Object-Oriented Programming

Lesson 14

Exception handling

Exceptions

If you encounter an exceptional situation in your code – that is, one where you don't have enough information in the current context to decide what to do – you can send information about the error into a larger context by creating an object containing that information and "throwing" it out of your current context. This is called throwing an exception.

If you're inside a function and you throw an exception, that function will exit in the process of throwing. If you don't want a throw to leave a function, you can set up a special block within the function where you try to solve your actual programming problem (and potentially generate exceptions). This is called the try block because you try your various function calls there.

Of course, the thrown exception must end up someplace. This is the exception handler, and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword catch.

Exceptions

Exception handling follows a different execution path of the normal program execution and is only used when problems during arise the course of the program

Advantages:

- An exception cannot be ignored (as opposite to an error code).

- The exception processing code is separated from the normal program code.

- Exceptions may permit to recover from problematic situations.

- Exceptions permit to construct more robust systems.

Exceptions

A situation is exceptional if there is not enough information to solve the problem in the current context.

When an exceptional situation is found it is possible to send information about the exception to a larger context.

For this an object is created that contains information about the error and this object is thrown out of the current context - throwing an exception.

```
class MyError {
       const char* const data;
public:
       MyError(const char* const msg = 0) : data (msg) {}
};
```

```
int& CVector::operator[](unsigned pos)
{      if (pos >= m_nElements)
          throw (MyError("Range exception in vector"));
       return m_arElements[pos];
}
```

The throw keyword

The keyword throw:

1) creates a copy of the object to be thrown;
2) destroys all local objects whose construction was completed by the time of throw - stack unwinding;
3) the object is, in effect, “returned” from the function, even though that object type isn’t normally what the function is designed to return (*where* you return to is someplace completely different than for a normal function call);
4) searches the nearest exception handler and transfers control to it.

You can throw as many different types of objects as you want. Typically, you’ll throw a different type for each different type of error. The idea is to store the information in the object and the *type* of object, so someone in the bigger context can figure out what to do with your exception.

The try keyword

If a function throws an exception, it must assume that exception is caught and dealt with.

If you’re inside a function and you throw an exception (or a called function throws an exception), that function will exit in the process of throwing. If you don’t want a **throw** to leave a function, you can set up a special block within the function where you try to solve your actual programming problem (and potentially generate exceptions). This is called the try block because you try your various function calls there.

```
try
{
    CVector v(3);
    v[10] = 7;
}
```

Instead of testing all possible errors that may occur, all potentially "dangerous" code is placed inside the try block without any error test.

The catch keyword

The thrown exception must end up someplace. This is the exception handler, and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword catch.

```
try
{
    CVector v(3);
    v[10] = 7;
}
catch(MyError& e)
{
   //handle the exception of type MyError
}
```

Exception handlers

The handlers must appear directly after the try block. If an exception is thrown, the exception handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled.

```
try
{
  // Code that may generate exceptions
}
catch(type1& id1)
{  // Handle exceptions of type1
}
catch(type2& id2)
{  // Handle exceptions of type2
}
catch(type3& id3)
{  // Handle exceptions of type3
}

// Normal execution resumes here...
```

To avoid that an additional copy of the exception is done, it is better to catch exceptions by reference.

Exception handlers

Automatic type conversion does not work with exceptions.

```
class Except1 {};
class Except2
{
  public:
     Except2(const Except1&) {}
};

void f() { throw Except1(); }
```

```
int main()
{
  try
  {
      f();
  }
  catch (Except2&)
  {
     cout << "Except2";
  }
  catch (Except1&)
  {
     cout << "Except1";
  }
}
```

This exception handler will be activated →

Exception handlers

A handler `catch(...)` will catch any exception:

```
catch(...)
{
  cout << "an exception was thrown" << endl;
}
```

Exception handlers

If none of the exception handlers following a particular try block matches an exception, that exception moves to the next-higher context, that is, the function or try block surrounding the try block that failed to catch the exception.

If no handler at any level catches the exception, it is “uncaught” or “unhandled” (programming error!). If an exception is uncaught, the special function `terminate()` is called which will call the function `abort()`.

The function `abort()` immediately exits the program with no calls to the normal termination procedures (which means that destructors for global and static objects might not be called).

The function `terminate()` is also called when a local object destructor launches an exception during stack unwinding.

Thus, a destructor that throws an exception or causes one to be thrown is a design error.

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Volume 2, chapter 7