

CPSC 427: Object-Oriented Programming

Michael J. Fischer

Preview Lecture 20
November 12, 2018

Rethrowing Exceptions

Uncaught Exceptions

Singleton Design Pattern

Smart Pointer Demo

Rethrowing Exceptions

Rethrow

A `catch` block can do some processing and then optionally rethrow the exception or throw a new exception.

- ▶ One exception can cause multiple `catch` blocks to execute.
- ▶ To rethrow the same exception, use `throw;` with no argument.
- ▶ To throw a new exception, use `throw` as usual with an argument.

A subtle fact about rethrow

Rethrowing the current exception is not the same as throwing an exception with the same exception object.

`throw e;` always **copies** object `e` to special memory using the copy constructor for `e`'s class.

`throw;` **does not make another copy** of the exception object but instead uses the copy already in special memory.

This difference becomes apparent if the copy is not identical to the original (possible for a custom copy constructor), or if the copy constructor has side effects (such as printing output).

Example of rethrowing an exception (demo 20a-Exceptions-throw)

```
1  #include <iostream>
2  using namespace std;
3  class MyException {
4  public:
5      MyException() {}
6      MyException( MyException& e ) {
7          cout << "Copy constructor called\n"; }
8      ~MyException() {}
9  } myex; // declares class and instantiates it
10
11 int main () {
12     try {
13         try { throw myex; }
14         catch (MyException& e) {
15             cout << "Exception caught by inner catch\n"; throw; }
16         }
17     catch (MyException& err) {
18         cout << "Exception caught by outer catch\n";
19     }
20     return 0;
21 }
```



Results

In the preceding example, the `throw myex` on line 12 causes a copy, but the `throw` on line 14 does not.

This produces the following output:

```
Copy constructor called  
Exception caught by inner catch  
Exception caught by outer catch
```

Uncaught Exceptions

Uncaught exceptions: Ariane 5

Uncaught exceptions have led to spectacular disasters.

The European Space Agency's Ariane 5 Flight 501 was destroyed 40 seconds after takeoff (June 4, 1996). The US\$1 billion prototype rocket self-destructed due to a bug in the on-board guidance software. [[Wikipedia](#)]

This is not about a programming error.

It is about system-engineering and **design failures**.

The software did what it was designed to do and what it was agreed that it should do.

Uncaught exceptions: Ariane 5 (cont.)

Heres a summary of the events and its import for system engineering:

- ▶ A decision was made to leave a program running after launch, even though its results were not needed after launch.
- ▶ An overflow error happened in that calculation,
- ▶ An exception was thrown and, by design, was not caught.
- ▶ This caused the vehicle's active and backup inertial reference systems to shut down automatically.

As the result of the unanticipated failure mode and a diagnostic message erroneously treated as data, the guidance system ordered violent attitude correction. The ensuing disintegration of the over-stressed vehicle triggered the pyrotechnic destruction of the launcher and its payload.

Termination

There are various conditions under which the exception-handling mechanism can fail. Two such examples are:

- ▶ Exception not caught by any `catch` block.
- ▶ A destructor issues a `throw` during the stack-unwinding process.

When this happens, the function `terminate()` is called, which by default aborts the process.¹

This is a **bad thing** in production code.

Conclusion: *All exceptions should be caught and dealt with explicitly.*

¹It's behavior can be changed by the user.

Singleton Design Pattern

Unique IDs

Unique identifiers (UIDs) are familiar in many situations: Appliance serial numbers, automobile VINs, social security numbers, and so forth.

They are useful in programming as well. Whenever a class has many instances, UIDs can help track objects from the time they are created until their eventual deletion. This is especially helpful when custody changes during the lifetime of the object.

UIDs are very helpful in identifying error comments during debugging. They are also helpful when included in log files.

How to generate UUIDs

A method for adding UUIDs to class instances involves the following steps:

1. Add a data member `const unsigned uid` to the class.
2. Add a static variable `unsigned nextUID` to the class.
3. Initialize `nextUID` to 0.
4. In every constructor, initialize `uid` to `nextUID++`.

Recall that static variables cannot be initialized within the class but rather in a `.cpp` file.

Drawbacks to the simple method

Drawbacks to the simple approach:

- ▶ It adds clutter to the class.
- ▶ It violates OO principles by mixing together the UID generation process with whatever else the class is doing.
- ▶ It results in code replication if UIDs are being used in more than one class.

A UID generator

What we want is a class `Serial` with a private data member `nextUID` and a public function `uidGen()` that returns and updates the next UID.

In order to call the function, we need a class instance `uidGen` of `Serial` that initializes `nextUID` and supports the public function `uidGen()`. Now, to generate a new serial number, simply call `uidGen.uidGen()`.

However, this solution has two problems:

1. How can one make the object `uidGen` available wherever needed?
2. Where should `Serial` be instantiated?

Singleton class

A singleton class solves both problems.



1. It has a static function that returns a pointer to the single instantiation whenever it is called.
2. Initially there is no instantiation, so it creates and remembers an instantiation the first time it is called. It uses a private static variable for this purpose.

Functors

A **functor** is an object that acts like a function.

Let `obj` be a functor. Then one can write `obj()`, pretending that `obj` is a function.


All that is needed to make this work is to define `operator()` within the class.

For our UID generator, we define the behavior of `obj` to be the same as for `uidGen()` discussed above.

Serial.hpp

```
// Singleton class for generating unique ID's

class Serial {
private:
    static Serial* Sobj;
    int nextUID=0;
    Serial() =default;
public:
    static Serial& uidGen() {
        if (Sobj == nullptr) Sobj = new Serial;
        return *Sobj;
    }
    const int operator()() { return nextUID++; }
};
```



Serial.cpp

```
// Initialize Serializer static variable  
Serial* Serial::Sobj = nullptr;
```

Smart Pointer Demo

Dangling pointers

Pointers can be used to permit object sharing from different contexts.

One can have a single object of some type `T` with many pointers in different contexts that all point to that object.

Problems with shared objects

If the different contexts have different lifetimes, the problem is to know when it is safe to delete the object.

It can be difficult to know when an object should be deleted.

Failure to delete an object will cause **memory leaks**.

If the object is deleted while there are still points pointing to it, then those pointers become invalid. We call these **dangling pointers**.

Failure to delete or premature deletion of objects are common sources of errors in C++.

Avoiding dangling pointers

There are several ways to avoid dangling pointers.

1. Have a top-level manager whose lifetime exceeds that of all of the pointers take responsibility for deleting the objects.
2. Use a garbage collection. (This is java's approach.)
3. Use reference counts. That is, keep track somehow of the number of outstanding pointers to an object. When the last pointer is deleted, then the object is deleted at that time.

Modern C++ Smart Pointers

Modern C++ has three kinds of **smart pointers**. These are objects that act very much like raw pointers, but they take responsibility for managing the objects they point at and deleting them when appropriate.

- ▶ `shared_ptr`
- ▶ `weak_ptr`
- ▶ `unique_ptr`

We will discuss them later in the course, time permitting. For now, we present a simplified version of shared pointer so that you can see the basic mechanism that underlies all of the various kinds of shared pointers.

Smart pointers

We define a class `SPtr` of reference-counted pointer-like objects.

An `SPtr` should act like a pointer to a `T`.

This means if `sp` is an `SPtr`, then `*sp` is a `T&`.

We need a way to create a smart pointer and to create copies of them.

Demo `20b-SmartPointer` illustrates how this can be done.