FE545 Design, Patterns and Derivatives Pricing

Design Pattern Revisited & Exception Handling

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

04/16/2024

Overview

Introduction to Exceptions

- Up till now we have focused on clarity and code reusability, we have not considered how to cope with things going wrong at run time.
- The mechanism in C++ designed for coping with errors is throwing an exception.
- Writing code that functions well in the presence of exceptions raises a host of issues that did not exist before.
  - Exceptions are raised by the *throw* command. We specify as an argument an object X, of any time Y.
  - Execution then immediately moves to the end of the current scope and objects going out of scope are destroyed.
  - If there is a catch command at the end of the scope, which catches objects of type Y, then control passes to the scope of the catch command.
  - Otherwise, then control passes to the end of the enclosing scope, and this keeps happening until the exception is caught, or the enclosing scope is the end of the program and execution terminates, i.e. your program crashes.

Introduction to Exceptions

- The great value of the approach is that we do not have to test the return value of every function or method call to ensure that the last call did not generate an error.
- The great downside is that code execution order becomes a lot less predictable, and this can cause problems. This is particularly a problem when memory allocation is in use. Consider the following code:

```
double evaluate(const PayOff& p, double y)
{
     PayOff* payOffPtr = p.clone();
     double x = (*payOffPtr)(y);
     delete payOffPtr;
     return x;
}
```

Introduction to Exceptions

```
double evaluate(const PayOff& p, double y)
{
     PayOff* payOffPtr = p.clone();
     double x = (*payOffPtr)(y);
     delete payOffPtr;
     return x;
}
```

- This a little artificial in that no useful purpose is served by making a copy, but if *operator*() were non-const, this could be useful.
- It serves to illustrate a point: it is possible that calling *operator*() will throw an exception.
- This will be caught somewhere outside the function.
- The catcher will have no idea that *payOffPtr* needs to be deleted.
- If this happens enough times, your application will run out of memory and crash.

Safety Guarantees

- There are two standard safety guarantees:
  - The *weak* guarantee: the object and program are left in a valid state, and no resources have been leaked.
  - The *strong* guarantee: if an exception is thrown during an operation (i.e. a call to a method or a function), then the program is left in the state it was at entry to the operation.
- The essential difference here is that with the weak guarantee an object's state can change even though the operation failed, whereas with the strong guarantee the class is promising to undo all changes before throwing.
- The strong guarantee is harder to implement than the weak one. However, it is important to realize that code that is not written with exception safety in mind will satisfy neither.

The Use of Smart Pointers

- Consider again the example of the introduction. What we want to happen is that when the function is exited the memory allocated by the clone command is deallocated by a call to *delete*.
- Exiting can occur either in the conventional way via the return statement, or by the exception being thrown.
- For both of these, all automatic (i.e. ordinary local) variables are destroyed at the end of the scope. So the solution is to make the deletion a side-effect of these destructions.
- We have already looked at one smart pointer $Wrapper < T >$. If we use it here, the code snippet becomes:

```
double evaluate(const PayOff& p, double y)
{
        Wrapper<PayOff> payOffPtr(p);
        double x = (*payOffPtr)(y);
        return x;
 }
```

The Use of Smart Pointers

- Recall that the Wrapper class will call the clone method internally. The *delete* command is no longer necessary because the destructor of *Wrapper* calls it automatically.
- As written, $Wrapper < T >$ cannot be used to take ownership of a raw pointer since it has no constructors that take pointers. However, we can easily add to the file *Wrapper.h* an extra constructor in the public section of the class:

```
Wrapper(T* DataPtr_)
{
        DataPtr = DataPtr_;
}
```

- and then it would be legitimate to code

```
PayOff* payPtr1 = p.clone();
Wrapper<PayOff> payOffPtr(payPtr1);
double x = (*payOffPtr)(y);
return x;
```

The Use of Smart Pointers

- The *Wrapper* $< T >$ class is just one example of a smart pointer. There are many examples both in Boost and the standard library. There are four obvious solutions to copying:
  - Copy the pointed-to object.
  - Make copying illegal.
  - Have the pointers share ownership of the object.
  - Transfer ownership of the pointer to the new object.
- The alternative is used by the *unique_ptr* (*auto_ptr* is deprecated in c++11) class in the standard library defined the the file memory. As with all the other smart pointers, when it goes out of scope the pointed-to object is deleted. However, suppose we code the following

```
double evaluate(const PayOff& p, double y)
{
        std::unique_ptr<PayOff> payPtr1 = p.clone();
        double z = (*payPtr1)(y);
        std::unique_ptr<PayOff> payPtr2(payPtr1);
        double x = (*payPtr1)(y);
        return x+z;
}
```

The Use of Smart Pointers

- *shared_ptr* : If you are concerned about freeing of resource/memory AND if you have more than one function that could be using the object AT-DIFFERENT times, then go with *shared_ptr*.

** By DIFFERENT-Times, think of a situation where the object-ptr is stored in multiple data-structure and later accessed. Multiple threads, of course is another example.

- *unique_ptr* : If all you are concerned is freeing memory, and the access to object is SEQUENTIAL, then go for *unique_ptr*.

** By SEQUENTIAL, I mean, at any point object will be accessed from one context. E.g. a object that was created, and used immediately after creation by the creator. After creation the object is stored in FIRST data-structure. Then either the object is destroyed after the ONE data-structure or is moved to SECOND data-structure.

The Rule of Almost Zero

- The rule of almost zero does not contradict the rule of three but supersedes it by saying that you should always be in the case of not defining any of them.
- How do we avoid the shallow copy problem discussed earlier? We use smart pointers to ensure that a shallow copy is sufficient.
  - Every data member will be either an ordinary object which can be copied, or a smart pointer which is copied and assigned in the fashion we have chosen.
  - So if we want objects to be shared between copies, we use *shared_ptr*; if we want to make copying illegal, we use *scope_ptr*; and if we want the pointed-to objects to be cloned, we use *Wrapper*.
  - There will be no memory leak issues because the smart pointers delete the pointed-to objects when the compiler-generated destructor is called.
  - We do not waste time writing copy constructors or assignment operators, and we do not have to remember to update them when we change the data members of the class.

Commands to Never Use

- Some commands to never use are:
  - *malloc*
  - *free*
  - *delete*
  - *new*[]
  - *delete*[]
- The first two of these are C commands not C++, and have been superseded by the versions of *new* and *delete*.
- The *delete* command is never necessary because of smart pointers. As long as you ensure that anything created by *new* is owned by a smart pointer, you need never code *delete*.
- The only time we use the *delete* command is where we write a smart pointer. However, with the advent of *Boost*, you should never need to do this.
- Since we never use *new*[], we never need to use *delete*[], and that allows the avoidance of nasty bugs caused by accidentally using the wrong version of the *delete* command.

Making the Wrapper Class Exception Safe

- We have argued that we should use smart pointers since they make exception safety easy. However, we must also make sure that the smart pointers we write for ourselves are exception safe.
- In fact the *Wrapper* class as we originally wrote is not exception safe. Consider the assignment operator

```
Wrapper& operator=(const Wrapper<T>& original)
{
    if (this != &original)
    {
        if (DataPtr!=0)
            delete DataPtr;

        DataPtr = (original.DataPtr !=0) ? original.DataPtr->clone() : 0;
    }

    return *this;
}
```

Making the Wrapper Class Exception Safe

- If we call to the *clone* method of the *original* object passed in throws, we have a problem.
- We have already deleted *DataPtr* so the strong guarantee is violated. But worse, any attempt to access the underlying object will be an attempt to access a dead object, and we can expect a crash.
- We therefore need to recode the assignment operator in *Wrapper* to avoid this problem.

```
Wrapper& operator=(const Wrapper<T>& original)
{
    if (this != &original)
    {
        T* newPtr = (original.DataPtr != 0) ? original.DataPtr->clone() :0;

        if (DataPtr!=0)
            delete DataPtr;

        DataPtr = newPtr;
    }

    return *this;
}
```

Throwing in Special Functions

- There is the issue of what to do when an error occurs in a constructor or destructor. In summary, it is okay to throw in a constructor but never throw in a destructor.
- The main danger of throwing in a constructor is that resources acquired may not be released.
- If the destructor carries out some non-trivial operations such as calling *delete*, we have a problem. This can be tackled in two ways:
  - The first is simply to do any tidying up that the destructor would have done before calling throw.
  - The second is to follow the rule of almost zero and have a trivial destructor. The second approach is much safer in that exceptions could arise in unexpected places.
  - One subtlety to be aware of is that the constructor of one of the data members of the class could also throw. These constructors are all called before the main routine is entered.

Throwing in Special Functions

- What about destructor? Suppose we write a destructor for a class A that throws when it is unhappy. We let B have data member of type A. Now consider the following snippet:

```
bool flag = true;
try
{
        {
          B testObject;
          if (flag)
                throw("flag is true");
        }
}
catch( ...)
{
}
```

- When the *throw* is called, the stack is unwound and the object of type B is destroyed. As part of this is destruction, its data-member of type A is destroyed.
- If the destructor of A throws, the application terminates, i.e. crashes. This is is specified in the C++ standard; the reason being that the compiler will not know which exception to deal with.

Breakout Exercise: Wrapper Class

- We have a new *Wrapper* class that will handle the memory management of the PayOff*.
  - Spot: 50
  - Strike: 50
  - Strike: 51
  - Strike: 52
- Please use the given classes and try to implement throw an exception and catch it.
- Please try to throw an exception in the destructor and see what will happen (artificially throw an exception).

QuantLib Project

- The QuantLib project is aimed at providing a comprehensive software framework for quantitative finance. QuantLib is a free/open-source library for modeling, trading, and risk management in real-life.
- The reposit project facilitates deployment of object libraries to end user platforms and is used to generate QuantLibXL, an Excel add-in for QuantLib, and QuantLibAddin, QuantLib addins for other platforms such as LibreOffice Calc.
- The library could be exploited across different research and regulatory institutions, banks, software companies, and so on. Being a free/open-source project, quants contributing to the library would not need to start from scratch every time.
  - Students could master a library that is actually used in the real world and contribute to it in a meaningful way. This would potentially place them in a privileged position on the job market.

QuantLib Project

- The library could be exploited across different research and regulatory institutions, banks, software companies, and so on. Being a free/open-source project, quants contributing to the library would not need to start from scratch every time.
  - Researchers would have a framework at hand, which vastly reduces the amount of low-level work necessary to build models, so to be able to focus on more complex and interesting problems.
  - Financial firms could exploit QuantLib as base code and/or benchmark, while being able to engage in creating more innovative solutions that would make them more competitive on the market.
  - Regulatory institutions may have a tool for standard pricing and risk management practices.

QuantLib Installation

- Boost Installation:
  - The preferred way to get Boost is through Homebrew (http://brew.sh/). By default, Homebrew will install Boost in /usr/local. From Terminal, run:

    brew install boost

- QuantLib Installation
  - Installation from Homebrew: If you don't need to modify the library, you might want to skip the compilation and install a precompiled binary version. Unofficial binaries are available from Homebrew; run

    brew install quantlib

- Alternatively, you can use MacPorts (http://www.macports.org/) which installs in /opt/local instead.

QuantLib Installation

- That will then build and put all the files in /usr/local/Cellar/quantlib under some version number that is not of importance. The important thing is the the tools are then linked into /usr/local/bin so all you need to do is make sure that /usr/local/bin is in your PATH.
  - That gives you access to the tool quantlib-config which is always linked to the latest version and it knows which version that is. So, if you run:

    quantlib-config –cflags

  - Likewise, if you run:

    quantlib-config –libs

QuantLib Setup

Makefile

```
PROJECT_ROOT = $(dir $(abspath $(lastword $(MAKEFILE_LIST))))
CXX = g++

OBJS = quant_lib_options.o

ifeq ($(BUILD_MODE),debug)
 CFLAGS += -g -std=c++11 -stdlib=libc++ -mmacosx-version-min=10.9
 CPLUS_INCLUDE_PATH=/opt/local/include -I/opt/local/libexec/boost/1.76/include
 DYLD_LIBRARY_PATH=/opt/local/lib  -L/opt/local/libexec/boost/1.76/lib -lQuantLib
 LDFLAGS += -stdlib=libc++ -mmacosx-version-min=10.9
else ifeq ($(BUILD_MODE),run)
 CFLAGS += -O2 -std=c++11 -stdlib=libc++ -mmacosx-version-min=10.9
 CPLUS_INCLUDE_PATH=/opt/local/include -I/opt/local/libexec/boost/1.76/include
 DYLD_LIBRARY_PATH=/opt/local/lib  -L/opt/local/libexec/boost/1.76/lib -lQuantLib
 LDFLAGS += -stdlib=libc++ -mmacosx-version-min=10.9
else ifeq ($(BUILD_MODE),linuxtools)
 CFLAGS += -g -pg -fprofile-arcs -ftest-coverage
 LDFLAGS += -pg -fprofile-arcs -ftest-coverage
 EXTRA_CLEAN += quant_lib_options.gcda quant_lib_options.gcno $(PROJECT_ROOT)gmon.out
 EXTRA_CMDS = rm -rf quant_lib_options.gcda
else
    $(error Build mode $(BUILD_MODE) not supported by this Makefile)
endif

all: quant_lib_options

quant_lib_options: $(OBJS)
 $(CXX) $(LDFLAGS) -L$(DYLD_LIBRARY_PATH) -o $@ $^
 $(EXTRA_CMDS)
...
```

Key Points

- Exceptions can cause memory leaks.
- The weak or basic exception safety guarantee says that a program will be in a valid state after an exception is thrown.
- The strong exception safety guarantee says that if an exception is thrown during an operation, then the program will be left in the state it was in at the start of the operation.
- The memory leaks can be avoided by the use of smart pointers.
- The rule of almost zero advises never to write code that requires non-trivial copy constructors, assignment operators, and destructors.
- Avoid the *new* [], *delete* and *delete* [] commands.
- We have to take care when writing the assignment operators of smart pointers to avoid memory leaks when *new* fails.
- Floating point errors do not by default cause C++ exceptions but they can be made to do so.