Back to '4.2.2: Exceptions in C++'

C++ Programming: "Exception Handling"

Read this overview of the role of exceptions generated by the C++ library. This section also covers throw and try/catch concepts.

Exception Handling

Exception handling is a construct designed to handle the occurrence of exceptions, that is special conditions that changes the normal flow of program execution. Since when designing a programming task (a class or even a function), one cannot always assume that application/task will run or be completed correctly (exit with the result it was intended to). It may be the case that it will be just inappropriate for that given task to report an error message (return an error code) or just exit. To handle these types of cases, C++ supports the use of language constructs to separate error handling and reporting code from ordinary code, that is, constructs that can deal with these **exceptions** (errors and abnormalities) and so we call this global approach that adds uniformity to program design the **exception handling**.

An exception is said to be **thrown** at the place where some error or abnormal condition is detected. The throwing will cause the normal program flow to be aborted, in a **raised exception**. An exception is thrown programmatic, the programmer specifies the conditions of a throw.

In **handled exceptions**, execution of the program will resume at a designated block of code, called a **catch block**, which encloses the point of throwing in terms of program execution. The catch block can be, and usually is, located in a different function/method than the point of throwing. In this way, C++ supports *non-local error handling*. Along with altering the program flow, throwing of an exception passes an object to the catch block. This object can provide data that is necessary for the handling code to decide in which way it should react on the exception.

Consider this next code example of a **try** and **catch** block combination for clarification:

```
void AFunction()
{
    // This function does not return normally,
    // instead execution will resume at a catch block.
    // The thrown object is in this case of the type char const*,
    // i.e. it is a C-style string. More usually, exception
    // objects are of class type.
    throw "This is an exception!";
}

void AnotherFunction()
{
    // To catch exceptions, you first have to introduce
    // a try block via " try { ... } ". Then multiple catch
    // blocks can follow the try block.
    // " try { ... } catch(type 1) { ... } catch(type 2) { ... }"
    try
    {
        AFunction();
       // Because the function throws an exception,
       // the rest of the code in this block will not
       // be executed
    }
    catch(char const* pch)  // This catch block
                            // will react on exceptions
                            // of type char const*
    {
        // Execution will resume here.
        // You can handle the exception here.
    }
               // As can be seen
    catch(...) // The ellipsis indicates that this
               // block will catch exceptions of any type.
    {
       // In this example, this block will not be executed,
       // because the preceding catch block is chosen to
       // handle the exception.
```

```
    }
}
```

Unhandled exceptions on the other hand will result in a function termination and the stack will be unwound (stack allocated objects will have destructors called) as it looks for an exception handler. If none is found it will ultimately result in the termination of the program.

From the point of view of a programmer, raising an exception is a useful way to signal that a routine could not execute normally. For example, when an input argument is invalid (e.g. a zero denominator in division) or when a resource it relies on is unavailable (like a missing file, or a hard disk error). In systems without exceptions, routines would need to return some special error code. However, this is sometimes complicated by the semi-predicate problem, in which users of the routine need to write extra code to distinguish normal return values from erroneous ones.

Because it is hard to write exception safe code, you should only use an exception when you have to—when an error has occurred that you can not handle. Do *not* use exceptions for the normal flow of the program.

This example is *wrong*, it is a demonstration on what to avoid:

```
void sum(int iA, int iB)
{
    throw iA + iB;
}

int main()
{
    int iResult;

    try
    {
        sum(2, 3);
    }
    catch(int iTmpResult)
    {
        // Here the exception is used instead of a return value!
        // This is  wrong!
        iResult = iTmpResult;
    }

    return 0;
}
```

Stack unwinding

Consider the following code

```
void g()
{
    throw std::exception();
}

void f()
{
    std::string str = "Hello"; // This string is newly allocated
    g();
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    { }
}
```

The flow of the program:

- `main()` calls `f()`
- `f()` creates a local variable named `str`
- `str` constructor allocates a memory chunk to hold the string `"Hello"`
- `f()` calls `g()`
- `g()` throws an exception
- `f()` does not catch the exception.

Because the exception was not caught, we now need to exit `f()` in a clean fashion.
At this point, all the destructors of local variables previous to the throw
are called—This is called 'stack unwinding'.

- The destructor of `str` is called, which releases the memory occupied by it.

As you can see, the mechanism of 'stack unwinding' is essential to prevent resource leaks—without it, `str` would never be destroyed, and the memory it used would be lost until the end of the program (even until the next loss of power, or cold boot depending on the Operative System

memory management).

- `main()` catches the exception
- The program continues.

The 'stack unwinding' guarantees destructors of local variables (stack variables) will be called when we leave its scope.

Throwing objects

There are several ways to throw an exception object.

Throw a pointer to the object:

```
void foo()
{
    throw new MyApplicationException();
}

void bar()
{
    try
    {
        foo();
    }
    catch(MyApplicationException* e)
    {
        // Handle exception
    }
}
```

But now, who is responsible to delete the exception? The handler? This makes code uglier. There must be a better way!

How about this:

```
void foo()
{
    throw MyApplicationException();
}

void bar()
{
    try
    {
        foo();
    }
    catch(MyApplicationException e)
    {
        // Handle exception
    }
}
```

Looks better! But now, the catch handler that catches the exception, does it by value, meaning that a copy constructor is called. This can cause the program to crash if the exception caught was a `bad_alloc` caused by insufficient memory. In such a situation, seemingly safe code that is assumed to handle memory allocation problems results in the program crashing with a failure of the exception handler. Moreover, catching by value may cause the copy to have different behavior because of object slicing.

The correct approach is:

```
void foo()
{
    throw MyApplicationException();
}

void bar()
{
    try
    {
        foo();
    }
    catch(MyApplicationException const& e)
    {
        // Handle exception
    }
}
```

This method has all the advantages—the compiler is responsible for destroying the object, and no copying is done at catch time!

The conclusion is that exceptions should be thrown by value, and caught by (usually const) reference.

Constructors and destructors

When an exception is thrown from a constructor, the object is not considered instantiated, and therefore its destructor will *not* be called. But all destructors of already successfully constructed base and member objects of the same master object will be called. Destructors of not yet constructed base or member objects of the same master object will not be executed. Example:

```
class A : public B, public C
{
public:
    D sD;
    E sE;
    A(void)
    :B(), C(), sD(), sE()
    {
    }
};
```

Let's assume the constructor of base class C throws. Then the order of execution is:

- `B`
- `C` (throws)
- `~B`

Let's assume the constructor of member object sE throws. Then the order of execution is:

- `B`
- `C`
- `sD`
- `sE` (throws)
- `~sD`
- `~C`
- `~B`

Thus if some constructor is executed, one can rely on that all other constructors of the same master object executed before, were successful. This enables one, to use an already constructed member or base object as an argument for the constructor of one of the following member or base objects of the same master object.

What happens when we allocate this object with `new`?

- Memory for the object is allocated
- The object's constructor throws an exception
  - The object was not instantiated due to the exception
- The memory occupied by the object is deleted
- The exception is propagated, until it is caught

The main purpose of throwing an exception from a constructor is to inform the program/user that the creation and initialization of the object did not finish correctly. This is a very clean way of providing this important information, as constructors do not return a separate value containing some error code (as an initialization function might).

In contrast, it is strongly recommended not to throw exceptions inside a destructor. It is important to note when a destructor is called:

- as part of a normal deallocation (exit from a scope, delete)
- as part of a stack unwinding that handles a previously thrown exception.

In the former case, throwing an exception inside a destructor can simply cause memory leaks due to incorrectly deallocated object. In the latter, the code must be more clever. If an exception was thrown as part of the stack unwinding caused by another exception, there is no way to choose which exception to handle first. This is interpreted as a failure of the exception handling mechanism and that causes the program to call the function terminate.

To address this problem, it is possible to test if the destructor was called as part of an exception handling process. To this end, one should use the standard library function uncaught_exception, which returns true if an exception has been thrown, but hasn't been caught yet. All code executed in such a situation must not throw another exception.

Situations where such careful coding is necessary are extremely rare. It is far safer and easier to debug if the code was written in such a way that destructors did not throw exceptions at all.

Writing exception safe code

Exception safety

A piece of code is said to be **exception-safe**, if run-time failures within the code will not produce ill effects, such as memory leaks, garbled stored data, or invalid output. Exception-safe code must satisfy invariants placed on the code even if exceptions occur. There are several levels of exception safety:

1. **Failure transparency**, also known as the **no throw guarantee**: Operations are guaranteed to succeed and satisfy all requirements even in presence of exceptional situations. If an exception occurs, it will not throw the exception further up. (Best level of exception safety.)
2. **Commit or rollback semantics**, also known as **strong exception safety** or **no-change guarantee**: Operations can fail, but failed operations are guaranteed to have no side effects so all data retain original values.
3. **Basic exception safety**: Partial execution of failed operations can cause side effects, but invariants on the state are preserved. Any stored data will contain valid values even if data has different values now from before the exception.
4. **Minimal exception safety** also known as **no-leak guarantee**: Partial execution of failed operations may store invalid data but will not cause a crash, and no resources get leaked.
5. **No exception safety**: No guarantees are made. (Worst level of exception safety)

Partial handling

Consider the following case:

```
void g()
{
    throw "Exception";
}

void f()
{
    int* pI = new int(0);
    g();
    delete pI;
}

int main()
{
    f();
    return 0;
}
```

Can you see the problem in this code? If `g()` throws an exception, the variable `pI` is never deleted and we have a memory leak.

To prevent the memory leak, `f()` must catch the exception, and delete `pI`. But `f()` can't handle the exception, it doesn't know how!

What is the solution then?

`f()` shall catch the exception, and then re-throw it:

```
void g()
{
    throw "Exception";
}

void f()
{
    int* pI = new int(0);

    try
    {
        g();
    }
    catch (...)
    {
        delete pI;
        throw; // This empty throw re-throws the exception we caught
               // An empty throw can only exist in a catch block
    }

    delete pI;
}

int main()
{
    f();
    return 0;
}
```

There's a better way though; using **RAII** classes to avoid the need to use exception handling.

Guards

If you plan to use exceptions in your code, you must always try to write your code in an exception safe manner. Let's see some of the problems that can occur:

Consider the following code:

```
void g()
{
    throw std::exception();
}

void f()
{
    int* pI = new int(2);

    *pI = 3;
    g();
    // Oops, if an exception is thrown, pI is never deleted
    // and we have a memory leak
    delete pI;
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    { }

    return 0;
}
```

Can you see the problem in this code? When an exception is thrown, we will never run the line that deletes `pI`!

What's the solution to this? Earlier we saw a solution based on `f()` ability to catch and re-throw. But there is a neater solution using the 'stack unwinding' mechanism. But 'stack unwinding' only applies to destructors for objects, so how can we use it?

We can write a simple wrapper class:

```
// Note: This type of class is best implemented using templates, discussed in the next chapter.
class IntDeleter {
public:
   IntDeleter(int* piValue)
   {
       m_piValue = piValue;
   }

   ~IntDeleter()
   {
       delete m_piValue;
   }

   // operator *, enables us to dereference the object and use it
   // like a regular pointer.
   int&  operator *()
   {
       return *m_piValue;
   }

private:
    int* m_piValue;
};
```

The new version of `f()`:

```
void f()
{
  IntDeleter pI(new int(2));

  *pI = 3;
  g();
  // No need to delete pI, this will be done in destruction.
  // This code is also exception safe.
}
```

The pattern presented here is called a *guard*. A guard is very useful in other cases, and it can also help us make our code more exception safe. The guard pattern is similar to a *finally* block in other languages.

Note that the C++ Standard Library provides a templated guard by the name of unique_ptr.

Exception hierarchy

You may throw as exception an object (like a class or string), a pointer (like `char*`), or a primitive (like `int`). So, which should you choose? You should throw objects, as they ease the handling of exceptions for the programmer. It is common to create a class hierarchy of exception classes:

- class MyApplicationException {};
  - class MathematicalException : public MyApplicationException {};
    - class DivisionByZeroException : public MathematicalException {};
  - class InvalidArgumentException : public MyApplicationException {};

An example:

```
float divide(float fNumerator, float fDenominator)
{
    if (fDenominator == 0.0)
    {
        throw DivisionByZeroException();
    }

    return fNumerator/fDenominator;
}

enum MathOperators {DIVISION, PRODUCT};

float operate(int iAction, float fArgLeft, float fArgRight)
{
    if (iAction == DIVISION)
    {
        return divide(fArgLeft, fArgRight);
    }
    else if (iAction == PRODUCT))
    {
        // call the product function
        // ...
    }

    // No match for the action! iAction is an invalid agument
    throw InvalidArgumentException();
}

int main(int iArgc, char* a_pchArgv[])
{
    try
    {
        operate(atoi(a_pchArgv[0]), atof(a_pchArgv[1]), atof(a_pchArgv[2]));
    }
    catch(MathematicalException& )
    {
        // Handle Error
    }
```

```
    catch(MyApplicationException& )
    {
        // This will catch in InvalidArgumentException too.
        // Display help to the user, and explain about the arguments.
    }

    return 0;
}
```

The order of the catch blocks is important. A thrown object (say, InvalidArgumentException) can be caught in a catch block of one of its super-classes. (e.g. `catch (MyApplicationException& )` will catch it too). This is why it is important to place the catch blocks of derived classes before the catch block of their super classes.

Exception specifications

The use of exception specifications has been deprecated in the new standard C++11. It is recommended that nobody use them. They are included here for historical reasons(not everybody is using C++11 yet)

The range of exceptions that can be thrown by a function are an important part of that function's public interface. Without this information, you would have to assume that any exception could occur when calling any function, and consequently write code that was extremely defensive. Knowing the list of exceptions that can be thrown, you can simplify your code since it doesn't need to handle every case.

This exception information is specifically part of the *public* interface. Users of a class don't need to know anything about the way it is implemented, but they do need to know about the exceptions that can be thrown, just as they need to know the number and type of parameters to a member function. One way of providing this information to clients of a library is via code documentation, but this needs to be manually updated very carefully. Incorrect exception information is worse than none at all, since you may end up writing code that is less exception-safe than you intended to.

C++ provides another way of recording the exception interface, by means of *exception specifications*. An exception specification is parsed by the compiler, which provides a measure of automated checking. An exception specification can be applied to any function, and looks like this:

```
double divide(double dNumerator, double dDenominator) throw (DivideByZeroException);
```

You can specify that a function cannot throw any exceptions by using an empty exception specification:

```
void safeFunction(int iFoo) throw();
```

Shortcomings of exception specifications

C++ does not programmatically enforce exception specifications at compile time. For example, the following code is legal:

```
void DubiousFunction(int iFoo) throw()
{
    if (iFoo < 0)
    {
        throw RangeException();
    }
}
```

Rather than checking exception specifications at compile time, C++ checks them at run time, which means that you might not realize that you have an inaccurate exception specification until testing or, if you are unlucky, when the code is already in production.

If an exception is thrown at run time that propagates out of a function that doesn't allow the exception in its exception specification, the exception will not propagate any further and instead, the function `RangeException()` will be called. The `RangeException()` function doesn't return, but can throw a different type of exception that may (or may not) satisfy the exception specification and allow exception handling to carry on normally. If this still doesn't recover the situation, the program will be terminated.

Many people regard the behavior of attempting to translate exceptions at run time to be worse than simply allowing the exception to propagate up the stack to a caller who may be able to handle it. The fact that the exception specification has been violated does not mean that the caller *can't* handle the situation, only that the author of the code didn't expect it. Often there will be a `catch (...)` block somewhere on the stack that can deal with *any* exception.

Some coding standards require that exception specifications are not used. In the C++ language standard C++11(C++0x), the use of exception specifications as specified in the current version of the standard (C++03), is deprecated. The use of exception specifications has been deprecated entirely, meaning that it is highly recommended that nobody use them.

Last modified: Monday, November 6, 2017, 1:20 PM