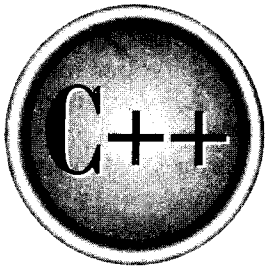
The Complete Reference




# Chapter 19

## Exception Handling

This chapter discusses the exception handling subsystem. *Exception handling* allows you to manage run-time errors in an orderly fashion. Using exception handling, your program can automatically invoke an error-handling routine when an error occurs. The principal advantage of exception handling is that it automates much of the error-handling code that previously had to be coded "by hand" in any large program.

## Exception Handling Fundamentals

C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The following discussion elaborates upon this general description.

Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.) Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown. The general form of **try** and **catch** are shown here.

```
try {
  // try block
}
catch (type1 arg) {
  // catch block
}
catch (type2 arg) {
  // catch block
}
catch (type3 arg) {
  // catch block
}
.
.
.
catch (typeN arg) {
  // catch block
}
```

The **try** can be as short as a few statements within one function or as all-encompassing as enclosing the **main( )** function code within a **try** block (which effectively causes the entire program to be monitored).

When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement associated with a **try**. Which **catch** statement is used is determined by the type of the exception. That is, if the data type specified by a **catch** matches that of the exception, then that **catch** statement is executed (and all others are bypassed). When an exception is caught, *arg* will receive its value. Any type of data may be caught, including classes that you create. If no exception is thrown (that is, no error occurs within the **try** block), then no **catch** statement is executed.

The general form of the **throw** statement is shown here:

throw *exception*;

**throw** generates the exception specified by *exception*. If this exception is to be caught, then **throw** must be executed either from within a **try** block itself, or from any function called from within the **try** block (directly or indirectly).

If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination may occur. Throwing an unhandled exception causes the standard library function **terminate( )** to be invoked. By default, **terminate( )** calls **abort( )** to stop your program, but you can specify your own termination handler, as described later in this chapter.

Here is a simple example that shows the way C++ exception handling operates.

```
// A simple exception handling example.
#include <iostream>
using namespace std;

int main()
{
  cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    throw 100; // throw an error
    cout << "This will not execute";
  }
  catch (int i) { // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }

  cout << "End";
```

```
  return 0;
}
```

This program displays the following output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Look carefully at this program. As you can see, there is a **try** block containing three statements and a **catch(int i)** statement that processes an integer exception. Within the **try** block, only two of the three statements will execute: the first **cout** statement and the **throw**. Once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is, **catch** is *not* called. Rather, program execution is transferred to it. (The program's stack is automatically reset as needed to accomplish this.) Thus, the **cout** statement following the **throw** will never execute.

Usually, the code within a **catch** statement attempts to remedy an error by taking appropriate action. If the error can be fixed, execution will continue with the statements following the **catch**. However, often an error cannot be fixed and a **catch** block will terminate the program with a call to **exit( )** or **abort( )**.

As mentioned, the type of the exception must match the type specified in a **catch** statement. For example, in the preceding example, if you change the type in the **catch** statement to **double**, the exception will not be caught and abnormal termination will occur. This change is shown here.

```
// This example will not work.
#include <iostream>
using namespace std;

int main()
{
  cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    throw 100; // throw an error
    cout << "This will not execute";
  }
  catch (double i) { // won't work for an int exception
    cout << "Caught an exception -- value is: ";
```

```
    cout << i << "\n";
  }

  cout << "End";

  return 0;
}
```

This program produces the following output because the integer exception will not be caught by the **catch(double i)** statement. (Of course, the precise message describing abnormal termination will vary from compiler to compiler.)

```
Start
Inside try block
Abnormal program termination
```

An exception can be thrown from outside the **try** block as long as it is thrown by a function that is called from within **try** block. For example, this is a valid program.

```
/* Throwing an exception from a function outside the
   try block.
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "Inside Xtest, test is: " << test << "\n";
  if(test) throw test;
}

int main()
{
  cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    Xtest(0);
    Xtest(1);
    Xtest(2);
  }
  catch (int i) { // catch an error
```

```
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }

  cout << "End";

  return 0;
}
```

This program produces the following output:

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

A **try** block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. For example, examine this program.

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

// Localize a try/catch to a function.
void Xhandler(int test)
{
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << "Caught Exception #: " << i << '\n';
  }
}

int main()
{
  cout << "Start\n";

  Xhandler(1);
```

```
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "End";

  return 0;
}
```

This program displays this output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset.

It is important to understand that the code associated with a **catch** statement will be executed only if it catches an exception. Otherwise, execution simply bypasses the **catch** altogether. (That is, execution never flows into a **catch** statement.) For example, in the following program, no exception is thrown, so the **catch** statement does not execute.

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

int main()
{
  cout << "Start\n";

  try { // start a try block
    cout << "Inside try block\n";
    cout << "Still inside try block\n";
  }
  catch (int i) { // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
```

```
  cout << "End";

  return 0;
}
```

The preceding program produces the following output.

```
Start
Inside try block
Still inside try block
End
```

As you see, the **catch** statement is bypassed by the flow of execution.

## Catching Class Types

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error. The following example demonstrates this.

```
// Catching class type exceptions.
#include <iostream>
#include <cstring>
using namespace std;

class MyException {
public:
  char str_what[80];
  int what;

  MyException() { *str_what = 0; what = 0; }

  MyException(char *s, int e) {
    strcpy(str_what, s);
    what = e;
  }
};
```

```
int main()
{
  int i;

  try {
    cout << "Enter a positive number: ";
    cin >> i;
    if(i<0)
      throw MyException("Not Positive", i);
  }
  catch (MyException e) { // catch an error
    cout << e.str_what << ": ";
    cout << e.what << "\n";
  }

  return 0;
}
```

Here is a sample run:

```
Enter a positive number: -4
Not Positive: -4
```

The program prompts the user for a positive number. If a negative number is entered, an object of the class **MyException** is created that describes the error. Thus, **MyException** encapsulates information about the error. This information is then used by the exception handler. In general, you will want to create exception classes that will encapsulate information about an error to enable the exception handler to respond effectively.

## Using Multiple catch Statements

As stated, you can have more than one **catch** associated with a **try**. In fact, it is common to do so. However, each **catch** must catch a different type of exception. For example, this program catches both integers and strings.

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

// Different types of exceptions can be caught.
```

```
void Xhandler(int test)
{
  try{
    if(test) throw test;
    else throw "Value is zero";
  }
  catch(int i) {
    cout << "Caught Exception #: " << i << '\n';
  }
  catch(const char *str) {
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}

int main()
{
  cout << "Start\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "End";

  return 0;
}
```

This program produces the following output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

As you can see, each **catch** statement responds only to its own type.

In general, **catch** expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other **catch** blocks are ignored.

## Handling Derived-Class Exceptions

You need to be careful how you order your **catch** statements when trying to catch exception types that involve base and derived classes because a **catch** clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the **catch** sequence. If you don't do this, the base class **catch** will also catch all derived classes. For example, consider the following program.

```
// Catching derived classes.
#include <iostream>
using namespace std;

class B {
};

class D: public B {
};

int main()
{
  D derived;

  try {
    throw derived;
  }
  catch(B b) {
    cout << "Caught a base class.\n";
  }
  catch(D d) {
    cout << "This won't execute.\n";
  }

  return 0;
}
```

Here, because **derived** is an object that has **B** as a base class, it will be caught by the first **catch** clause and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error. Either way, to fix this condition, reverse the order of the **catch** clauses.

# Exception Handling Options

There are several additional features and nuances to C++ exception handling that make it easier and more convenient to use. These attributes are discussed here.

## Catching All Exceptions

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**.

```
catch(...) {
  // process all exceptions
}
```

Here, the ellipsis matches any type of data. The following program illustrates **catch(...)**.

```
// This example catches all exceptions.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
  }
  catch(...) { // catch all exceptions
    cout << "Caught One!\n";
  }
}

int main()
{
  cout << "Start\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "End";
```

```
  return 0;
}
```

This program displays the following output.

```
Start
Caught One!
Caught One!
Caught One!
End
```

As you can see, all three **throw**s were caught using the one **catch** statement.

One very good use for **catch(...)** is as the last **catch** of a cluster of catches. In this capacity it provides a useful default or "catch all" statement. For example, this slightly different version of the preceding program explicity catches integer exceptions but relies upon **catch(...)** to catch all others.

```
// This example uses catch(...) as a default.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
  }
  catch(int i) { // catch an int exception
    cout << "Caught an integer\n";
  }
  catch(...) { // catch all other exceptions
    cout << "Caught One!\n";
  }
}

int main()
{
  cout << "Start\n";
```

```
  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "End";

  return 0;
}
```

The output produced by this program is shown here.

```
Start
Caught an integer
Caught One!
Caught One!
End
```

As this example suggests, using **catch(...)** as a default is a good way to catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

## Restricting Exceptions

You can restrict the type of exceptions that a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a **throw** clause to a function definition. The general form of this is shown here:

*ret-type func-name*(*arg-list*) throw(*type-list*)
{
 // ...
}

Here, only those data types contained in the comma-separated *type-list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw *any* exceptions, then use an empty list.

Attempting to throw an exception that is not supported by a function will cause the standard library function **unexpected( )** to be called. By default, this causes **abort( )** to be called, which causes abnormal program termination. However, you can specify your own unexpected handler if you like, as described later in this chapter.

The following program shows how to restrict the types of exceptions that can be thrown from a function.

```
// Restricting function throw types.
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // throw int
  if(test==1) throw 'a'; // throw char
  if(test==2) throw 123.23; // throw double
}

int main()
{
  cout << "start\n";

  try{
    Xhandler(0); // also, try passing 1 and 2 to Xhandler()
  }
  catch(int i) {
    cout << "Caught an integer\n";
  }
  catch(char c) {
    cout << "Caught char\n";
  }
  catch(double d) {
    cout << "Caught double\n";
  }

  cout << "end";

  return 0;
}
```

In this program, the function **Xhandler( )** may only throw integer, character, and **double** exceptions. If it attempts to throw any other type of exception, an abnormal program termination will occur. (That is, **unexpected( )** will be called.) To see an example of this, remove **int** from the list and retry the program.

It is important to understand that a function can be restricted only in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block *within* a

function may throw any type of exception so long as it is caught *within* that function. The restriction applies only when throwing an exception outside of the function.

The following change to **Xhandler( )** prevents it from throwing any exceptions.

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
  /* The following statements no longer work.  Instead,
     they will cause an abnormal program termination. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

## Rethrowing an Exception

If you wish to rethrow an expression from within an exception handler, you may do so by calling **throw**, by itself, with no exception. This causes the current exception to be passed on to an outer **try/catch** sequence. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a **catch** block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate outward to the next **catch** statement. The following program illustrates rethrowing an exception, in this case a **char** * exception.

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "hello"; // throw a char *
  }
  catch(const char *) { // catch a char *
    cout << "Caught char * inside Xhandler\n";
    throw ; // rethrow char * out of function
  }
}
```

```
int main()
{
  cout << "Start\n";

  try{
    Xhandler();
  }
  catch(const char *) {
    cout << "Caught char * inside main\n";
  }

  cout << "End";

  return 0;
}
```

This program displays this output:

```
Start
Caught char * inside Xhandler
Caught char * inside main
End
```

## Understanding terminate( ) and unexpected( )

As mentioned earlier, **terminate( )** and **unexpected( )** are called when something goes wrong during the exception handling process. These functions are supplied by the Standard C++ library. Their prototypes are shown here:

```
void terminate( );
void unexpected( );
```

These functions require the header **<exception>**.

The **terminate( )** function is called whenever the exception handling subsystem fails to find a matching **catch** statement for an exception. It is also called if your program attempts to rethrow an exception when no exception was originally thrown. The **terminate( )** function is also called under various other, more obscure circumstances. For example, such a circumstance could occur when, in the process of unwinding the stack because of an exception, a destructor for an object being destroyed throws an exception. In general, **terminate( )** is the handler of last resort when no other handlers for an exception are available. By default, **terminate( )** calls **abort( )**.

The **unexpected( )** function is called when a function attempts to throw an exception that is not allowed by its **throw** list. By default, **unexpected( )** calls **terminate( )**.

## Setting the Terminate and Unexpected Handlers

The **terminate( )** and **unexpected( )** functions simply call other functions to actually handle an error. As just explained, by default **terminate( )** calls **abort( )**, and **unexpected( )** calls **terminate( )**. Thus, by default, both functions halt program execution when an exception handling error occurs. However, you can change the functions that are called by **terminate( )** and **unexpected( )**. Doing so allows your program to take full control of the exception handling subsystem.

To change the terminate handler, use **set_terminate( )**, shown here:

terminate_handler set_terminate(terminate_handler *newhandler*) throw( );

Here, *newhandler* is a pointer to the new terminate handler. The function returns a pointer to the old terminate handler. The new terminate handler must be of type **terminate_handler**, which is defined like this:

typedef void (*terminate_handler) ( );

The only thing that your terminate handler must do is stop program execution. It must not return to the program or resume it in any way.

To change the unexpected handler, use **set_unexpected( )**, shown here:

unexpected_handler set_unexpected(unexpected_handler *newhandler*) throw( );

Here, *newhandler* is a pointer to the new unexpected handler. The function returns a pointer to the old unexpected handler. The new unexpected handler must be of type **unexpected_handler**, which is defined like this:

typedef void (*unexpected_handler) ( );

This handler may itself throw an exception, stop the program, or call **terminate( )**. However, it must not return to the program.

Both **set_terminate( )** and **set_unexpected( )** require the header **<exception>**.

Here is an example that defines its own **terminate( )** handler.

```
// Set a new terminate handler.
#include <iostream>
```

```
#include <cstdlib>
#include <exception>
using namespace std;

void my_Thandler() {
  cout << "Inside new terminate handler\n";
  abort();
}

int main()
{
  // set a new terminate handler
  set_terminate(my_Thandler);

  try {
    cout << "Inside try block\n";
    throw 100; // throw an error
  }
  catch (double i) { // won't catch an int exception
    // ...
  }

  return 0;
}
```

The output from this program is shown here.

```
Inside try block
Inside new terminate handler
abnormal program termination
```

## The uncaught_exception( ) Function

The C++ exception handling subsystem supplies one other function that you may find useful: **uncaught_exception( )**. Its prototype is shown here:

```
bool uncaught_exception( );
```

This function returns **true** if an exception has been thrown but not yet caught. Once caught, the function returns **false**.

## The exception and bad_exception Classes

When a function supplied by the C++ standard library throws an exception, it will be an object derived from the base class **exception**. An object of the class **bad_exception** can be thrown by the unexpected handler. These classes require the header **<exception>**.

## Applying Exception Handling

Exception handling is designed to provide a structured means by which your program can handle abnormal events. This implies that the error handler must do something rational when an error occurs. For example, consider the following simple program. It inputs two numbers and divides the first by the second. It uses exception handling to manage a divide-by-zero error.

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

void divide(double a, double b);

int main()
{
  double i, j;

  do {
    cout << "Enter numerator (0 to stop): ";
    cin >> i;
    cout << "Enter denominator: ";
    cin >> j;
    divide(i, j);
  } while(i != 0);

  return 0;
}

void divide(double a, double b)
{
  try {
    if(!b) throw b; // check for divide-by-zero
    cout << "Result: " << a/b << endl;
  }
  catch (double b) {
    cout << "Can't divide by zero.\n";
```

```
    }
}
```

While the preceding program is a very simple example, it does illustrate the essential nature of exception handling. Since division by zero is illegal, the program cannot continue if a zero is entered for the second number. In this case, the exception is handled by not performing the division (which would have caused abnormal program termination) and notifying the user of the error. The program then reprompts the user for two more numbers. Thus, the error has been handled in an orderly fashion and the user may continue on with the program. The same basic concepts will apply to more complex applications of exception handling.

Exception handling is especially useful for exiting from a deeply nested set of routines when a catastrophic error occurs. In this regard, C++'s exception handling is designed to replace the rather clumsy C-based **setjmp( )** and **longjmp( )** functions.

Remember, the key point about using exception handling is to provide an orderly way of handling errors. This means rectifying the situation, if possible.