Design Patterns in C++

Safety to exceptions

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

April 15, 2011

Exception safety

- Informally, "exception safety" means that a piece of software is "well-behaved" with respect to exceptions
- The program must remain in a consistent state when an exception is raised (no memory leaks, no half modified structures, etc.)
- Exception safety is always important,
  - but it is particularly important when writing libraries, because the customer expects a well behaved library to not lose resources or remain inconsistent when an exception is raised
  - the user expects to catch the exception and continue with the program, so the data structures must remain consistent
- Exception safety is particularly difficult when dealing with templates, because we do not know the types, so we do not know what can raise an exception

An example of unsafe code

```
//  In some header file:
void f( T1*, T2* );

//  In some implementation file:
f( new T1, new T2 );
```

- This can lead to a memory leak. Suppose that the evaluation order is the following:
  1. Memory is allocated for T1, then T1 is constructed
  2. Memory is allocated for T2, then T2 is constructed
  3. function f() is called
- If the constructor of T2 throws an exception for whatever reason, the memory for T1 is not deallocated, and T1 is not destroyed
- **Rule:** don't use two `new` operations in the same expression, because some of them can leak in case of exception
- notice that we do not know the exact order of execution of steps 1 and 2.
- see example `exc_function`

Definition

- There are three properties that have to do with exception safety (**Abrahams Guarantees**)
- **Basic Guarantee:** If an exception is thrown, no resources are leaked, and objects remain in a destructible and usable – but not necessarily predictable – state.
  - This is the weakest usable level of exception safety, and is appropriate where client code can cope with failed operations that have already made changes to objects' state.
- **Strong Guarantee:** If an exception is thrown, program state remains unchanged.
  - This level always implies global commit-or-rollback semantics, including that no references or iterators into a container be invalidated if an operation fails.
- **Nothrow Guarantee:** The function will not emit an exception under any circumstances.
  - It turns out that it is sometimes impossible to implement the strong or even the basic guarantee unless certain functions are guaranteed not to throw (e.g., destructors, deallocation functions).
  - As we will see below, an important feature of the standard `auto_ptr` is that no `auto_ptr` operation will throw.

Problem

- How to write a generic implementation of the Stack container that is safe to exceptions?
- We will solve the problem step by step (see “Exceptional C++”, by Herb Sutter)

```
template <class T>
class Stack {
public:
    Stack();
    ~Stack();
    ...
private:
    T*     v_;
    size_t vsize_;
    size_t vused_;
};
```


Requirements

- first requirement: **weak guarantee** (no memory leak)
- second requirement: **strong guarantee** (state is always consistent)
- third requirement: **transparency** (all exceptions must be forwarded to the user of Stack)

- Let’s start by writing the constructor and destructor

Constructor

```
template<class T>
Stack<T>::Stack() :
    v_(0),
    vsize_(10),
    vused_(0)
{
    v_ = new T[vsize_];
}
```

- The implementation is correct because:
  - It is transparent (no exception is caught)
  - no memory leak
  - if an exception is thrown, Stack is not constructed, so everything remains in consistent state

Memory leak?

```
    v_ = new T[vsize_];
```

- To understand why there are no memory leaks, let's see how the new operator works
- The new operator
  1. first allocates memory for 10 objects of type T
  2. then, it invokes the default constructor of T 10 times
- Both (1) and (2) can throw an exception
  - if (1) throws a `bad_alloc` (because we run out of memory), no memory is allocated, everything works correctly
  - if any of the default constructors of T throw an exception the previously constructed objects are destructed and the memory is released
- **Warning:** the destructors **must not throw** exceptions, otherwise it is impossible to build anything that is safe to exceptions
  - We implicitly require that `T::~T()` does not throw

Destructor

```
template<class T>
Stack<T>::~Stack()
{
    delete v_[];
}
```

- We just said that we require that `T::~T()` will never throw an exception
- if this is true, also `delete` cannot throw, and also our destructor
- therefore, this function respects the **no-throw guarantee**

Copy and assignment

- We now write the copy constructor and the assignment operator

```
template <class T>
class Stack {
public:
    Stack();
    ~Stack();
    Stack(const Stack &);
    Stack & operator=(const Stack &);
    ...
private:
    T*     v_;
    size_t vsize_;
    size_t vused_;
};
```

First try

- To simplify the implementation, we first implement an helper function that performs the copy, this will be used by the copy constructor and by the assignment operator

```
template<class T>
T* Stack<T>::NewCopy(const T* src,
                     size_t srcsize,
                     size_t destsize)
{
    assert(destsize >= srcsize);
    T *dest = new T[destsize];
    try {
        copy(src, src+destsize, dest);
    } catch (...) {
        delete dest[]; // cannot raise exceptions
        throw;         // re-throws the same exception
    }
    return dest;
}
```


Analysis

```
    ...
    T *dest = new T[destsize];
    ...
```

- If it throws an exception, the rest of the function is not executed, all memory is correctly deallocated (as in the constructor case)

```
    try {
        copy(src, src+destsize, dest);
    } catch (...) {
        delete dest[]; // cannot raise exceptions
        throw;         // re-throws the same exception
    }
```

- If the copy throws an exception (due to the assignment operator of T) we catch it, delete all memory, and re-throw (for transparency)

Assignment

```
template<class T>
Stack<T>& Stack<T>::operator=(const Stack<T>&other)
{
    if (this != other) {
        T* v_new = NewCopy(other_v, other.vsize_, other.vsize_);
        delete v_[];
        v_ = v_new;
        vsize_ = other.vsize_;
        vused _ = other.v_used_;
    }
    return *this;
}
}
```

- If `NewCopy` throws, nothing else is changed
- all other instructions cannot throw (they operate on std types)

Push and pop implementation

- We need extra care here. Suppose initially that we have the following prototype:

```
template <class T>
class Stack {
public:
    Stack();
    ~Stack();
    Stack(const Stack &);
    Stack & operator=(const Stack &);
    size_t Count() const { return vused_; }
    void Push(const T &);
    T Pop();
private:
    T*     v_;
    size_t vsize_;
    size_t vused_;
};
```

Push

- Again, let's first operate on a temporary, then we "commit" at the end

```
template<class T>
void Stack<T>::Push(const T& t)
{
    if (vused_ == vsize_) {
        size_t vsize_new = vsize_ * 2 + 1;
        T* v_new = NewCopy(v_, vsize_, vsize_new);
        delete v_[];
        v_ = v_new;
        vsize_ = vsize_new;
    }
    v_[vused_] = t;
    ++vused_;
}
```


Pop is a problem

```
template<class T>
T Stack<T>::Pop()
{
    if (vused_ == 0) {
        throw "Pop from an empty stack";
    } else {
        T result = v_[vused_ - 1];
        -- vused_;
        return result;
    }
}
```

- This function looks correct, but unfortunately has an hidden problem

```
Stack<string> s;
...
string s2;
s2 = s.Pop();
```

  - If the last copy fails, we extracted an element, but this will never reach destination
  - we lost an element!

Two solutions

- We can change the `Pop()` function in two ways:
- first solution

```
template<class T>
void Stack<T>::Pop(T &result)
{
    if (vused_ == 0) {
        throw "Pop from an empty stack";
    } else {
        result = v_[vused_ - 1];
        -- vused_;
    }
}
```


STL solution

- Second solution is to add a `Top()` function, and let `Pop()` only remove the element, without returning it

```
template<class T>
void Stack<T>::Pop()
{
    if (vused_ == 0)
        throw "...";
    else -- vused_;
}
```

```
template<class T>
T Stack<T>::Top()
{
    if (vused_ == 0)
        throw "...";
    else return v_[vused_ - 1];
}
```

- This is the way chosen by the STL implementation of stack and other containers

What we require

- Let's see what we require to class T for the Stack to work properly
  1. A default constructor
  2. a copy constructor
  3. an assignment operator
  4. a destructor that does not throw
- we can do better than this, namely remove requirements 1 and 3

Remove the try/catch block

- We can remove the try/catch block in the NewCopy by using a different technique
  - This technique is quite general and it is based on the **pimpl pattern**, plus a two-phase structure (do the work, then commit at the end)
  - it can be reused in you code quite easily
- Let's start by moving all implementation in a separate class

```
template<class T>
class StackImpl {
public:
    StackImpl(size_t size=0);
    ~StackImpl();
    void Swap(StackImpl &other) throw();

    T *v_;
    size_t vsize_;
    size_t vused_;
private:
    StackImpl(const StackImpl &);
    StackImpl& operator=(const StackImpl &);
};
```

Constructor

```
template<class T>
StackImpl<T>::StackImpl(size_t size):
    v_(0), vsize_ (size), vused_(0)
{
    if (size > 0) v_ = operator new(sizeof(T)*size);
}
```

- The `operator new` only allocates memory, but it does not call the constructor of T
  - That's quite different from calling `new T[size_]`
  - therefore, no object of T is built
  - if `operator new` throws `bad_alloc`, the object is not built and we are safe

The placement operator new

- Before continuing we need to analyse again the process of dynamically creating an object
- a call `new T` can be divided into two parts:
  1. Memory allocation
  2. Construction of the object on the specific address
- Step (1) is performed by the `operator new` that we have just used
- Step (2) is performed by the *placement operator new*

```
T *p = operator new(sizeof(T));  // step 1
new (p) T();                     // step 2

// the two steps above are equivalent to
// p = new T;
```

std::construct() and std::destroy()

- The STL library provides two nice wrappers for using such operators;

```
template<class T1, class T2>
void construct(T1* p, const T2& value)
{
    new (p) T1(value);
}

template<class T>
void destroy(T* p)
{
    p->~T();
}

template<class FwdIter>
void destroy(FwdIter first, FwdIter last)
{
    while (first != last) {
        destroy(first);
        ++first;
    }
}
```


Destructor

```
template<class T>
StackImpl<T>::~StackImpl()
{
    destroy(v_, v_+vused_);
    operator delete(v_);
}
```

- `destroy()` calls all destructors for `vused_` objects
- `destroy()` cannot raise any exception
- `operator delete` is the dual of `operator new`: it just frees the memory (without calling any destructor)

The Swap function

- Now we can continue by looking at a very important function

```
template <class T>
void StackImpl<T>::Swap(StackImpl &other) throw()
{
    swap(v_, other.v_);
    swap(vused_, other.vused_);
    swap(vsize_, other.vsize_);
}
```

- Here we are only swapping pointers or size_t members, there is no function call, so no exception is possible
- this function swaps the two internal representations of Stack

Stack class

- Now we are ready to implement the Stack class

```
template <class T>
class Stack : private StackImpl<T>
{
public:
    Stack(size_t size = 0);
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    size_t Count() const;
    void Push(const T&);
    T& Top();
    void Pop();
};
```

Constructor and destructor

```
template<class T>
Stack<T>::Stack(size_t size) : StackImpl<T>(size)
{
}

template<class T>
Stack<T>::~Stack()
{}
```

- The destructor of StackImpl is automatically called, and Stack has nothing to destruct (we could also remove the definition, because the compiler provides a standard one for us)

Copy constructor

```
template<class T>
Stack<T>::Stack(const Stack<T>& other) :
    StackImpl<T>(other.vused_)
{
    while(v_used_ < other.vused_) {
        construct(v_ + vused_, other.v_[vused_]);
        ++vused_;
    }
}
```

- StackImpl constructor can raise an exception
  - Nothing bad can happen
- A copy constructor of T can raise an exception
  - In that case, the destructor of StackImpl will `destroy` exactly all objects that have been created (see ∼StackImpl())

Assignment operator

```
template<class T>
Stack<T>& Stack<T>::operator=(const Stack<T>& other)
{
    Stack<T> temp(other); // constructs a temporary copy
    Swap(temp);           // swaps internal implementations
    return *this;         // temp will be destroyed
}
```

- If the copy constructor fails, nothing bad happens
- the Swap cannot throw
- It follows that this is safe to exceptions

Push

```
template<class T>
void Stack<T>::Push(const T& elem)
{
    if (vused_ == vsize_) {
        Stack<T> temp(vsize_*2+1);
        while (temp.Count() < vused_)
            temp.Push(v_[temp.Count()]);
        temp.Push(elem);
        Swap(temp);
    } else {
        construct(v_ + vused_, elem);
        ++vused_;
    }
}
```

- Discuss why this is safe
- Push and Pop did not change

A general technique

- It turns out that this is a general technique
  - Put all implementation in a inner class
  - Your class will have a pointer to the implementation, or derive privately from the implementation
  - do all the work on a copy
  - when everything is safe, swap the pointers (cannot throw exceptions)
- Exercise: Write the Stack implementation using the pimpl idiom instead of the private inheritance