The Complete Reference C++

Chapter 38

Exception Handling and Miscellaneous Classes

This chapter describes the exception handling classes. It also describes the **auto_ptr** and **pair** classes, and gives a brief introduction to the localization library.

Exceptions

The Standard C++ library defines two headers that relate to exceptions: **<exception>** and **<stdexcept>**. Exceptions are used to report error conditions. Each header is examined here.

<exception>

The **<exception>** header defines classes, types, and functions that relate to exception handling. The classes defined by **<exception>** are shown here.

```
class exception {
public:
  exception() throw();
  exception(const bad_exception &ob) throw();
  virtual ~exception() throw();

  exception &operator=(const exception &ob) throw();
  virtual const char *what(() const throw();
};

class bad_exception: public exception {
public:
  bad_exception() throw();
  bad_exception(const bad_exception &ob) throw();
  virtual ~bad_exception() throw();

  bad_exception &operator=(const bad_exception &ob) throw();
  virtual const char *what(() const throw();
};
```

The **exception** class is a base for all exceptions defined by the C++ standard library. The **bad_exception** class is the type of exception thrown by the **unexpected( )** function. In each, the member function **what( )** returns a pointer to a null-terminated string that describes the exception.

Several important classes are derived from **exception**. The first is **bad_alloc**, thrown when the **new** operator fails. Next is **bad_typeid**. It is thrown when an illegal **typeid** expression is executed. Finally, **bad_cast** is thrown when an invalid dynamic cast is attempted. These classes contain the same members as **exception**.

The types defined by **<exception>** are:

Type	**Meaning**
terminate_handler	void (*terminate_handler) ( );
unexpected_handler	void (*unexpected_handler) ( );

The functions declared in **<exception>** are shown in Table 38-1.

<stdexcept>

The header **<stdexcept>** defines several standard exceptions that may be thrown by C++ library functions and/or its run-time system. There are two general types of exceptions defined by **<stdexcept>**: logic errors and run-time errors. Logic errors occur because of mistakes made by the programmer. Run-time errors occur because of mistakes in library functions or the run-time system, and are beyond programmer control.

Function	**Description**
terminate_handler set_terminate(terminate_handler *fn*) throw( );	Sets the function specified by *fn* as the terminate handler. A pointer to the old terminate handler is returned.
unexpected_handler set_unexpected(unexpected_handler *fn*) throw( );	Sets the function specified by *fn* as the unexpected handler. A pointer to the old unexpected handler is returned.
void terminate( );	Calls the terminate handler when a fatal exception is unhandled. Calls **abort( )** by default.
bool uncaught_exception( );	Returns true if an exception is uncaught.
void unexpected( );	Calls the unexpected exception handler when a function throws a disallowed exception. By default, **terminate( )** is called.

Table 38-1. *The Functions Defined Within* ***<exception>***

The standard exceptions defined by C++ caused by logic errors are derived from the base class **logic_error**. These exceptions are shown here.

Exception	Meaning
domain_error	Domain error occurred.
invalid_argument	Invalid argument used in function call.
length_error	An attempt was made to create an object that was too large.
out_of_range	An argument to a function was not in the required range.

The following run-time exceptions are derived from the base class **runtime_error**.

Exception	Meaning
overflow_error	Arithmetic overflow occurred.
range_error	An internal range error occurred.
underflow_error	An underflow occurred.

auto_ptr

A very interesting class is **auto_ptr**, which is declared in the header **<memory>**. An **auto_ptr** is a pointer that owns the object to which it points. Ownership of this object can be transferred to another **auto_ptr**, but some **auto_ptr** always owns the object. The key purpose of this scheme is to ensure that dynamically allocated objects are properly destroyed in all circumstances (that is, that the object's destructor is always properly executed). For example, when one **auto_ptr** object is assigned to another, only the target of the assignment will own the object. When the pointers are destroyed, the object will only be destroyed once, when the pointer holding ownership is destroyed. One benefit of this approach is that dynamically allocated objects can be destroyed when an exception is handled.

The template specification for **auto_ptr** is shown here:

```
template <class T> class auto_ptr
```

Here, **T** specifies the type of pointer stored by the **auto_ptr**.

Here are the constructors for **auto_ptr**:

explicit auto_ptr(T **ptr* = 0) throw( );

auto_ptr(auto_ptr &*ob*) throw( );

template <class T2> auto_ptr(auto_ptr<T2> &*ob*) throw( );

The first constructor creates an **auto_ptr** to the object specified by *ptr*. The second constructor creates a copy of the **auto_ptr** specified by *ob* and transfers ownership to the new object. The third converts *ob* to type T (if possible) and transfers ownership.

The **auto_ptr** class defines the =, *, and –> operators. Here are two of its member functions:

T *get( ) const throw( );

T *release( ) const throw( );

The **get( )** function returns a pointer to the stored object. The **release( )** function removes ownership of the stored object from the invoking **auto_ptr** and returns a pointer to the object. After a call to **release( )**, the pointed-to object is not automatically destroyed when the **auto_ptr** object goes out-of-scope.

Here is a short program that demonstrates the use of **auto_ptr**.

```
// Demonstrate an auto_ptr.
#include <iostream>
#include <memory>
using namespace std;

class X {
public:
  X() { cout << "constructing\n"; }
  ~X() { cout << "destructing\n"; }
  void f() { cout << "Inside f()\n"; }
};

int main()
{
  auto_ptr<X> p1(new X), p2;

  p2 = p1; // transfer ownership
  p2->f();

  // can assign to a normal pointer
  X *ptr = p2.get();
  ptr->f();

  return 0;
}
```

The output produced by this program is shown here:

```
constructing
Inside f()
Inside f()
destructing
```

Notice that **X**'s member function **f( )** can be called either through an **auto_ptr** or through the "normal" pointer returned by **get( )**.

The pair Class

The **pair** class is used to house pairs of objects, such as might be stored in an associative container. It has this template specification:

```
template <class Ktype, class Vtype> struct pair {
  typedef Ktype first_type;
  typedef Vtype second_type;
  Ktype first;
  Vtype second;

  // constructors
  pair();
  pair(const Ktype &k, const Vtype &v);
  template<class A, class B> pair(const<A, B> &ob);
}
```

The value in **first** typically contains a key, and the value in **second** typically contains the value associated with that key.

The following operators are defined for **pair**: ==, !=, <, <=, >, and >=.

You can construct a pair using either one of **pair**'s constructors or by using **make_pair( )**, which constructs a pair object based upon the types of the data used as parameters. **make_pair( )** is a generic function that has this prototype:

template <class *Ktype*, class *Vtype*>
  pair<*Ktype*, *Vtype*> make_pair(const *Ktype* &*k*, const *Vtype* &*v*);

As you can see, it returns a pair object consisting of values of the types specified by *Ktype* and *Vtype*. The advantage of **make_pair( )** is that the types of the objects being stored are determined automatically by the compiler rather than being explicitly specified by you.

The **pair** class and the **make_pair( )** function require the header **<utility>**.

Localization

Standard C++ provides an extensive localization class library. These classes allow an application to set or obtain information about the geopolitical environment in which it is executing. Thus, it defines such things as the format of currency, time and date, and collation order. It also provides for character classification. The localization library uses the header **<locale>**. It operates through a series of classes that define facets (bits of information associated with a locale). All facets are derived from the class **facet**, which is a nested class inside the **locale** class.

Frankly, the localization library is extraordinarily large and complex. A description of its features is beyond the scope of this book. While most programmers will not make direct use of the localization library, if you are involved in the preparation of internationalized programs, you will want to explore its features.

Other Classes of Interest

Here are a few other classes defined by the Standard C++ library that may be of interest.

Class	Description
type_info	Used in conjunction with the **typeid** operator and fully described in Chapter 22. Uses the header **<typeinfo>**.
numeric_limts	Encapsulates various numeric limits. Uses the header **<limits>**.
raw_storage_iterator	Encapsulates allocation of uninitialized memory. Uses the header **<memory>**.