Exception Safety

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, October 28, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell
Unit Overview
Handling Data & Sequences

Major Topics
✓ ▪ Data abstraction
✓ ▪ Introduction to Sequences
  ▪ Smart arrays
    ✓ ▪ Array interface
    ✓ ▪ Basic array implementation
  (part) ▪ Exception safety
    ▪ Allocation & efficiency
    ▪ Generic containers
  ▪ Linked Lists
    ▪ Node-based structures
    ▪ More on Linked Lists
  ▪ Sequences in the C++ STL
 ▪ Stacks
 ▪ Queues
Our problem for much of the rest of the semester:
- Store: a collection of data items, all of the same type.
- Operations:
  - Access items [one item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- All this needs to be efficient in both time and space.

A solution to this problem is a **container**.

**Generic containers**: those in which client code can specify the type of data stored.
In **data abstraction**, we separate the various aspects of dealing with data, from the implementation of the data:

- The conceptual form of the data.
- The **operations** available on the data.
- The method used to access the data.

**Important concepts**

- **Abstract data type** (ADT).
- **Interface.**
Abstract data type (ADT)
- A collection of data, along with a set of operations on that data.
- Independent of implementation and programming language.
- Examples: Sequence, SortedSequence.

Data structure
- A construct within a programming language that stores a collection of data.
- Examples: Array, Linked List.

Class
- A language feature in C++ and some other languages, intended to facilitate OOP.
- In C++ we usually implement a data structure using a class. However, we are not required to.
- Examples: `std::vector<int>`, `std::list<double>`.
Review
Array Interface

Ctors & Dctor
- Default ctor
- Ctor given size
- Copy ctor
- Dctor

Member Operators
- Copy assignment
- Bracket

Global Operators
- None

Associated Global Functions
- None

Named Public Member Functions
- size
- empty
- begin
- end
- resize
- insert
- remove
- swap
Review
Basic Array Implementation

What data members should class SmArray have?
- Size of the array: `size_type size_;`
- Pointer to the array: `value_type * data_;`

What class invariants should it have?
- Member “`size_`” should be nonnegative.
- Member “`data_`” should point to an `int` array, allocated with `new []`, owned by `*this`, holding `size_` ints.

Note: This design has a serious (but not obvious) problem, as we will see.
An **error condition** (or “error”) is a condition occurring during runtime that cannot be handled by the normal flow of execution.

- Not necessarily a bug or a user mistake.
- Example: Could not read file.

Three ways to deal with a *possible* error condition in a function:

**Prevention**
Client code must prevent the error (precondition).

**Containment**
Fix the problem inside the function.

**Signal the Client Code**
Idea: When we cannot fulfill our postconditions.

Methods for signaling an error condition to the client code:

- Return an error code
- Set a flag, checked by a separate function
- Throw an **exception**
Exceptions are objects that are “thrown”, generally to signal error conditions.

We can catch all exceptions, using “...”.

- In this case, we do not get to look at the exception, since we do not know what type it is.

```java
try {
    p = new Foo[mySize];  // May throw
}
catch (...) {
    fixThingsUp();
    throw;
}
```

- Inside any catch block, we can re-throw the same exception using throw with no parameters.
The following can throw in C++:

- "throw" throws.
- "new" may throw std::bad_alloc if it cannot allocate.
- A function that (1) calls a function that throws, and (2) does not catch the exception, will throw.
- Functions written by others may throw. See their doc’s.

The following do not throw:

- Built-in operations on built-in types.
  - Including the built-in operator[].
- Deallocation done by the built-in version of "delete".
  - Note: "delete" also calls destructors. These can throw.
- C++ Standard I/O Libraries (default behavior)

If a destructor is called between a throw and a catch, and that destructor throws, then the program terminates.
- Therefore, destructors should not throw.
Issues: Does a function ever signal client code that an error has occurred, and if it does:
- Are the data left in a usable state?
- Do we know something about that state?
- Are resource leaks avoided?
These issues are collectively referred to as safety. We consider these in the context of exceptions: exception safety. However, most of the ideas we will discuss apply to any kind of error signaling technique.
There are a number of commonly used safety levels.
  - These are stated in the form of guarantees that a function makes.
In this class, we will adopt the convention that a function throws when it cannot satisfy its postconditions.
  - When a function exits without satisfying its postconditions, we will say it has failed.
Thus, a function we write must do one of two things:
  - Succeed (satisfy its postconditions), or
  - Fail, throw an exception, and satisfy its safety guarantee.
Exception Safety
The Three Standard Guarantees

**Basic Guarantee**
- Data remain in a usable state, and resources are never leaked, even in the presence of exceptions.

**Strong Guarantee**
- If the operation throws an exception, then it makes no changes that are visible to the client code.

**No-Throw Guarantee**
- The operation never throws an exception.

**Notes**
- First set out by Dave Abrahams in the mid-1990s.
  - Thus, they are sometimes called the “Abrahams Guarantees”.
- Written as part of the effort to standardize the C++ Standard Library.
  - However, they are applicable to more general contexts, not just C++.
- Each guarantee includes the previous one.
- The Basic Guarantee is the minimum standard for all code.
- The Strong Guarantee is the one we generally prefer.
- The No-Throw Guarantee is required in some special situations.
Exception Safety
The Three Standard Guarantees — Basic Guarantee

Data remain in a usable state, and resources are never leaked, even in the presence of exceptions.

- When a member function throws, an object may end up in an unknown state, but it must be a valid state, with invariants maintained.

This is minimum standard that we expect well-written code to meet.

- What happens if this standard is not met, and an exception is thrown?
Exception Safety
The Three Standard Guarantees — Strong Guarantee

If the operation throws an exception, then it makes no changes that are visible to the client code.

- Changes can be made, but the client must not see them.
- In practice, we exempt things like logging from these requirements.
- Generally, any work that has been done, must be undone.
- Thus, this is also called commit or roll-back semantics.

We like this level of safety, and we write code that meets it whenever it is reasonable to do so.

- But sometimes it is not reasonable, often due to efficiency concerns.
Exception Safety
The Three Standard Guarantees — No-Throw Guarantee

**The operation never throws an exception.**

- This is also know as the “No-Fail Guarantee”.

This is the **highest** level of exception safety, but it is not necessarily the **best** level.

- Exceptions are not “bad”. They are a tool that can help us deal with problematic situations. If we make the No-Throw Guarantee, then we have prohibited ourselves from using this tool.

- The No-Throw Guarantee does not say “errors do not occur”; rather, it says that if an error condition occurs, then we are not allowed to signal the client; we must fix it ourselves.

Sometimes it is important to make the No-Throw Guarantee, often in situations in which we are “finishing something”.

- One such situation: destructors.

- Another situation (“commit functions”) will be covered shortly.
Exception Safety
Writing Exception-Safe Code — Ideas

To make sure code is exception-safe:

- Look at **every** place an exception might be thrown.
- For each, make sure that, if an exception is thrown, either
  - we terminate normally and meet our postconditions, or
  - we throw and meet our guarantees.

That can be a lot of work. However, **modularity** helps.

- Once we can certify a function as exception-safe, we can use it as such without re-examining it.

A bad design can force us to be unsafe.

- Thus, good design is part of of exception safety.
- In particular, an often helpful idea is that every module has exactly one well defined responsibility (the **Single Responsibility Principle**). Code that follows this principle is **cohesive**.
  - Suppose that a function has two things to do, and the second thing produces an error.
  - Suppose that the second thing, above, is when the function returns a value.
  - Thus, the rule: A non-const member function should not return an object by value.
Exception Safety
Writing Exception-Safe Code — Write It

TO DO

- Figure out and comment the exception-safety guarantees made by all functions implemented so far in class SmArray.

  Done. See the latest versions of smarray.h, smarray.cpp, on the web page.

- Can/should any of these be improved?
  - No. All the constructors offer the Strong Guarantee, which cannot be improved, since they do dynamic allocation, and so might fail. All other functions written so far offer the No-Throw Guarantee.
Exception Safety
Commit Functions — The Need

Often it is tricky to offer the Strong Guarantee when modifying multiple parts of a large object.

- If we make several changes, and then we get an error, it can be difficult to undo the changes.
- In fact, it may be impossible, if the undo operation itself may result in an error.

Solution

- Create an entirely new object with the new value.
- If there is an error, destroy the new object. The old object has not changed, so there is nothing to roll back.
- If there is no error, **commit** to our changes using a non-throwing operation.

For many purposes, a good commit function is a non-throwing **swap** function.
Swap member functions usually look like this:

```c++
void MyClass::swap(MyClass & other);
```

This should exchange the values of *this and other.

Swap functions can *usually* be written very easily.
- Just swap the data members.
- Ownership issues are easy to handle properly (right?).

In fact, it is usually easy to write a swap function that is:
- Non-throwing.
- Very fast.
class MyClass {
private:
    int x;
    double y;
public:
    void swap(MyClass & other);  // Does not throw

We can implement MyClass::swap like this:

void MyClass::swap(MyClass & other)  // Does not throw
{
    int tempi = x;
    x = other.x;
    other.x = tempi;

    double tempd = y;
    y = other.y;
    other.y = tempd;
}
Alternatively, we can use `std::swap`, in `<algorithm>`:

```cpp
void MyClass::swap(MyClass & other) {
    std::swap(x, other.x);
    std::swap(y, other.y);
}
```

If we need to swap members that are `objects`, we might want to avoid `std::swap`.
- Algorithm `std::swap` uses the copy ctor and copy assignment. These might throw.
- If the objects have their own non-throwing swap member function, we can use that:

```cpp
void MyClass::swap(MyClass & other) {
    ...  
    z.swap(other.z);  // z is a member
                    // of class type
}
```

- Many C++ Standard Library classes have such a member function.

Note: if `Foo` is a class, then `Foo` is not a built-in type; it is an object. However, `(Foo *)` is a pointer, which is a built-in type.
We can use a non-throwing swap function to get the Strong Guarantee.

To give our object a new value:
- First, **try to construct** a temporary object holding this new value.
- If this fails, exit. No change.
  - Exiting is automatic, if the failing operation throws.
- If the construction succeeds, then **swap** our object with the temporary object holding the new value.
- Exit. The destructor of the temporary object cleans up the old value of our object.
  - Destruction is automatic.
  - And it should never fail.

Note: boldface = code we write.
Thus, we can set an object to a new value, while offering the Strong Guarantee, as long as we have a way to construct the new value that offers the Strong Guarantee, along with a destructor and a swap function that offer the No-Throw Guarantee.

Procedure

- **Try to construct** a temporary object holding the new value.
- **Swap** with this temporary object.

Example: “clear” by swapping with a default-constructed temporary object.

```cpp
void MyClass::clear()  // Strong Guarantee
{
    MyClass temp;         // If there is a problem creating temp, then an exception is
    swap(temp);           // thrown, and “nothing” happens (Strong Guarantee).
}
```

The old value of *this* is cleaned up by temp’s destructor.
Exception Safety
Commit Functions — Usage [3/3]

This idea lets us write a copy assignment operator that makes the Strong Guarantee. We need:

- A copy ctor that offers the Strong Guarantee (this is usually not too difficult).
- A swap member function that makes the No-Throw Guarantee (usually easy).
- A dctor that makes the No-Throw Guarantee (of course).

Code:

```cpp
MyClass & MyClass::operator=(const MyClass & rhs)  // Strong Guarantee
{
    if (this != &rhs) {
        MyClass temp(rhs);  // Try to construct a temporary copy of rhs.
        swap(temp);          // Swap with the temporary copy.
    }
    return *this;          // Always end an assignment operator this way.
}
```

Admittedly this is a bit mind-twisting. However, assuming the requirements are met, it is easy to write, and it always works.