# Thoughts on Assignment 2
# Error Handling
# Using Exceptions

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, September 13, 2024

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

## Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Example class
- ✓ Parameter passing II
- ✓ Invisible functions I
- ✓ Managing resources in a class
- ✓ Containers & iterators
- ✓ Invisible functions II
- Error handling
- Using exceptions

## Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Assertions
- ✓ Testing
- ✓ Invariants

# Review

# Review
# Software Engineering Concepts: Invariants

An **invariant** is a condition that is always true at a particular point in a computation. Example: the condition used in an assertion.

Three Special Kinds

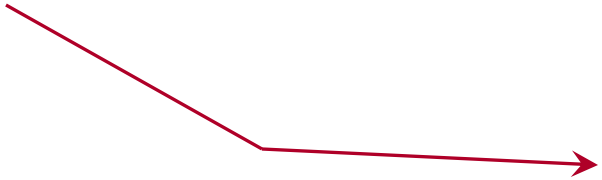
- **Precondition**. Invariant at the beginning of a function. The responsibility for ensuring the preconditions hold lies with the caller.
  - What must be true for the function to execute properly.
- **Postcondition**. Invariant at the end of a function. Tells what services the function has performed.
  - Describe the function's effect using statements about values.
- **Class invariant**. Invariant that holds whenever an object of the class exists, and execution is not inside a member function.
  - Statements about data members that indicate what it means for an object to be valid or usable.
  - These are preconditions for all member functions except ctors, and postconditions for all member functions except the dctor.

Pre- and postconditions are the basis for **operation contracts**.

# Software Engineering Concepts: Invariants — TRY IT (Exercise)

**Exercise**

- Write pre- and postconditions for the one-parameter constructor for class `Abc`.



*See next slide for answers.*

```
// class Abc
// Invariants:
//      0 <= _n && _n < 100
class Abc {
public:
    Abc(int nn)
        :_n(nn)
    {}
    [other stuff here]
private:
    int _n;
};  // End class Abc
```

# Software Engineering Concepts: Invariants — TRY IT (Answers)

## Exercise

- Write pre- and postconditions for the one-parameter constructor for class `Abc`.

## Answers

```
// Pre:
//      0 <= nn && nn < 100
// Post:
//      _n == nn
```

`0 <= _n && _n < 100` is also a postcondition. But that is already in the class invariants, and we do not need to repeat it.

```
// class Abc
// Invariants:
//      0 <= _n && _n < 100
class Abc {
public:
    Abc(int nn)
        :_n(nn)
    {}
    [other stuff here]
private:
    int _n;
};  // End class Abc
```

# Thoughts on Assignment 2

# Thoughts on Assignment 2
# Overview of Ideas

In Assignment 2 you write a "moderately smart" array (`MSArray`). This will require applying some recently covered ideas.

- Managing Resources in a Class
  - Are you doing dynamic allocation correctly? When you allocate something, is it always freed?
  - `MSArray` uses RAII.
  - What type to use for the size of an `MSArray`? For array indices?
- Containers & Iterators
  - `MSArray` is a generic container. Its member functions `begin` and `end` return iterators.
- Software Engineering Concepts: Invariants
  - You will need to document preconditions for each function that has them and class invariants for each class.
- Invisible Functions II
  - `MSArray` directly manages a resource. You will need to define all of the Big Five. (Do not apply the Rule of Zero in Assignment 2!)

# Thoughts on Assignment 2
# Templates [1/2]

`MSArray` is a **class template**, like `std::vector`.

```
MSArray<int> ia;
```

Define it like this:

```
template <typename ValType>
class MSArray {
public:
    using value_type = ValType;

    …
};
```

**Template parameter**. Name this whatever you want.

A template parameter can be used inside the template definition (and only there!).

If an `MSArray` variable is declared as above, then `ValType` means `int` here.

Normal class definition, with member functions, etc.

Templates go entirely in the header. Do not write a separate source file for `MSArray`.

# Thoughts on Assignment 2
# Templates [2/2]

Each associated global function is actually a **function template**.
Define one of these (in the header file!) like this:

```
template <typename ValType>
bool operator==(const MSArray<ValType> & a,
                const MSArray<ValType> & b)
{
    return …
}
```

Inside the class template definition (see the previous slide) and definitions of member functions, write `MSArray`. In other places, like **here**, write `MSArray<`*whatever*`>`.

# Thoughts on Assignment 2
## Documentation

Document **class invariants** for all classes and **preconditions** for all functions that have them.

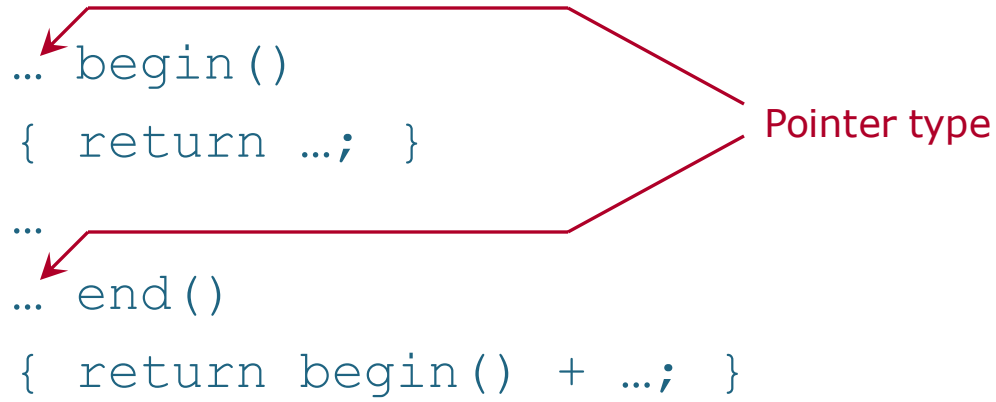
```
// Invariants: …
template <typename ValType>
class MSArray {
    …
};


// Pre: …
template <typename ValType>
bool operator==( … )
```

# Thoughts on Assignment 2
# Iterators

Member functions `begin` and `end` return iterators.

- *These can be pointers.* Do not write a separate iterator class.
- Function `begin` returns an iterator to the first array item. You already have a pointer to the first array item (*think …*); use it.
- Function `end` returns an iterator to just past the last array item. Add a number to the return value of `begin` (*what number?*).



# Thoughts on Assignment 2
# Access to Internal Data

A `const MSArray` has non-modifiable data. If a function can give access to data *in modifiable form*, then write two versions.

```
… operator[]( … )
{ … }
const … operator[]( … ) const
{ … }
```

```
… begin()
{ … }
const … begin() const
{ … }
```

```
… end()
{ … }
```

In each pair, the two functions should be identical, except for (1) the `const` at the end of the first line, and (2) what is returned—or how it is returned.

In this particular case, we will allow repetition of code.

# Thoughts on Assignment 2
# Writing the Big Five

You will need to write the Big Five in `MSArray`.

Items in C++ built-in arrays are always default-constructed. We cannot set their values to anything else in a member initializer. Therefore, the copy ctor will need a loop* in the function body.

```
// Copy ctor
MSArray(const MSArray & other)
    :_arrayptr(new … ),
     …
{ … }
…
value_type * _arrayptr;
…
```

Initialize array items with a loop* *here*.

*Or perhaps one of the generic algorithms from the STL? (Hint, hint.)

For the rest, see *Invisible Functions II*, and do what it says!

# 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**.

- 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.
- But it would be really great if 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 often 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** (some call it 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.
- They can lead to complicated code.

Because of these issues, another method was developed.

Method 3. Throw an exception.

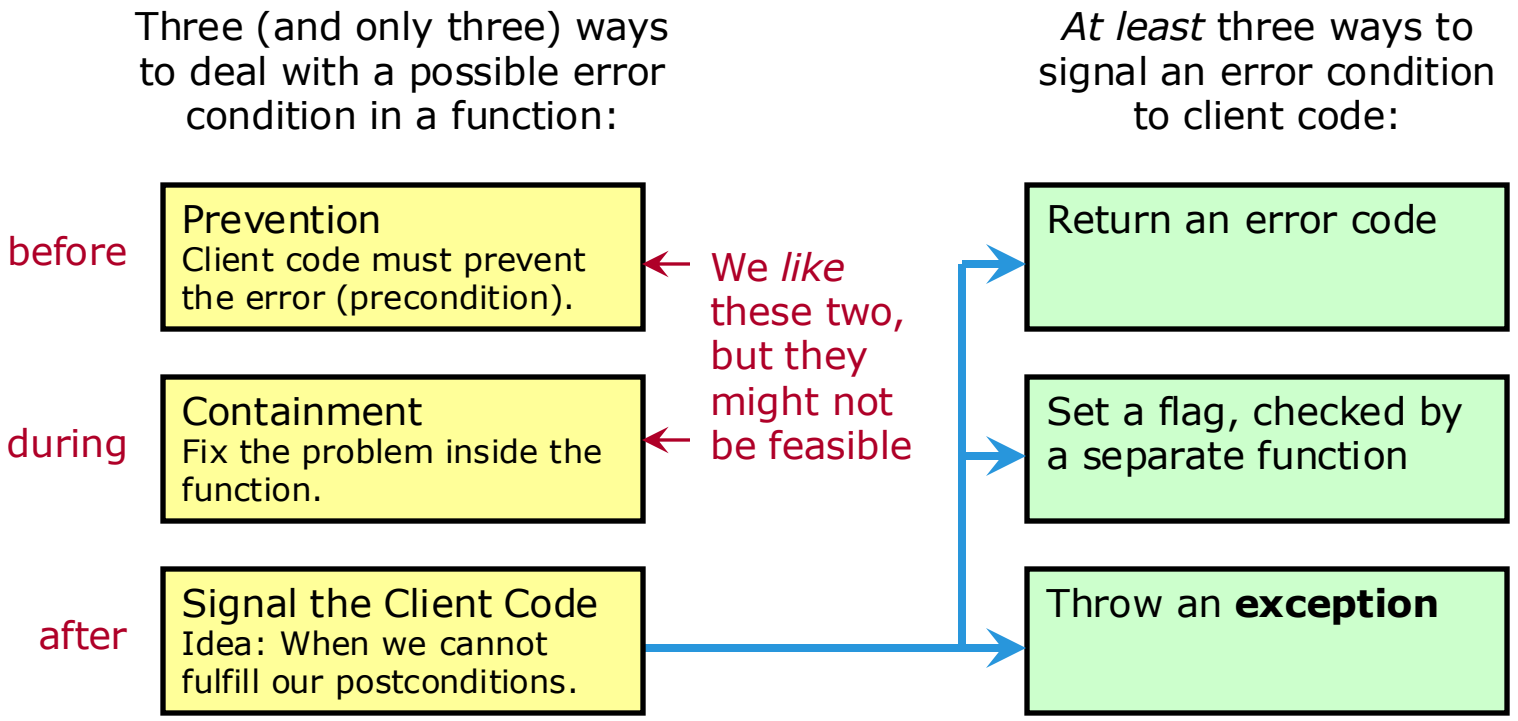
Exceptions are available in many programming languages: C++, Java, Python, JavaScript, etc. They are associated with OOP.

*In our next topic, we will 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 immediately passes to the catch-block that is associated with the smallest enclosing try-block that catches the proper type. Derived classes are handled as usual.
- In all other circumstances, a catch-block is not executed.
- An uncaught exception terminates the program.

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

# 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.

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.

## 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 …

… but not a `std::bad_alloc` thrown by this statement.

## 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`.

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).

The following can throw:

- `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.
- Lastly, `throw` always throws. *See the next subtopic.*

# Using Exceptions
# Throwing [1/2]

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 just like the syntax of `return`.

# Using Exceptions
# Throwing [2/2]

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 are set up to allow for derived classes; in particular, they all have virtual destructors.

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

# 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.

# Using Exceptions
## 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)`". However, this is *EVIL*. ☹

*EVIL!*

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

# Using Exceptions
# CODE

TO DO

- 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`.

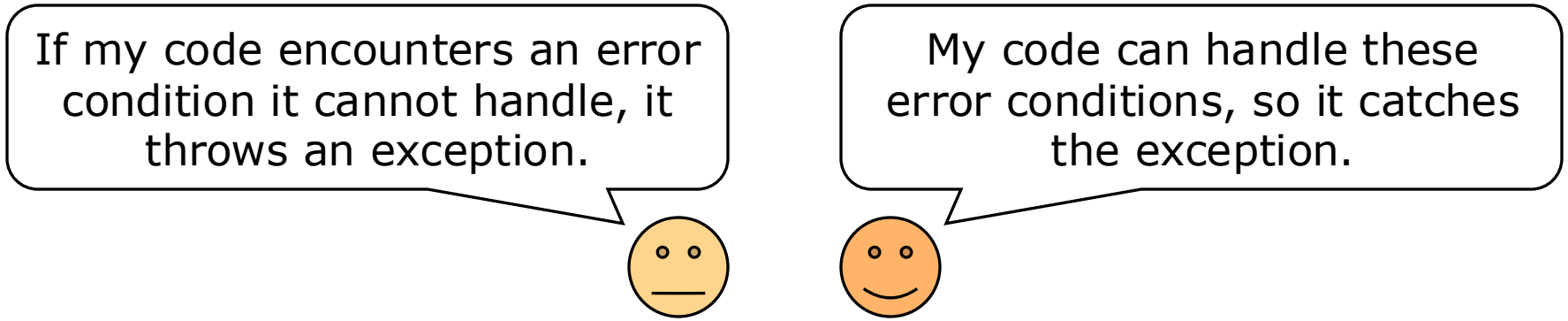
*See* `allocate2_raii.cpp`.

# Using Exceptions
# Final Thoughts [1/2]

When to Do Things

- **Catch** when you can handle an error condition that may be signaled by some function you call.
- **Throw** when your function 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 these.** If someone writes a throw, the catch is typically written by someone else.



# Using Exceptions
# Final Thoughts [2/2]

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 software is hard work*.

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.