第38章　异常处理和杂项类

本章描述异常处理类，也将描述auto_ptr和pair类，并将简要介绍本地化库。

38.1　异常

标准C++库定义了两个与异常相关的头文件：<exception>和<stdexcept>。人们使用异常来报告错误条件。下面讨论每个头文件。

38.1.1　<exception>

<exception>头定义了与异常处理有关的类、类型和函数。由<exception>定义的类如下所示。

```
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();
};
```

exception类是由C++标准库定义的所有异常的基础。bad_exception类是unexpected( )函数抛出的异常类型。在这两个类中，成员函数what( )返回一个指向描述异常的以null结束的字符串的指针。

从exception中派生出了几个重要的类。第一个是bad_alloc，它是在new运算符失败时抛出的。接着是bad_typeid，它是在不合法的typeid表达式执行时抛出的。最后，当尝试无效动态转换时抛出了bad_cast。这些类包含的成员与exception的相同。

<exception>定义的类型有：

类型	含义
terminate_handler	void (*terminate_handler) ( );
unexpected_handler	void (*unexpected_handler) ( );

在<exception>中声明的函数示于表38.1中。

表 38.1　在<exception>定义的函数

函数	说明
terminate_handler set_terminate(terminate_handler *fn*) throw( );	设置由fn指定的作为终止处理程序的函数。返回指向老的终止处理程序的指针
unexpected_handler set_unexpected(unexpected_handler *fn*) throw( );	设置由fn指定作为意外处理程序的函数。返回一个指向老的意外处理程序的指针
void terminate( );	当一个致命异常未被处理时，调用终止处理程序。默认时调用abort( )
bool uncaught_exception( );	如果异常未被捕获，返回真
void unexpected( );	当函数抛出一个未被允许的异常时，调用意外异常处理程序。默认时，调用 terminate( )

38.1.2　<stdexcept>

头文件<stdexcept>定义了几个可以被 C++ 库函数和 / 或它的运行时系统抛出的标准异常。<stdexcept>定义了两个通用类型的异常：逻辑错误和运行时错误。逻辑错误的出现是程序员出错造成的，运行时错误的出现则是由于库函数或运行时系统出错造成的，是在程序员的控制之外的。

由逻辑错误引起的C++定义的标准异常是从基类logic_error中派生的。下表列出了这些异常。

异常	含义
domain_error	出现域错误
invalid_argument	函数调用中使用的无效变元
length_error	尝试创建一个不太大的对象
out_of_range	不在所要求范围内的函数变元

下表列出的运行时异常是从基类 runtime_error 中派生的。

异常	含义
overflow_error	出现算术上溢
range_error	出现内部范围错误
underflow_error	出现下溢

38.2　auto_ptr

一个非常有趣的类是 auto_ptr，它是在头文件<memory>中声明的。auto_ptr 是一个指针，该指针拥有它指向的对象。这个对象的所有权可以转移给另一个 auto_ptr，但是某个 auto_ptr 总是拥有这个对象。这个模式的主要目的是要保证在所有情况下动态分配的对象被适当地销毁（即，总是适当地执行对象的析构函数）。例如，当把一个 auto_ptr 对象赋值给另一个时，仅仅赋值的目标将拥有这个对象。当销毁指针时，对象仅被销毁一次，即当持有所有权的指针被销毁时才如此。这种方法的一个好处是当处理异常时，动态分配的对象能被销毁。

auto_ptr 的模板规范如下所示：

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

其中，T 指定了由 auto_ptr 存储的指针类型。

下面是auto_ptr的构造函数：

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

auto_ptr(auto_ptr &ob) throw( );

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

第一个构造函数创建由ptr指定的对象的auto_ptr。第二个构造函数创建一个由ob指定的auto_ptr的副本并把所有权转换到新对象。第三个把ob转换为类型T(如果可能)并转移所有权。

auto_ptr类定义了=, * 和 ->运算符。下面是它的两个成员函数：

```
T *get( ) const throw( );
T *release( ) const throw( );
```

get( )函数返回一个指向存储对象的指针。release( )函数从调用的auto_ptr中去除存储对象的所有权并返回一个指向对象的指针。在调用release( )后，当auto_ptr对象在范围以外时，所指的对象不被自动销毁。

下面是一个演示auto_ptr用法的小程序。

```
// 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;
}
```

这个程序产生的输出如下所示：

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

注意X的成员函数可以通过一个auto_ptr或由get( )返回的正常的指针调用。

38.3 pair类

我们使用pair类来容纳对象对，即可能存储在一个关联容器中的对象对。它的模板规范如下：

```
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);
}
```

典型情况下，在first中的值包含一个键，在second中的值包含与那个键关联的值。

下面的运算符是为pair: ==, !=, <, <=, >和>= 定义的。

可以使用pair的构造函数之一或make_pair( )构造一对，make_pair( )根据用做参数的数据类型构建一对对象。make_pair( )是一个通用函数，其原型如下：

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

可以看到，它返回一对由Ktype和Vtype指定的类型值组成的对象。make_pair( )的优点是所存储的对象类型由编译器自动决定，而不是由程序员显式地指定。

pair类和make_pair( )函数都要求头文件<utility>。

38.4 本地化

标准C++提供了一个范围很广的本地化类库。这些类允许应用程序设置或获得关于它正在其中执行的地域环境的信息。因此，它定义了关于货币格式、时间和日期和校勘次序等。它也提供了字符分类情况。本地化库使用头文件<locale>。它通过一系列定义外部信息（关于本地情况的一些信息）的类来操作。所有的外部情况都是从类facet中派生的，facet是在locale类中的一个嵌套类。

坦率地讲，本地化库大而复杂，关于它的特征的描述超出了本书范围。虽然大多数程序员没有直接使用本地化库，如果涉及到国际化程序的准备工作，可能就想要探索它的特征了。

38.5 其他有趣的类

下表列出了由标准C++库定义的其他一些类。

类	说明
type_info	和typeid运算符一起使用，在第22章全面讨论过。使用头文件<typeinfo>
numeric_limts	封装各种数字限制。使用头文件<limits>
raw_storage_iterator	封装未初始化内存的分配。使用头文件<memory>