Error Handling
Using Exceptions

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, September 11, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Parameter passing II
- ✓ Invisible functions I
- ✓ Integer types
- ✓ Managing resources in a class
- ✓ Containers & iterators
- ✓ Invisible functions II
- Error handling
- Using exceptions
- A little about Linked Lists

Major Topics: S.E. Concepts

- ✓ Invariants
- ✓ Testing
- Abstraction

Review

Review
Containers & Iterators [1/3]

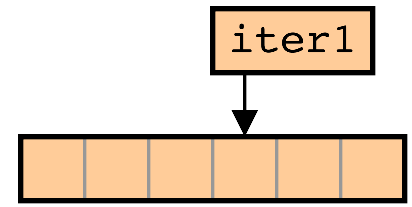
A **container** is a data structure that can hold multiple items, usually all of the same type.

A **generic container** is a container that can hold items of a client-specified type. One kind is a C++ built-in array. Others are in the C++ **Standard Template Library** (**STL**): `std::vector`, `std::list`, `std::map`, etc.

Review
Containers & Iterators [2/3]

An iterator refers to an item in a container—or acts like it does.

An iterator does not own the item it refers to.



I practice, I would use `auto` here.

```
vector<int>::iterator iter1 = begin(vv)+3;
vector<int>::const_iterator citer;
```

Cannot be used to modify the item it refers to.

An iterator may be a **wrapper**, to make data look like a container.

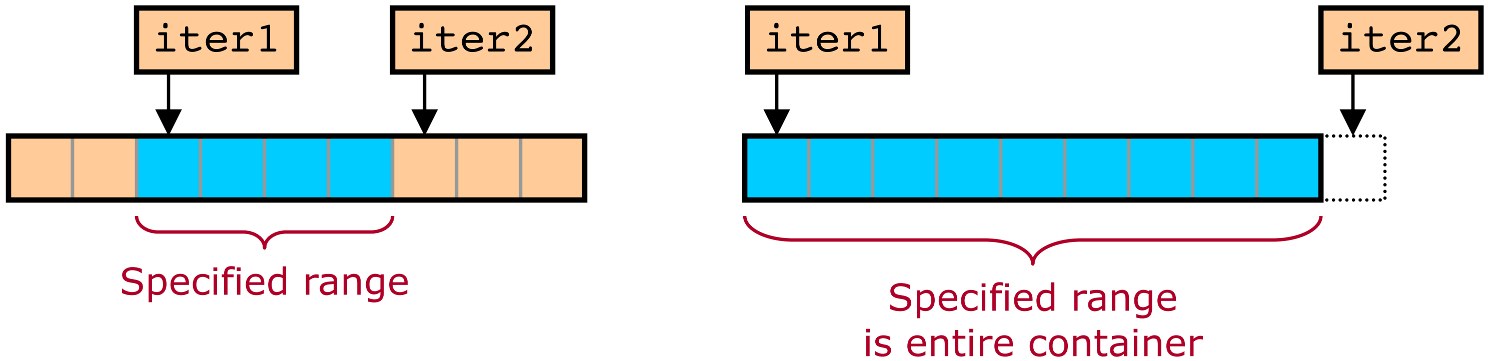
```
#include <iterator>
std::ostream_iterator<int> coolIter(std::cout, "\n");

*coolIter++ = 3;          // Same effect as next line
std::cout << 3 << "\n";
```

Review
Containers & Iterators [3/3]

To specify a **range**, we use two iterators:

- An iterator to the first item in the range.
- An iterator to just past the last item in the range.



STL **generic algorithms** follow this convention.

```
copy(begin(v), end(v), begin(v2));
bool isEq = equal(begin(v), end(v), begin(v2), end(v2));
sort(begin(v), end(v));
fill(begin(v), end(v), the_value);
```

Each underlined pair of arguments forms a range specification.

See `iterators.cpp`.

Review
Invisible Functions II

The **Rule of Five**:

If you define one of the **Big Five**, then define or =delete all of them.

The usual reason is that an object directly manages a resource.

```
~Dog();                                // Dctor
Dog(const Dog & other);                // Copy ctor
Dog & operator=(const Dog & rhs);      // Copy assignment op
Dog(Dog && other);                     // Move ctor
Dog & operator=(Dog && rhs);           // Move assignment op
```

We prefer to write none of them.

The **Rule of Zero**:

Where possible, do not explicitly define any of the Big Five. Resources should be managed by data members that are objects of RAII classes.

But if we write one of those RAII classes—see the slides from last time.

Error Handling

Error Handling
Error Conditions [1/2]

An **error condition** (often *error*) is a condition occurring during runtime that cannot be handled by the normal flow of execution.

An error condition is not the same as a bug in the code.

- We are not referring to compilation errors.
- *Some* error conditions are caused by bugs, but our discussion of error handling will focus on properly written code.

An error condition does not mean the user did something wrong.

- *Some* error conditions are caused by user mistakes.

Example

- A function `copyFile` opens a file, reads its contents, and writes them to another file.
- `copyFile` is called to read a file that is accessed via a network.
- Halfway through reading the file, the network goes down.
- It is now impossible to read the file. The normal flow of execution cannot handle this situation. We have an error condition.

Error Handling
Error Conditions [2/2]

How do we deal with possible error conditions?

Sometimes we can **prevent error conditions**:

- Write a precondition that requires the caller to keep a certain problem from happening.
- Example. Insisting on a non-zero parameter, to prevent a division-by-zero error condition.

Handle a possible error condition **before** the function.

Sometimes we can **contain error conditions**, by handling them ourselves:

- If something is not right, then deal with it.
- Example. A fast algorithm needs more memory than we have; we use a slow method instead.

Handle a possible error condition **in** the function.

But sometimes neither of these two is feasible.

Then we must **signal the client code**.

- Rule of Thumb. Signal the client code when the function is unable to fulfill its postconditions.
- Example. The earlier file-reading troubles.

Handle a possible error condition **after** the function.

Error Handling
Preview of Goals and Guarantees

When client code might need to be informed of an error condition, we may have these three goals:

- Error conditions must not wreck our program. It must continue running, and later end properly. Objects must be usable. Resources must not leak.
- Even better, it would be good if each operation we attempt either completes successfully, or, if there is an error condition, has no effect.
- Best of all, it would be great we never need to inform client code of errors at all.

Later in the class, we will formalize these as **safety guarantees**.

→ The first goal is the fundamental standard that all code must meet. We call it the **Basic Guarantee**.

→ The second is preferred, although sometimes not feasible. We call it the **Strong Guarantee**.

→ The third is mostly wishful thinking. Sometimes we simply *must* inform client code of an error condition. But in special cases—often involving *finishing* something—we do require this standard. We call it the **No-Throw Guarantee** (or the **No-Fail Guarantee**).

Error Handling
Signaling the Client Code [1/2]

When we cannot prevent or contain an error condition, then we must signal the client code. How can we do this?

Method 1. Return an error code.

```
int c = getc(myFile);
if (c == EOF)
    printf("End of file\n");
```

The old C-language I/O library uses this method.

Method 2. Set a flag to be checked by a separate error-checking function.

```
char c;
myFileStream >> c;
if (myFileStream.eof())
    cout << "End of file" << endl;
```

C++ file streams *default* to using this method.

Error Handling
Signaling the Client Code [2/2]

Return codes and separate error-checking functions are acceptable methods for flagging error conditions, but they have downsides.

- They can be difficult to use in places where a value cannot be returned, or an error condition cannot be checked for.
  - Constructors, in the middle of an expression, etc.
  - When you call someone else’s function, and that calls your function, which needs to signal an error condition.
- They can lead to complicated code.
  - A function calls a function that calls a function that calls a function—and an error occurs. To handle the error, we must back out of all of these.

Because of these issues, a third method was developed.

Method 3. Throw an exception.

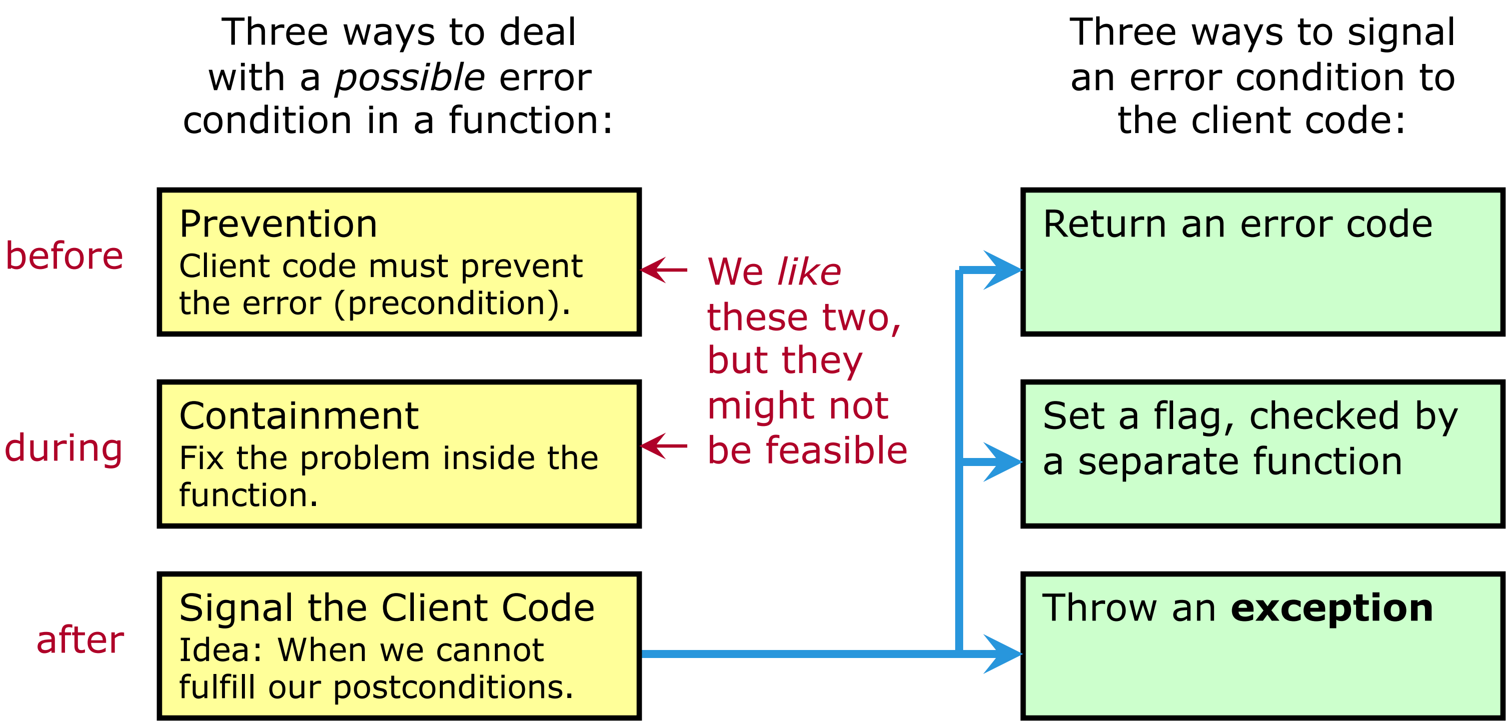
Exceptions are available in many programming languages (C++, Java, Python, JavaScript, etc.), and are associated with OOP.

In our next topic, we look at exceptions in C++.

Error Handling
Summary

An **error condition** (often *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.



Using Exceptions

Exceptions & Catching — The Idea

Exception: an object that is **thrown** to signal an error condition.

- `new` throws `std::bad_alloc` or a derived class, if allocation fails.

To handle an exception, **catch** it using `try` … `catch`.

```
#include <new>  // for std::bad_alloc

Foo * p;
bool success = true;
try {
    p = new Foo;
}
catch (std::bad_alloc & e) {
    success = false;
    cerr << "Allocation failed: " << e.what() << endl;
}
```

`catch` gets an exception that is thrown inside the corresponding try-block, if it has the proper type.

`e` is the **exception**.

Catch exceptions by reference.

Standard exception types have a member function `what`. It returns `string`.

Exceptions & Catching — What is Caught? [1/4]

Under what circumstances is a thrown exception caught?

If it is caught, then *where* in the code is it caught?

How It Works

- When an exception is thrown inside a try-block, control passes to the catch-block that (1) is associated with the smallest possible enclosing try-block, and (2) catches the proper type. Derived classes are handled as usual.
- In all other circumstances, a catch-block is not executed.

That’s it! Exception handling is not complicated—even if some of the examples we cover make it seem complicated.

Using Exceptions
Exceptions & Catching — What is Caught? [2/4]

A `catch` only gets an exception that is:

- Thrown inside the corresponding try-block.
- Of an appropriate type.

Once an exception is thrown, the try-block is exited.

If no exception is thrown, the catch-block is not executed.

```
Foo * p1, p2;
p1 = new Foo;
try {
    p2 = new Foo;
    myFunc(p2);
}
catch (std::bad_alloc & e) {
    [exception-handling code goes here]
}
```

The catch-block below will not catch any exception thrown by this statement. (→ `p1 = new Foo;`)

If the `new` throws, then this function call is not made.

If this function throws an exception that is not `std::bad_alloc` or a derived class, then the catch-block below is not executed. (→ `myFunc(p2);`)

Exceptions & Catching — What is Caught? [3/4]

`catch` gets exceptions of the proper type that are thrown inside the corresponding `try`.

This includes an exception thrown in a called function, if it is not caught inside that function—that is, if it **escapes** the function.

```
void myFunc()
{
    globalP1 = new Foo;
    globalFlag = true;
    try {
        globalP2 = new Foo;
    }
    catch (std::bad_alloc & e) {
        globalFlag = false;
    }
```

Function `main` would be able to catch an exception thrown by this statement … [arrow to `globalP1 = new Foo;`]

… but not a `std::bad_alloc` thrown by this statement. [arrow to `globalP2 = new Foo;`]

Exceptions & Catching — What is Caught? [4/4]

Exceptions can propagate out of nested function calls.

Catching by reference will catch exceptions of derived types.

```
void xx();  // May throw std::bad_alloc

void yy()
{ xx(); }

void zz()
{
    try {
        yy();
    }
    catch (std::exception & e) {
        …
```

When function `zz` is called, if **function `xx`** throws `std::bad_alloc`, then the exception will be caught **here**.

Because we catch by reference, derived classes of `std::exception` will be caught.

All standard exception classes, including `std::bad_alloc`, are derived from `std::exception`.

An uncaught exception terminates the program.

```
void myFunc2();  // May throw std::out_of_range

int main()
{
    Foo * p1 = new Foo;
    try {
        myFunc2();
    }
    catch (std::bad_alloc & e) {
        …
    }
    …
```

An exception **here** or **here** will not be caught, and so will terminate the program.

Using Exceptions
Throwing [1/3]

We can throw our own exceptions, using `throw`.

We do not do this very much!

```
class CC {
public:
    int & operator[](std::size_t ix)
        // May throw std::out_of_range
    {
        if (ix >= _arrsize)
            throw std::out_of_range("CC::op[]: bad ix");
        return _arr[ix];
    }
private:
    int *       _arr;
    std::size_t _arrsize;
```

The syntax of `throw` is the same as the syntax of `return`.

Using Exceptions
Throwing [2/3]

When throwing your own exception—*which you will not do very much!*—use or derive from one of the standard exception types.

Standard exception types have a string member, for a human-readable message. This is a ctor parameter. Access it via the `what()` member function.

To make your own exception type, derive from a standard exception type. Standard exception types are set up to allow this. (In particular, they all have virtual destructors.)

Using Exceptions
Throwing [3/3]

The following can throw:

- `throw` throws.
- `new` may throw `std::bad_alloc` or a derived class (default behavior).
- 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 documentation.

The following do *not* throw:

- Built-in operations, other than `new`, on built-in types.
  - Including `operator[]`.
- Deallocation done by the built-in version of `delete`.
  - Note: `delete` calls destructors, which conceivably *might* throw—but *should* not, as we will see.
- C++ Standard I/O Libraries (default behavior).

Using Exceptions
Catch All & Re-Throw

Use `catch(...)` to catch *all* exceptions.

Inside a catch-block, "`throw;`" will re-throw the same exception.

These two are used together, to ensure that clean-up gets done.

```
try {
    myFunc3();
}
catch (...) {
    doNecessaryCleanUp();
    throw;
}
```

This is *not* my indication that something is missing. The code actually contains three dots.

Catch all & re-throw is used in C++ similarly to the way "`finally`" is used in some other programming languages (e.g., Java, Python).

Now we know two ways to ensure that clean-up is done before we leave: (1) RAII, (2) catch all & re-throw.

Exceptions, Dctors, noexcept [1/2]

Fact 1. An automatic object's dctor is called when it goes out of scope, even if this is due to an exception.

Fact 2. If an exception is thrown, and one of the destructors called before it is caught also throws, then the program terminates.

Dctors are only called for **fully constructed** objects. If a ctor throws, then the dctor for that object will not be called.

Put these two facts together, and we conclude:

Destructors should not throw.

It is okay for constructors to throw.

The above is a technical argument based on the specification of C++. From a more philosophical point of view, *finishing-up* operations—like destructors—generally should not throw.

Exceptions, Dctors, noexcept [2/2]

Because dctors should not throw, they are generally marked noexcept implicitly, unless otherwise specified.

If a noexcept function throws, then the program terminates.

> Recall: `noexcept` is a promise that a function will not throw.

We *can* make a destructor that is not noexcept using "`noexcept(false)`". But this is *EVIL*. ☹

EVIL! (annotation pointing to `~Foo() noexcept(false)`)

```
class Foo {
public:
    ~Foo() noexcept(false)
    {
        …
    }
    …
```

Using Exceptions
CODE [1/2]

TO DO

- Examine and execute some code that uses exceptions.

See `except.cpp`*.*

Using Exceptions
CODE [2/2]

TO DO

- Write a function `allocate1` that:
  - Attempts to allocate a dynamic object.
  - Returns a pointer to this object, using a reference parameter.
  - If the allocation fails, throws `std::bad_alloc`.
  - Has no memory leaks.
- Write a function `allocate2` that:
  - Attempts to allocate *two* dynamic objects.
  - Returns pointers to these objects, using reference parameters.
  - If either allocation fails, throws `std::bad_alloc`.
  - Has no memory leaks.
- Look at example code showing how RAII can simplify these situations.

Done. See `allocate2.cpp`*.*

See `allocate2_raii.cpp`*.*

Using Exceptions
Final Thoughts [1/3]

When to Do Things

- **Catch** when you can handle an error condition that may be signaled by some function you call.
- **Throw** when a function you are writing is unable to fulfill its postconditions and must signal an error condition.
- **Catch all & re-throw** when you call a function that may throw, you cannot handle the error, but you do need to do some clean-up before your function exits.

Typically we do not write more than one of the above three.

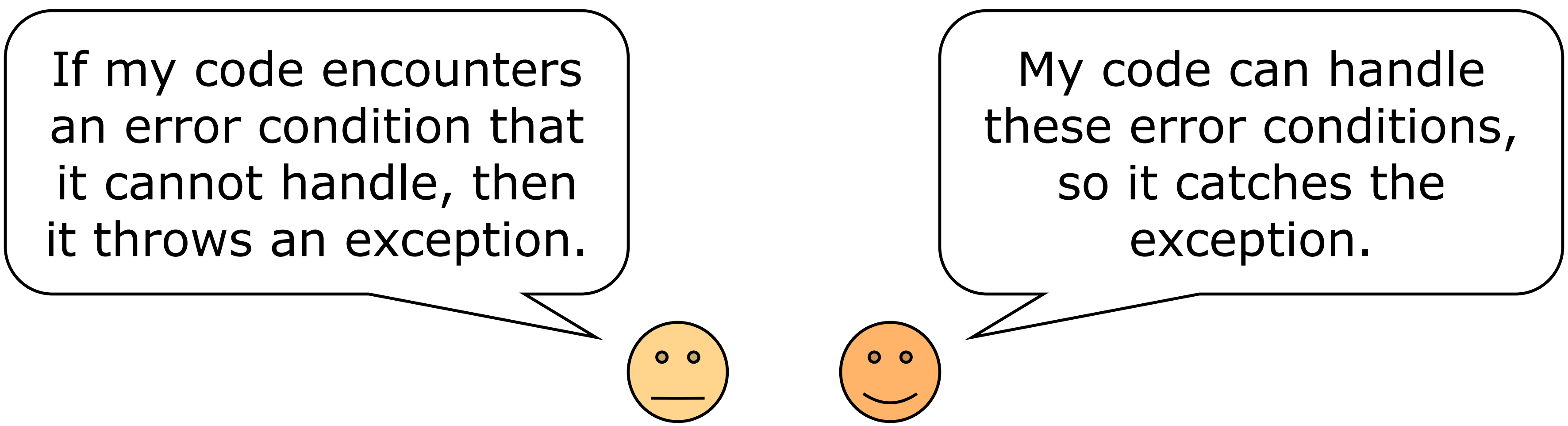
- Writing more than one of the above three is a **code smell**: an indication that something may not be right.

Great term!

Using Exceptions
Final Thoughts [2/3]

Again, we typically write only one of the three (catch, throw, catch all & re-throw).

That means that if someone writes a throw, then the associated catch will probably be written by someone else.



Using Exceptions
Final Thoughts [3/3]

Some people do not like exceptions. Some of these people are very vocal about their dislike. But I think that *some* of them dislike exceptions for the wrong reasons.

A bad reason to dislike exceptions is that they require lots of work.

- Dealing with error conditions is work. Writing software that *works* is work. Exceptions are one tool we can use to achieve this goal.
- Handling exceptions properly is hard work because *writing correct, robust software is hard work*.

Software is **robust** if it can gracefully handle anything tossed at it.

What *might* be a good reason to dislike exceptions is that they add hidden execution paths.

But remember that other error-handling methods have their own downsides—which is why exceptions were invented.