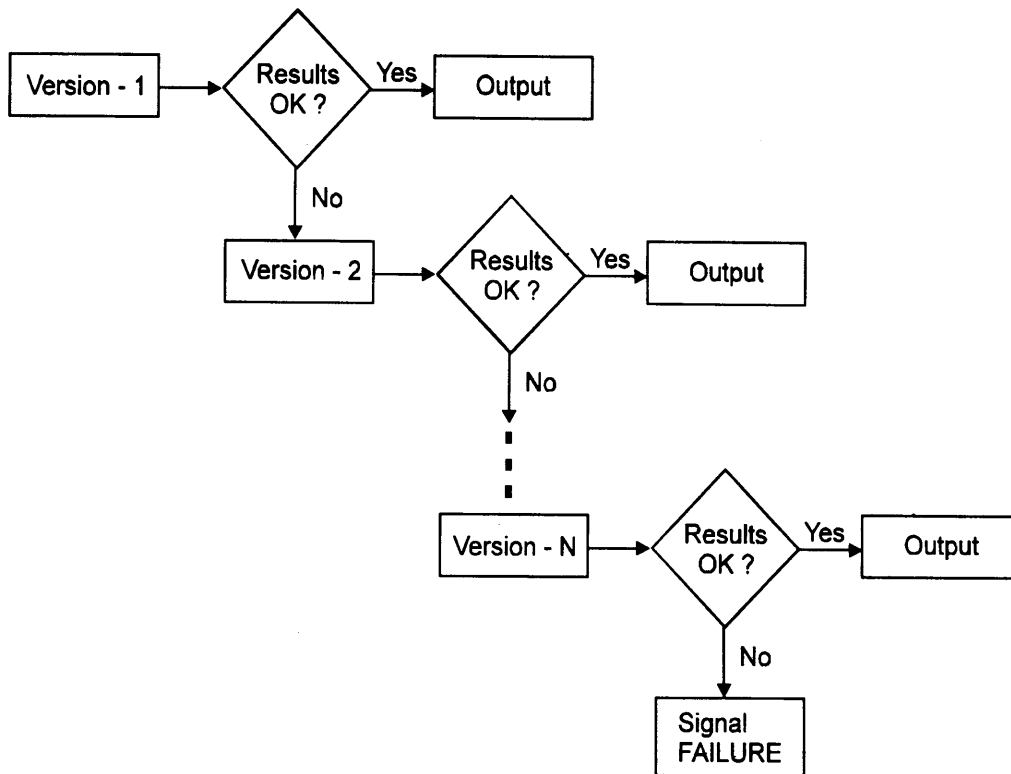


Figure 19.8: Recovery block programming model

19.14 Case-Study on Software Fault Tolerance

A simple example is chosen for the study of fault tolerance programming and the same is used for implementation in C++. C++ does not provide any explicit constructs for fault tolerance, however, the constructs `throw`, `try`, and `catch` can be suitably used to simulate the action of fault tolerance. These exception handling constructs are suitable for implementing the recovery block technique.

Consider a procedure (P) for computing:

```
sum = i'+j'+k';
```

The body of P is the sequential composition of the operation,

```
(c1) i = i+j;
(c2) i = i+k;
```

The behavior of the above procedure P can be examined by considering various versions of the procedure P (`proc p`) for different values of the variables `i`, `j`, and `k`.

Version 1:

```
proc P signal OW
begin
  i = i + j [ OV -> signal OW ];
  i = i + k [ OV -> i = i - j; signal OW ];
end
```

The semantic definition of the assignment operator `=` specifies that whenever the evaluation of the right hand side expression *terminates exceptionally* (overflow occurs, `OV`), no new value is assigned to the left hand side variable. Then, P will terminate exceptionally by executing the recovery block (if it exists) and signals an `OW` (overflow word) exception label in the final state.

Data Case 1: `i <- MaxValue`, `j <- MaxValue`, and `k <- (-MaxValue)`

Operation `i+j+k` (as per data case 1) is valid, but `i+j` exceeds the representation limit leading to an exception.

Version 2:

```
proc P signal OW
begin
  i = i + k [ OV -> signal OW ];
  i = i + j [ OV -> i = i - k; signal OW ];
end
```

This version terminates with a valid final state for the data case 1.

Data Case 2: `i <- MaxValue`, `j <- (-MaxValue)`, and `k <- (MaxValue)`

Operation `i+j+k` (as per data case 2) is valid, but `(i+k)` exceeds the representation limit leading to an exception.

Version 3:

```
proc P signal OW
begin
  j = j + k [ OV -> signal OW ];
  i = i + j [ OV -> j = j - k; signal OW ];
end
```

This version terminates with a valid final state for the data case 1 and case 2.

Data Case 3: $i \leftarrow (-\text{MaxValue})$, $j \leftarrow \text{MaxValue}$, and $k \leftarrow (\text{MaxValue})$

Operation $i+j+k$ (as per data case 3) is valid, but $j+k$ exceeds the representation limit leading to an exception.

Recovery Block for Procedure P: $i \leftarrow i+j+k$:

```

    Ensure no exception
    By Version - 1
    Else
        By Version - 2
        Else By Version - 3
        Else FAIL

```

Recovery Block Implementation

The recovery block technique can be implemented by nesting the exception handling constructs of C++. To understand the concepts of fault tolerant programming, consider a computer system having a 4-bit processor, supporting both signed and unsigned numbers. Some of its characteristics are the following.

- ◆ Maximum signed number can be represented is $7 (2^{4-1} - 1)$.
- ◆ Maximum unsigned number can be represented is $15 (2^4 - 1)$.
- ◆ Overflow will be indicated if the result exceeds the limit of representation.

The program `recovery.cpp` handles all the three data cases and demonstrates the characteristics desired in a fault tolerance program.

```

// recovery.cpp: Recover Block of sum( i, j, k )
#include <iostream.h>
const int MAX_SIG_INT = 7;    // say, maximum signed integer number
const int MAX_UNSIG_INT = 15; // say, maximum unsigned integer number
class OVERFLOW {}; // Overflow abstract class
int sum( int i, int j, int k )
{
    int result;
    try
    {
        // Version1 procedure
        result = i+j;
        if( result > MAX_SIG_INT )
            throw OVERFLOW();
        result = result+k;
        if( result > MAX_SIG_INT )
            throw OVERFLOW();
        cout << "Version-1 succeeds" << endl;
    }
    catch( OVERFLOW )
    {
        cout << "Version-1 fails" << endl;
        try
        {
            // Version2 procedure
            result = i+k;

```

```

        if( result > MAX_SIG_INT )
            throw OVERFLOW();
        result = result+j;
        if( result > MAX_SIG_INT )
            throw OVERFLOW();
        cout << "Version-2 succeeds" << endl;
    }
    catch( OVERFLOW )
    {
        cout << "Version-2 fails" << endl;
        try
        {
            // Version3 procedure
            result = j+k;
            if( result > MAX_SIG_INT )
                throw OVERFLOW();
            result = result+i;
            if( result > MAX_SIG_INT )
                throw OVERFLOW();
            cout << "Version-3 succeeds" << endl;
        }
        catch( OVERFLOW )
        {
            cout << "Error: Overflow. All versions failed" << endl;
        }
    }
}
return result;
}
void main()
{
    int result;
    cout << "Sum of 7, -3, 2 computation..." << endl;
    result = sum( 7, -3, 2 ); // version1 computes
    cout << "Sum = " << result << endl;
    cout << "Sum of 7, 2, -3 computation..." << endl;
    result = sum( 7, 2, -3 ); // version2 computes
    cout << "Sum = " << result << endl;
    // Device data such that version-3 succeeds
    cout << "Sum of 3, 3, 2 computation..." << endl;
    result = sum( 3, 3, 2 ); // all version fails
    cout << "Sum = " << result << endl;
}

```

Run

```

Sum of 7, -3, 2 computation...
Version-1 succeeds
Sum = 6
Sum of 7, 2, -3 computation...
Version-1 fails

```

```

Version-2 succeeds
Sum = 6
Sum of 3, 3, 2 computation...
Version-1 fails
Version-2 fails
Error: Overflow. All versions failed
Sum = 8

```

19.15 Memory Allocation Failure Exception

The operator `new` tries to create an object of the data type `Type` dynamically by allocating (if possible) `sizeof(Type)` bytes in free store (also called the *heap*). It calculates the size of `Type` without the need for an explicit `sizeof` operator. Further, the pointer returned is of the correct type, *pointer to Type*, without the need for explicit casting. The storage duration of the new object is from the point of creation until the operator `delete` destroys it by deallocating its memory, or until the end of the program. If successful, `new` returns a pointer to the new object. By default, an allocation failure (such as insufficient or fragmented heap memory) results in the predefined exception `xalloc` being thrown. The user program should always be prepared to catch the `xalloc` exception before trying to access the new object (unless user-defined `new`-handler function is defined). The program `new1.cpp` illustrates the simple mechanism of handling exceptions raised by the `new` operator.

```

// new1.cpp: new operator memory allocation test
#include <except.h>
#include <iostream.h>
void main(void)
{
    int * data;
    int size;
    cout << "How many bytes to be allocate: ";
    cin >> size;
    try
    {
        data = new int[ size ];
        cout << "Memory allocation success, address = " << data;
    }
    catch( xalloc ) // new fail exception
    { // Enter this block only if xalloc is thrown.
        // You could request other actions before terminating
        cout << "Could not allocate. Bye ...";
        exit(1);
    }
    delete data;
}

```

Run1

```

How many bytes to be allocate: 100
Memory allocation success, address = 0x16be

```


Run2

How many bytes to be allocate: 30000
 Could not allocate. Bye ...

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers. The program `new2.cpp` illustrates the handling of exceptions while allocating memory for matrix.

```
// new2.cpp: Allocate a two-dimensional space, initialize, and delete it.
#include <except.h>
#include <iostream.h>
void display( long **data, int m, int n );
void de_allocate( long **data, int m );
long main(void)
{
    int m, n; // m rows and n columns
    long **data;
    cout << "Enter rows and columns count: ";
    cin >> m >> n;
    try
    { // Test for exceptions
        data = new long *[m]; // Step 1: Set up the rows.
        for (int j = 0; j < m; j++)
            data[j] = new long[n]; // Step 2: Set up the columns
    }
    catch( xalloc )
    { // Enter this block only if xalloc is thrown.
        // Other actions could be requested before terminating
        cout << "Could not allocate. Bye ...";
        exit(1);
    }
    for( long i = 0; i < m; i++)
        for(long j = 0; j < n; j++)
            data[i][j] = i + j; // Arbitrary initialization
    display( data, m, n );
    de_allocate( data, m );
    return 0;
}
void display( long **data, int m, int n )
{
    for( int i = 0; i < m; i++ )
    {
        for( int j = 0; j < n; j++ )
            cout << data[i][j] << " ";
        cout << endl;
    }
}
void de_allocate( long **data, int m)
{
    for( int i = 0; i < m; i++ )
        delete[] data[i]; // Step 1. Delete the columns
    delete[] data; // Step 2: Delete the rows
}
```

Run1

```
Enter rows and columns count: 3 4
0 1 2 3
1 2 3 4
2 3 4 5
```

Run2

```
Enter rows and columns count: 100 300
Could not allocate. Bye ...
```

19.16 Ten Rules for Handling Exceptions Successfully

The amount of modification required to fully exploit the feature of exception handling in existing software is high. Experts point out ... *If you want to design your own exceptions and integrate them into preexisting classes, first understand the engineering effort—not only throwing exceptions but to handle them as well.* Many experts are concerned that exceptions will lull programmers into a false sense of security, believing that their code is handling errors, while in reality the exceptions are compounding more errors and hindering the software development. Implementing a real class such that it is *exception safe* can be challenging; sometimes it is not feasible.

In general, the use of exception handling is complicated by the interaction of C++ language features with certain C/C++ idioms, as well as the demanding robustness requirements expected of exception-safe. For instance, the combination of exception handling, templates, dynamic memory, and destructors make expressions containing multiple side-effects difficult to program robustly. For instance, consider the following simple C++ pseudocode function:

```
template <class T>
void SomeClass::add( parameters )
{
    element_array[ element_number++ ] = T( parameters );
    // ...
}
```

which uses a standard C/C++ idiom (auto incrementing) for adding a new element into an array. However, both the (unknown) constructors of T and its assignment operator might potentially throw exceptions. In both the cases, it is unclear whether `element_number` will be incremented or not. Moreover, the array element being assigned to, will also be in an uncertain state, which might even cause the destructor of the class `SomeClass` to fail!

Resources

The most vexing problems of exception handling arise from improper resource management. It leads to unrelease or double-release of resources. Here, the central concept of a *resource* is *something that provides functionality*. In many cases, a resource is equivalent to a data structure. However, a data structure is considered as a resource if it lives beyond a single operation. This constraint implies that resources have an internal state. This state is identified by all the resource's data values, which may be modified by operations on the resource. Often, a resource corresponds to one or more components in a subsystem such as a search table or a database. Smaller entities can also be considered as resources such as single elements of a search table or records in a database. Likewise, large systems such as a all-user processes in an operating system or a network of computers can be viewed as resources.

An important operation on a resource is releasing it, i.e., changing the state of a program in such a way that this resource is no longer active. In C++, this release is usually accomplished by a destructor—either in a delete expression, at the end of a block, or within another destructor. However, other operations can be used to release resources such as:

- ◆ The C standard library function `fclose()` releases a resource of the type `FILE *`.
- ◆ A list node might be *shut down* by putting it into a free-list rather than returning it to heap memory by calling `delete`.
- ◆ A stack class may store its elements in an array. In this case, releasing the resource (i.e., *top element of the stack*) is often accomplished by a simple decrement of the index. Thus, the top element is no longer accessible after this operation.

It is necessary to design all the resources in an exception safe way because exceptions might be thrown at arbitrary places in a program.

Problems with Exception Handling

There are several ways to integrate exception handling into a subsystem. One way is to design it during the initial development of the subsystem. Often, however, exception handling declarations and statements are added to an existing subsystem after it has been designed with the intent of making it more robust. In both the cases, especially in the latter, the following issues might be considered and solved.

1. The design of the exception class types and the class hierarchy. It should address the issues such as, which exceptions should be distinguishable by their type, which should be distinguished by data member values, which standard exceptions are to be reused, or which special purpose exception classes are to be defined ?.
2. How to throw an exception i.e., the C++ syntax for raising an exception.
3. How to pass exceptions *upwards* i.e., what must be done to correctly manage the resources that are affected as the stack unwinds.
4. How to handle an exception, i.e., remedying the problem that was the original reason for throwing an exception.
5. Syntactic and readability issues. For instance, indentation, grouping of handlers etc.
6. Use of exception handling in large systems. For example, how to handle more than one exception at the same time, how to indicate more than one problem with more than one resource etc.
7. Testability of programs with exception handling. For example, how should the “all branches”-coverage criterion for sufficient testing be redefined in the presence of exception handling ?
8. Maintenance of exception handling declarations and statements in the life cycle of software systems. For example, how does the presence of exception handling influence the understandability of code? How might the extension of class hierarchy interact with exception handling (—for example, if virtual functions in derived classes need to throw exception different from those in base class ?).

The concept of *simply throw an exception if you do not know what to do* will reduce program robustness and frustrate programmers who have to deal with all these exceptions. Therefore, the ten rules discussed below need to be followed in order to manage the exceptions well:

Rule 1: Do not throw an exception unless absolutely necessary.

A basic principle of software engineering: *Allow composition of resources* i.e., complex resources are composed from simpler ones. C++ has many construction methods to facilitate resource composition. Improper handling of an exception in such systems can lead to bad (inconsistent) states. A bad re-

source cannot be repaired — sometimes it may not even be possible to destroy it. Consider the following definition of the member function `push()` in the `Stack` class:

```
template <class T>
void Stack<T>::push( T e )
{
    .....
    vec[top++] = e; // vector insertion can cause exception
    .....
}
```

An exception in the assignment will leave the top index incremented, yet the assignment to the new top element will not occur. Any access to the top element will find an *unassigned value*. Such exceptions must be carefully designed so that consistency of resource is maintained. Throwing exceptions cause some resources to be in bad state that could be cleaned up by some handler.

Rule 2: It is not advisable to simply throw some exceptions deep in the call stack and then let C++ unwind the stack until a handler is found; this might leave behind damaged resources that cannot even be destroyed afterwards.

Two appealing solutions for handling bad resources are:

- a) Reorder the statements in each update method so that no bad composite states are encountered, even between two sub-resources.
- b) Modify each update so that if a resource enters a bad state it is restored to the original state it had before the update occurred.

The `push()` member function of the `Stack` class can be reordered as follows:

```
template <class T>
void Stack<T>::push( T e )
{
    .....
    vec[top] = e; // vector insertion can cause exception
    ++top;
    .....
}
```

In the above case, the stack index `top` will not lead to a bad state when exception occurs at assignment of `e` to `vec`.

Restoring the state back to its original value before the operation is started is complex with non-trivial C++ programs. Classes with virtual functions and templates are commonly used to write code that calls functions which are unknown at the time when the calling code is written. Therefore, it is much more harder to integrate exception handling into C++, compared to C. However, it is possible to handle exceptions without too much effort.

Rule 3: All the resources should be designed in such a way that every technically possible state is a shut-down state.

The following design principle can be concluded when resources are designed according to Rule 3: The only thing an exception handler can do with a damaged resource is to shut it down (release or free).

Rule 4: The responsibility for managing a resource lies either with a class (i.e., the destructor of the class releases the resource); or with the block that acquired the resources (i.e., the resource is released on exit from the block).

Consider a simple example of `Stack` data structure. It has a `push()` function that sometimes has to allocate a new array. It does this in the following way:

```
if( buffer is too small )
{
    T *new_buffer = new T[ nelems ];    // (a)
    ...fill new_buffer...;
    delete [] vec;                      // (b)
    vec = new_buffer;
}
```

At step (a) in the above segment, the resource `new_buffer` is created under the responsibility of the block. If anything goes wrong after this point, it would be the responsibility of the block to delete the buffer again (which it does not do in the code). At step (b), the responsibility is transferred to the stack object by assigning it to the member `vec` of the class `Stack`. The responsibility to release resources now lies with the object's destructor. Thus, if a function is exited due to an exception, the destructor has to release the buffer.

Rule 5: Symmetric resource management; resource management of a purely block-local resource: The responsibility of a block-local resource always lies with the acquiring block.

Of course, with this method, it is not possible to put a resource under the object responsibility, which is necessary for all asymmetric resource management problems. Two general schemes (or patterns) for solving this type of problem are 1) setting resource of an object and 2) replacing an object resource. As a building block for these patterns need *secure operations* that allow to transfer resource responsibilities without throwing exceptions i.e., all destructors in a C++ program should have an empty specification `throw()`. The first problem arises most often in constructors and assignment operators where a new dynamic resource is needed to store part of the object's value. Resource management for such a resource is done as indicated in the **Rule 6**. The second problem arises in the implementation of containers that automatically adjust their size, for example, the `Stack` class. Again, clear responsibility management is the key to the correct design as indicated in the **Rule 7**.

Rule 6: Resource management for a new object resource. To handle this, use the following pattern:

- a) A load resource of suitable size is acquired
- b) The resource is used (usually initialized) as necessary
- c) The resource is put under an object's responsibility

The responsibility of the resources lies with the acquiring block in the above step a) and b) and with some object after c). The responsibility transfer at c) must happen in such a way that the responsibility is always with exactly one agent—either the object or the block.

Rule 7: Resource management for replacing an object resource. To handle this situation, use the following pattern:

- a) A local resource of suitable size is acquired under block responsibility
- b) The resource is used (usually initialized) as necessary
- c) The responsibility for the object resource and local resource are exchanged
- d) The new local resource (the former object resource) is released

The following is an example of such a sequence:

```
template <class T>
void Stack::pop() ( T & e )    // throw( bad_alloc, ..T(),...)
{
```

```

    if( top == nelems )
    {
        nelems *= 2;
        AutoPtrArray <T> new_buffer = nelems;    // (a)
        for( int i = 0; i < n; ++i )            // (b)
            new_buffer[i] = vec[i];
        new_buffer.swap_with( vec );            // (c)
        /* destructor of new_buffer */          // (d)
    }
    vec[top++] = e;
}

```

Rule 8: When designing a throw-and-keep resource, all operations with side effects on subresources occurring in some resource constraint must be viewed as resource acquisitions.

Rule 9: Each modification of a subresource of a throw-and-keep resource that might throw an exception must be wrapped as shown in the following code:

```

try
{
    //... modification...;
}
catch(...)
{
    // make subresource invisible to all operations
    // except those that destroy it
    throw;
}

```

Moreover, all the actions in the catch-block must be secure operations.

Rule 10: Resource management for a new object resource with return statement. To handle this situation, use the following pattern:

- a) A local resource is acquired.
- b) The responsibility of the local and the object resources are swapped.
- c) The resource is used as necessary (including the return statement).
If an exception is thrown in (c), perform d) and e):
- d) The responsibility of the local and the object resources are swapped back.
- e) The exception is re-thrown (in order to avoid losing information about error occurrence and reason for its occurrence).

The following is an example of such a sequence:

```

template <class T>
T KeepableStack::pop() ( T & e )    // throw( XPopOnEmptyStack, ...T( T& ) )
{
    if( top == 0 )
        throw XPopOnEmptyStack( "Stack<T>::pop" );
    Auto_uinit new_top(top-1);      // (a)
    new_top.swap_with(top);         // (b)
    try
    {
        return vec[top];            // (c)
    }
}

```

```

catch( ... )
{
    new_top.swap_with( top );    // (d)
    throw;                      // (e)
}
}

```

Based on the background of the above ten rules in managing exception handling, it is possible to design new patterns. A new pattern for responsibility management includes transferring responsibilities from an acquiring block to a surrounding block; or from one object to another, and so on.

Review Questions

- 19.1 What are exceptions ? What are the differences between synchronous and asynchronous exceptions ?
- 19.2 Explain the techniques of building reliable software.
- 19.3 Explain the exception handling model of C++ with various constructs supported by it.
- 19.4 Write an interactive program to compute square root of a number. The input value must be tested for validity. If it is negative, the user defined function `my_sqrt()` should raise an exception.
- 19.5 What is the syntax for indicating a list of exceptions that a function can raise. What happens if an unspecified exception is raised ?
- 19.6 Write a program to demonstrate the catching of all exceptions.
- 19.7 What happens when an exception is raised in a try-block having a few constructed objects ? What is stack unwinding ?
- 19.8 What happens when a raised exception is not caught by catch-block ?
- 19.9 How does C++'s throwing and catching exceptions differ from C's `setjmp()` and `longjmp()` ?
- 19.10 Write a program which transfers the control to user defined terminate function when raised exception is uncaught.
- 19.11 When does the function `unexpected()` is invoked ? Write a program which installs the user defined `unexpected` function to handle exceptions.
- 19.12 Write an interactive program which divides two complex numbers. Overload divide (/) operator. Handle cases such as division-by-zero using exceptions.
- 19.13 Consider that the base class `Stack` is available. It does not take care of situations such as overflow or underflow. Enhance this class to `MyStack` which raises an exception whenever overflow or underflow error occurs.
- 19.14 What are the different fault tolerant design techniques available ? Explain recovery block programming technique with a suitable example.
- 19.15 When memory allocation fails, how does the `new` operator notify the error to the caller ?
- 19.16 Write a program to add two vectors. Each vector object, instance of the class `Vector`, is having dynamic allocation of their data members. Catch exception raised by `new` operator and take corrective actions.
- 19.17 Explain why addition of exceptions to most software is likely to diminish the overall reliability and impede the software development process if extraordinary care is not taken ?
- 19.18 List the ten rules for handling exceptions successfully
- 19.19 What are the issues that need to be considered while designing fault tolerant software ?
- 19.20 Write a program for matrix multiplication. The matrix multiplication function should notify if the order of matrix is invalid using exceptions.

OO Analysis, Design and Development

OOP systems are sold on the promise of improved productivity through object reuse and high level of code modularity. These aspects precisely lead to their greatest benefit, namely, improved software quality, considering “the objective of OO design is to mirror real world objects” in the software systems. OO Technology encompasses not only OOPs but also other OO concepts such as user interface, analysis, design, and data base management systems. Lastly, using OOPs facilitates an iterative style of development rather than the traditional *waterfall* approaches. The object-oriented approach centers around modeling the real world in terms of objects, in contrast to the traditional approaches which emphasize function oriented view and separates data-and-functions.

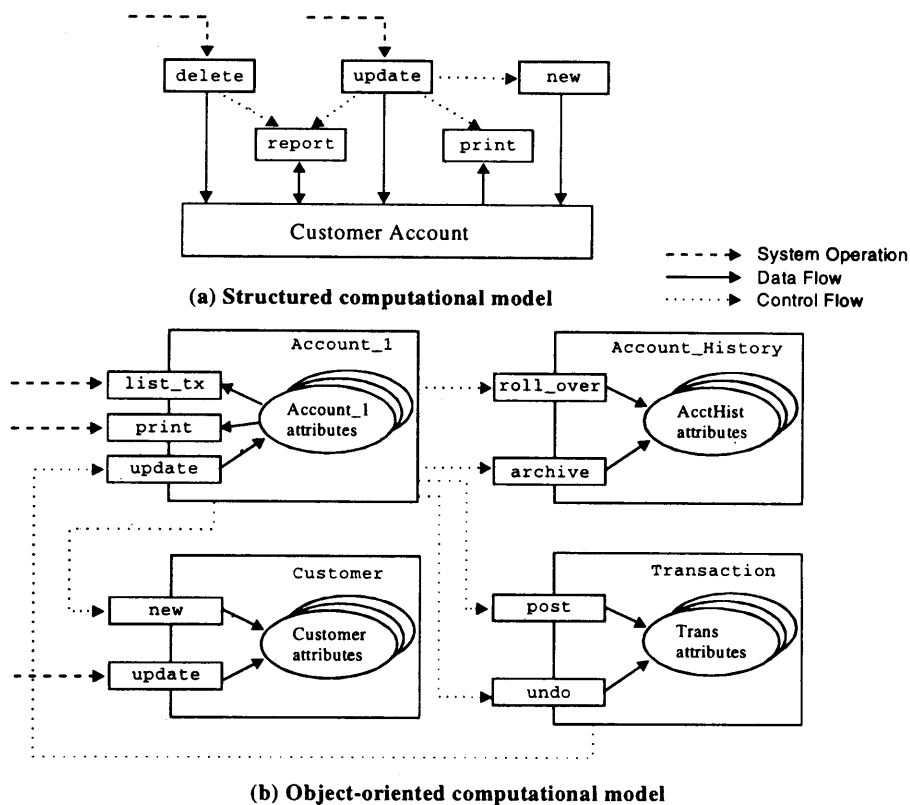


Figure 20.1: Structured Vs. Object-oriented computational model

Software engineering deals with the various tools, methods, and procedures required for controlling the complexity of software development, project management, and its maintenance. Object-oriented

development emphasizes on using programming languages with certain unique capabilities for real-world object modeling. Object model is the conceptual framework for object-oriented development. The four major elements of this model are encapsulation, abstraction, modularity, and hierarchy. The computational model of the structured and object-oriented model is shown in Figure 20.1. OO development tends to be iterative and incremental growth, compared to conventional development.

A systems development methodology combines tools and techniques to guide the process of developing large scale information systems. Dramatic improvement in hardware performance and the adoption of high-level languages has enabled to build large and more complicated systems. The conventional methodologies decompose the process of system development life cycle into discrete project phases with *frozen* deliverables or formal documents, which serve as the input to the next phase.

20.1 Software Life Cycle: Water-Fall Model

Software systems pass through two principal phases during their life cycle:

- ◆ The development phase
- ◆ The operations and maintenance phase

The development phase begins when the need for the product is identified; it ends when the implemented product is tested and delivered for operation. Operation and maintenance include all activities during the operation of the software, such as fixing bugs discovered during operation, making performance enhancements, adapting the system to its environment, adding minor features, etc. During this phase, the system may also evolve when major-functions are added. To illustrate the software life cycle, the *waterfall model* or *conventional life cycle model* (see Figure 20.2) has proven convenient.

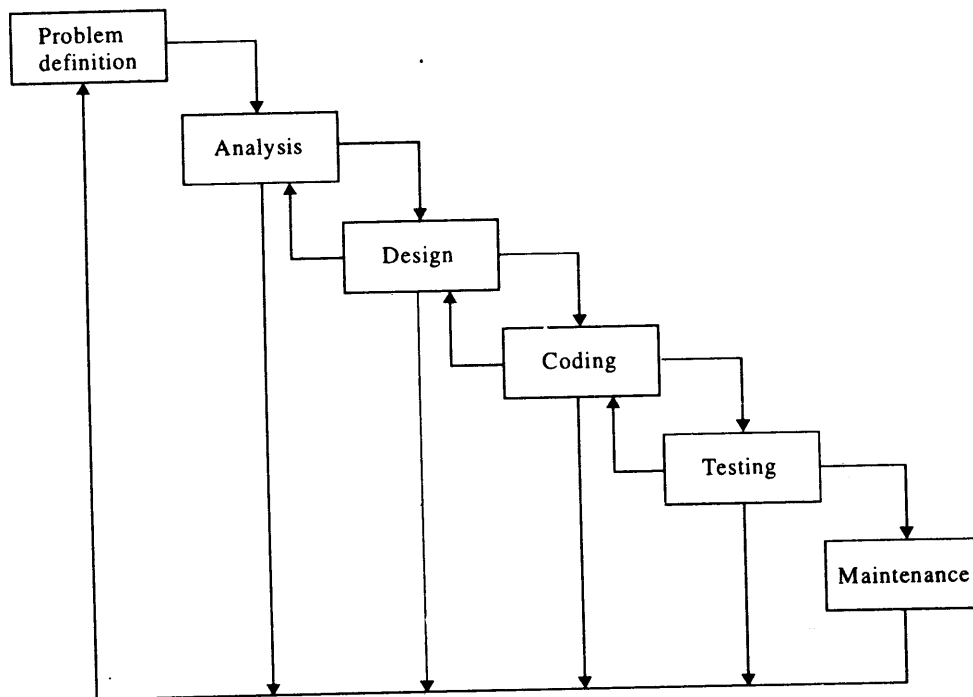


Figure 20.2: Water fall model for software development

Conventional life cycle of software development passes through various phases. They include definition of system requirements, generation of software requirements, software design, coding, and final testing and reliability modeling.

Problem Definition: The first stage in the development process is understanding the problem in question and its requirements. Requirements may be specified by the end-user, or, if the software system is *embedded* within a larger system, they may be derived from the system requirements. Requirements, therefore, include the context in which the problem arose, functionality expected from the system, and system constraints. At this point, the managers and software specialists decide whether it is feasible to build the system.

Analysis: A system analyst observes the feasibility of system development. If system development is cost effective based on the management approval, then design, coding, etc., phases will be executed, otherwise, it will be aborted; no progress of other phases will be made. Analysis phase delivers requirements specification. If project is approved, software specialists try to understand the requirements and define the specifications to meet those requirements. The system specification serves as an interface between the designer and implementor as well as between the implementor and user. This describes external behavior of the software without bothering about the internal implementation. Specification must be carefully checked for suitability, omission, inconsistencies, and ambiguities.

Design: Design is the process of mapping system requirements defined during analysis to an abstract representation of a specific-system implementation, meeting the cost and performance constraints. The detailed design involves the analysis of various alternatives, including tradeoff among the number of possible solutions based on the existing constraints.

It describes how the system is to be implemented so that, it meets the specification. Since the whole system may be very complex, the main design objective is decomposition. The system is divided into modules and their interactions. The modules may then be further decomposed into submodules and procedures until each module can be implemented easily.

Coding/Implementation: Once the specification and the design of the software is over, the choice of a programming language remains as one of the most critical aspect in producing reliable software. Implementation involves the actual production of code. Although it is one of the important phases, it takes only 20% of the total development time. The reliability of the code produced depends on the coding standards, implementation strategies and the facilities provided by the host language for reliable programming.

Testing: The truth hurts: *Many software development organizations pay lip service to quality—shipping untested software when deadline pressures dictate,* a not-so-surprising conclusion drawn from many surveys.

Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies the specified requirements. Normally, most of the testing and debugging is done after the system has been implemented (integrated testing). A large percentage of errors are discovered during testing originates in the requirement and design phases. Requirement and design errors are more expensive to correct (typically, about 100 times more expensive than implementation errors). Clearly, more efforts are needed to be spent in requirement definition and design, which must be considered as separate stages in software development. People must become more aware of the importance of earlier phases in the software life cycle.