Containers & Iterators continued
Software Engineering Concepts: Invariants
Invisible Functions II

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, September 11, 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
- (part) Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions

Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Assertions
- ✓ Testing
- Invariants

Review

Review
Managing Resources in a Class

Some **resources** need clean-up when we are done with them.

- Examples: dynamic objects or arrays, files to be closed, etc.
- We **acquire** a resource. Later, we **release** it.
- If we never release: there is a **resource leak**.

Own a resource = be responsible for releasing.

Ownership =
Responsibility
for Releasing

Prevent resource leaks with **RAII**.

- A resource is owned by an object.
- Therefore, its destructor releases—if this has not been done yet.
- Define, =delete, or =default each of the Big Five in an RAII class.

RAII =
An Object Owns
and, therefore, its
destructor releases

Review
Containers & Iterators [1/3]

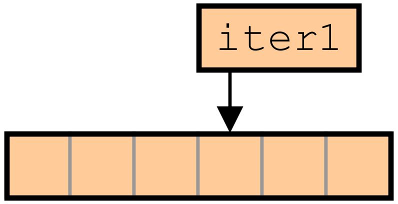
A **container** is a data structure that can hold multiple items, usually all of the same type.

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.

All the STL containers have interfaces that involve *iterators*.

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.

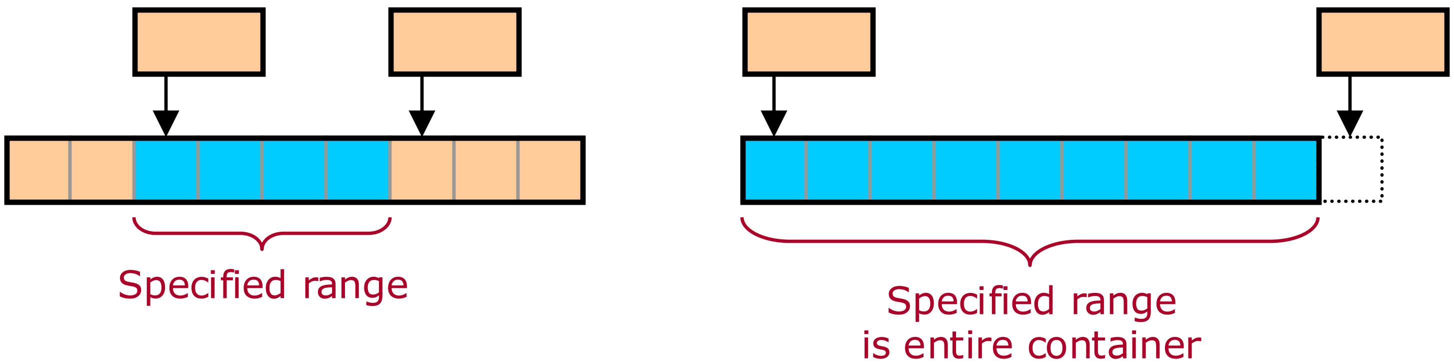
```
#include <iterator>
std::ostream_iterator<int> coolIter(std::cout, "\n");

*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.




```
sort(begin(v)+2, begin(v)+6);  // Sort v[2]..v[5]
sort(begin(v), end(v));        // Sort all of v
```

Containers & Iterators

continued

Containers & Iterators
Generic Algorithms [1/4]

The STL includes a number of **generic algorithms**, which can operate on arbitrary datasets. Most of these make use of iterators. All are defined in the header `<algorithm>`.

For example, algorithm `std::copy` copies the values in a range to another range.

```
#include <algorithm>
using std::copy;

vector<int> v(20);
vector<int> v2(20);
copy(begin(v), end(v), begin(v2));  // Copy v to v2.
copy(begin(v), end(v), coolIter);
    // Print the items in v, one on each line!
```

Containers & Iterators
Generic Algorithms [2/4]

Most of the STL generic algorithms take ranges. A range is specified using 2 iterators, in the way we have discussed.

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

`std::copy` has three parameters: 2 iterators specifying the range to read from, and an iterator to the first item in the range to write to.

```
copy(begin(v), end(v), begin(v2));
```

Range to read from

Start of range to write to

The second range must be large enough to hold all the items from the first range.

Containers & Iterators
Generic Algorithms [3/4]

In addition to `std::copy`, be familiar with these STL algorithms:

- `std::equal`: check if two ranges have the same size and hold the same values.

```
bool isEq = equal(begin(v), end(v), begin(v2), end(v2));
    // Another version takes 3 params, like std::copy;
    //  that one assumes the ranges are the same size
```

- `std::sort`: reorder the values in a range in ascending order.

```
sort(begin(v), end(v));  // Rearrange items in v
```

- `std::fill`: set all items in a range to a given value.

```
fill(begin(v), end(v), 6);  // Set every item in v to 6
```

Containers & Iterators
Generic Algorithms [4/4]

TO DO

- Run some code using iterators and STL generic algorithms.

See `iterators.cpp`*.*

Software Engineering Concepts: Invariants

Software Engineering Concepts: Invariants
Basics [1/2]

An **invariant** is a condition that is always true at a particular point in a computation. Typically, it says something about the values of variables.

```
if (ix < 0)
{
    flagError("Index too small");
    return;
}
// Invariant: ix >= 0
if (ix >= myVec.size())
{
    flagError("Index too large");
    return;
}
// Invariant: ix >= 0 && ix < myVec.size()
myItem = myVec[ix];
```

Suppose `myVec` is a `vector<int>`.

We wish to set (non-const) `int` variable `myItem` equal to `myVec[ix]`, if possible.

Q. When would it be impossible?

A. When `ix` is out of range, that is, when it is not a valid index for `myVec`.

Software Engineering Concepts: Invariants
Basics [2/2]

When we make assertions, the things we assert are invariants.

Invariant

```
assert(ix >= 0 && ix < myVec.size());
myItem = myVec[ix];
```

But there may also be invariants that we cannot write assertions for, since they cannot be expressed in code.

```
// Invariant: pp points to memory allocated with new [],
//   owned by *this.
delete [] pp;
```

In C++ there is no way to test for ownership or the method used to allocate memory.

Software Engineering Concepts: Invariants
Pre & Post [1/2]

We are particularly interested in two special kinds of invariants: *preconditions* and *postconditions*.

A **precondition** is an invariant at the beginning of a function.

- The responsibility for making sure the precondition is true rests with the calling code (that is, the client).
- In practice, a precondition **states what must be true for the function to execute properly**.

A **postcondition** is an invariant at the end of a function.

- It tells what services the function has performed for the client code.
- The responsibility for making sure the postcondition is true rests with the function itself.
- In practice, a postcondition **describes the function's effect using statements about values**.

A function offers an **operation contract** to its caller: "Caller, if you fulfill the preconditions, then I will fulfill the postconditions."

Software Engineering Concepts: Invariants Pre & Post [2/2]

Example

- Write reasonable pre- and postconditions for the following function, which is supposed to compute the average speed of an object, given the distance it travels and the time elapsed.

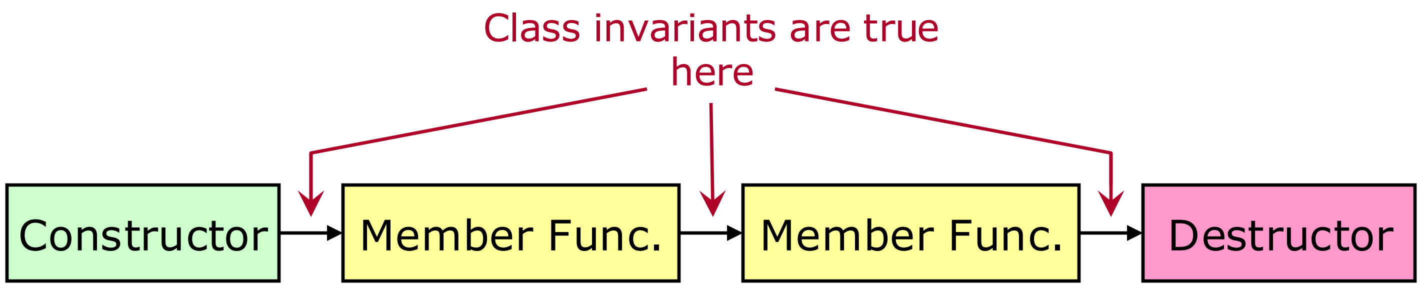
Preconditions:
What **must be true** for the function to execute properly?

```
// avgSpeed
// Pre:  time != 0.
// Post: return == dist/time, where the computation is
//        done using floating-point division.
double avgSpeed(int dist,
                int time)
{
    return double(dist) / double(time);
}
```

Postconditions:
Describe the function's effect using statements about values.

Software Engineering Concepts: Invariants
Class Invariants [1/2]

For a given class, a **class invariant** is an invariant that holds for an object of the class, whenever execution is not inside a member function.



- Class invariants are preconditions for every public member function, except constructors.
- Class invariants are postconditions for every public member function, except the destructor.
- Since we know this, you do not need to list class invariants in the pre- and postcondition lists of public member functions.
- In practice, class invariants are **statements about data members** that indicate what it means for an object to be **valid** or **usable**.

Class Invariants [2/2]

Example

- Write reasonable class invariants for the following class.

Class invariants:
statements about data members that indicate what it means for an object to be **valid** or **usable**.

```
// class Date
// Invariants:
//     1 <= _mo <= 12.
//     1 <= _day <= #days in month given by _mo.
class Date {
public:
    [Lots of code goes here]
private:
    int _mo;   // Month 1..12
    int _day;  // Day 1..#days in month given by _mo
};  // End class Date
```

Software Engineering Concepts: Invariants Documentation

We have seen preconditions and class invariants before. In files like the `TimeOfDay` package, and in Assignment 1, we typically made two kinds of assertions.

- Assertions about the parameters of a function.
- Assertions about the data members of an object.

Both are preconditions. The latter are usually class invariants.

We will require both preconditions and class invariants to be documented in comments. Class invariants are preconditions of all member functions except ctors, so we do *not* need to restate them as preconditions before every function.

TO DO

- Add comments indicating preconditions and class invariants to the `TimeOfDay` package.

Done. See `timeofday.hpp` *&* `timeofday.cpp`*.*

Invisible Functions II

Invisible Functions II
The Big Five [1/2]

Recall: the **Big Five** are the following.

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

All five are sometimes automatically generated.

Invisible Functions II
The Big Five [2/2]

The **Rule of Five**: If you define one of the Big Five, then consider whether to define or =delete each of the others. If, for one of these functions, you decide not to, then =default that one.

This typically happens when an object directly manages a resource.

We much prefer writing none of them. This is our usual way of operating.

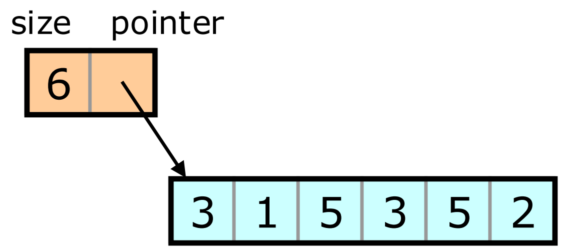
Thus, we have the **Rule of Zero**: *Where possible*, do not explicitly define any of the Big Five. Resources should be managed by data members that are objects of RAII classes.

But sometimes we need to write one of those RAII classes. And then we need to write the Big Five for that class.

Invisible Functions II
Copy vs. Move [1/3]

In order to write copy & move operations, it can be helpful to consider the difference between them.

Suppose we have an array object. Typically, this will have a pointer to a block of memory containing the array data, along with an integer whose value is the size of the array.

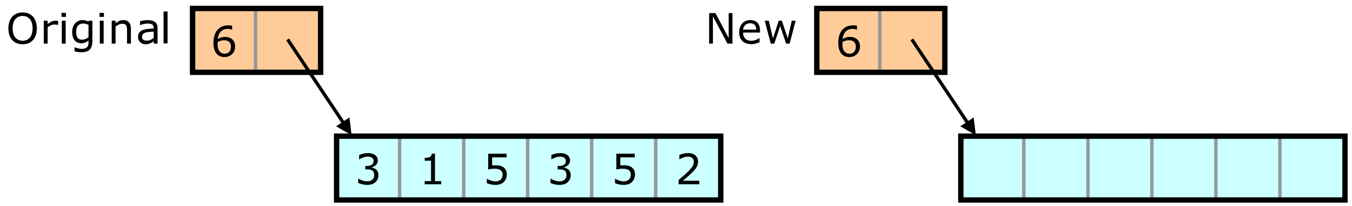


Now we want to create a new object just like it.

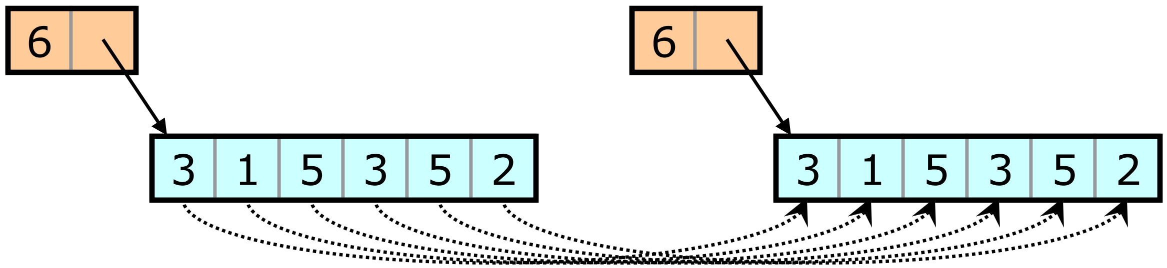
- If we are not allowed to alter the original, we are doing a **copy**.
- If we are allowed to alter the original, we are doing a **move**.

Invisible Functions II
Copy vs. Move [2/3]

To do a **copy**, we first create our new object, set its size member, and allocate a memory block of the correct size.



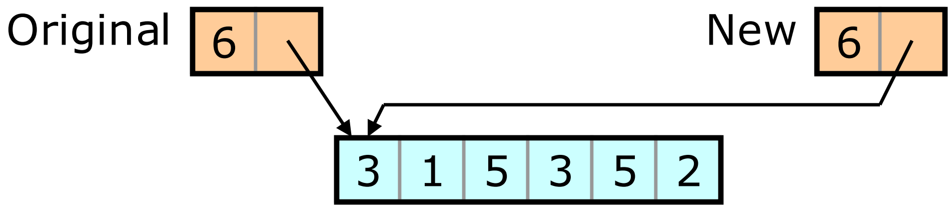
Then we copy each array item to the new memory.



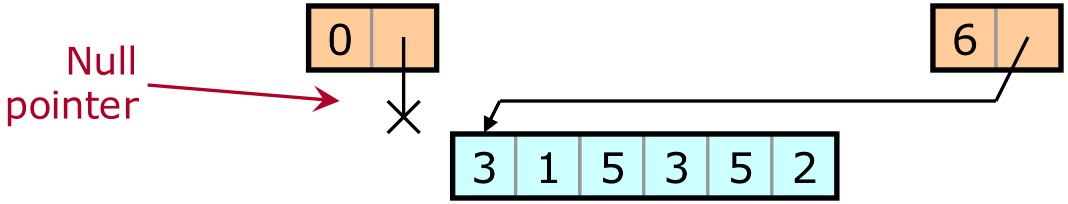
If the array is large, then this can be time-consuming. If the array items are complicated, then it is possible for an item to copy unsuccessfully, and we will have to deal with the error.

Invisible Functions II
Copy vs. Move [3/3]

A **move** can use a different strategy. First, set each data member of the new object to the corresponding member in the original.



The new object is finished. But leaving the original pointing to the same memory is a problem. So we set the original to a "nothing" value that can still be correctly destroyed.



And we are done. So a move operation can be both fast and free from the possibility of errors.

Invisible Functions II
Writing Them — Assumptions

We consider how to write the Big Five for a class under the following assumptions.

- Every data member has a built-in type: things like `int`, `std::size_t`, `double`, and any pointer type—including `(Foo *)` when `Foo` is a class we wrote.
- Objects of our class will be destructible, copyable, and moveable
  - So we will not =delete any of the Big Five.
- There are no inheritance hierarchies involved.
  - So there are no virtual functions and no base-class initializers.

On the following slides, we will discuss one way to write the Big Five for a class `Foo` with data members `_a` and `_b`.

Invisible Functions II
Writing Them — Dctor & Copy Ctor

Write the dctor and the copy ctor however we need to.

- The dctor must release any owned resources.
- The copy ctor needs to make a real copy.
  - If some member is a pointer referencing a dynamic array, then do *not* copy the pointer. Instead, allocate a new array and then copy from old array to new array.

```
class Foo {
public:
    // Dctor
    ~Foo()
    {
        …
    }
```

```
// Copy ctor
Foo(const Foo & other)
    :_a(…),
     _b(…)
{
    …
}
```

Invisible Functions II
Writing Them — Move Ctor

A move ctor makes an object with the same value as its parameter (`other`). It may alter `other`. But `other` still needs to be destructible.

Procedure

- Construct each data member from the corresponding member of `other`.
- Set `other` to a value that can be destroyed—without messing up our object.

A move ctor should be marked `noexcept`, which promises that it throws no exceptions. This allows optimizations that can improve efficiency.

We will discuss exceptions on another day

```
// Move ctor
Foo(Foo && other) noexcept
    :_a(other._a),
     _b(other._b)
{
    other._a = …;
    other._b = …;
}
```

Set `other` to a valid value, so its destructor still works. This value should be one whose destruction does *not* mess up our newly constructed object.

Invisible Functions II
Writing Them — Swap

A useful operation is a **swap** member function.

- Take another object of the same type.
- Swap the values of this object and the other object.

Swap can often be implemented very efficiently: call Standard Library function `swap` (`<utility>`) to swap each data member with the corresponding data member of the other object.

Generally, we should mark a swap member function as `noexcept`.

This member function will sometimes be private.

```
private:
    void mswap(Foo & other) noexcept
    {
        swap(_a, other._a);
        swap(_b, other._b);
    }
```

Traditionally, this member function is named `swap`. Here, I call it `mswap` (for “member swap”) to avoid confusion with `std::swap`. But if it is private, then you can call it whatever you want.

Invisible Functions II
Writing Them — Copy & Move Assignment

Once we can swap, the assignment operators are easy to write.

- Copy assignment swaps with a copy of its parameter.
- Move assignment swaps with its parameter. It should be marked `noexcept`.

```
Foo & operator=(const Foo & rhs)      // Copy assignment
{
    auto rhs_copy = rhs;
    mswap(rhs_copy);
    return *this;
}

Foo & operator=(Foo && rhs) noexcept  // Move assignment
{
    mswap(rhs);
    return *this;
}
```

This is *one way* to write assignment operators. It is easy, and it works.

For some classes, there may be better ways to write these—but we will not need to worry about that this semester.

An assignment operator should always return the current object.