Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called *handlers*.

异常提供了一种对程序中的异常情况(如运行时错误)作出反应的方法，方法是将控制转移到称为处理程序的特殊函数。

To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a *try-block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

为了捕获异常，将代码的一部分置于异常检查之下。这是通过将那部分代码封装在try块中来实现的。当该块中出现异常情况时，将抛出一个异常，将控制权传递给异常处理程序。如果没有抛出异常，则代码继续正常运行，并忽略所有处理程序。

An exception is thrown by using the `throw` keyword from inside the `try` block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the `try` block:

在try块内部使用throw关键字会抛出异常。异常处理程序是用关键字catch声明的，该关键字必须立即放在try块之后:

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

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

The code under exception handling is enclosed in a `try` block. In this example this code simply throws an exception:

异常处理下的代码包含在try块中。在这个例子中，这段代码只是抛出了一个异常:

```
throw 20;
```

A `throw` expression accepts one parameter (in this case the integer value `20`), which is passed as an argument to the exception handler.

throw表达式接受一个参数(在本例中是整数值20)，该参数作为参数传递给异常处理程序。

The exception handler is declared with the `catch` keyword immediately after the closing brace of the `try` block. The syntax for `catch` is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the `throw` expression is checked against it, and only in the case they match, the exception is caught by that handler.

异常处理程序是在try块的右括号之后用catch关键字声明的。catch的语法类似于带有一个参数的常规函数。这个形参的类型非常重要，因为由throw表达式传递的实参的类型会根据它进行检查，只有在它们匹配的情况下，该处理程序才会捕获异常。

Multiple handlers (i.e., `catch` expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the `throw` statement is executed.

多个处理程序(例如，catch表达式)可以被链接;每一个都有不同的参数类型。只有参数类型与throw语句中指定的异常类型匹配的处理程序才会被执行。

If an ellipsis (`...`) is used as the parameter of `catch`, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

如果使用省略号(...)作为catch的形参，则该处理程序将捕获任何异常，而不管抛出的异常类型是什么。这可以用作一个默认的处理程序，用来捕获其他处理程序没有捕获的所有异常:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case, the last handler would catch any exception thrown of a type that is neither `int` nor `char`.

在这种情况下，最后一个处理程序将捕获非int或char类型的任何异常。

After an exception has been handled the program, execution resumes after the *try-catch* block, not after the `throw` statement!.

在程序处理了异常后，在try-catch块之后继续执行，而不是在throw语句之后!

It is also possible to nest `try-catch` blocks within more external `try` blocks. In these cases, we have the possibility that an internal `catch` block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:

也可以在更多的外部try块中嵌套try-catch块。在这些情况下，内部catch块有可能将异常转发到外部级别。这是通过表达式throw完成的;不带参数。例如:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Exception specification

Older code may contain *dynamic exception specifications*. They are now deprecated in C++, but still supported. A *dynamic exception specification* follows the declaration of a function, appending a `throw` specifier to it. For example:

较旧的代码可能包含动态异常说明。它们现在在c++中已被弃用，但仍然受到支持。动态异常说明紧跟在函数声明之后，并附加一个抛出说明符。例如:

```
double myfunction (char param) throw (int);
```

This declares a function called `myfunction`, which takes one argument of type `char` and returns a value of type `double`. If this function throws an exception of some type other than `int`, the function calls std::unexpected instead of looking for a handler or calling std::terminate.

这声明了一个名为myfunction的函数，该函数接受一个char类型的实参，并返回一个double类型的值。如果该函数抛出非int类型的异常，则该函数调用std::unexpected，而不是寻找处理程序或调用std::terminate。

If this `throw` specifier is left empty with no type, this means that std::unexpected is called for any exception. Functions with no `throw` specifier (regular functions) never call std::unexpected, but follow the normal path of looking for their exception handler.

如果抛出说明符为空，没有类型，这意味着任何异常都会调用std::unexpected。没有抛出说明符的函数(常规函数)永远不会调用std::unexpected，而是遵循查找异常处理程序的正常路径。

```
int myfunction (int param) throw(); // all exceptions call unexpected
int myfunction (int param);         // normal exception handling
```

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called `std::exception` and is defined in the `<exception>` header. This class has a virtual member function called `what` that returns a null-terminated character sequence (of type `char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.

c++标准库提供了一个基类，专门用于声明作为异常抛出的对象。它被称为std::exception，定义在头文件中。这个类有一个名为what的虚成员函数，它返回一个以空字符结尾的字符序列(类型为char *)，并且可以在派生类中重写该函数以包含异常的某种描述。

```
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

We have placed a handler that catches exception objects by reference (notice the ampersand `&` after the type), therefore this catches also classes derived from `exception`, like our `myex` object of type `myexception`.

我们放置了一个处理程序，它通过引用捕获异常对象(注意类型后面的&号)，因此它也捕获从异常派生的类，比如myexception类型的myex对象。

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this `exception` class. These are:

c++标准库组件抛出的所有异常都抛出派生自这个异常类的异常。这些都是:

exception	description
`bad_alloc`	thrown by `new` on allocation failure
`bad_cast`	thrown by `dynamic_cast` when it fails in a dynamic cast
`bad_exception`	thrown by certain dynamic exception specifiers
`bad_typeid`	thrown by `typeid`
`bad_function_call`	thrown by empty `function` objects
`bad_weak_ptr`	thrown by `shared_ptr` when passed a bad `weak_ptr`

Also deriving from `exception`, header `<exception>` defines two generic exception types that can be inherited by custom exceptions to report errors:

header 也派生自exception，它定义了两种通用异常类型，可被自定义异常继承以报告错误:

exception	description
`logic_error`	error related to the internal logic of the program
`runtime_error`	error detected during runtime

A typical example where standard exceptions need to be checked for is on memory allocation:

需要检查标准异常的典型例子是内存分配:

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

The exception that may be caught by the exception handler in this example is a `bad_alloc`. Because `bad_alloc` is derived from the standard base class `exception`, it can be caught (capturing by reference, captures all related classes).

在本例中，异常处理程序可能捕获的异常是' bad_alloc '。因为' bad_alloc '派生自标准基类' exception '，它可以被捕获(通过引用捕获，捕获所有相关类)。