Managing Resources in a Class continued
Containers & Iterators

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, September 9, 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
- (part) Managing resources in a class
- Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions

Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Assertions
- ✓ Testing
- Invariants

Review

Review
Parameter Passing I/II [1/2]

Four methods for passing a parameter or returning a value are used in C++:

	By Value `U f(T x)`	**By Reference** `U & f(T & x)`	**By Reference-to Const** `const U & f(const T & x)`	**By Rvalue Reference** `U && f(T && x)`
Makes a copy	YES ☹	NO ☺	NO ☺	NO ☺
Allows for polymorphism	NO ☹	YES ☺	YES ☺	YES ☺
Allows implicit type conversions	YES ☺	NO ☹	YES ☺	YES ☺
Allows passing of:	Any copyable value ☺	Non-const Lvalues ☹?	Any value* ☺	Non-const Rvalues*

*Rvalues *prefer* to be passed by Rvalue reference.

We do not pass by Rvalue reference very often.

When we do so, we might write two versions of a function.

```
void g(Foo && p);        // Gets non-const Rvalues
void g(const Foo & p);   // Gets all other values
```

Since it is okay to “mess up” a non-const Rvalue, the first version can often be written to be faster. But if it cannot, then there is no point in writing it at all.

Review
Invisible Functions I [1/2]

A C++ compiler may write a number of member functions for us. Here are six important ones:

The **Big Five**

```
class Dog {
public:
    Dog();                                   Default ctor
    ~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
```

For each function, the automatically generated version calls the corresponding member function for each data member.

Review
Invisible Functions I [2/2]

Two special options.

- Force automatic generation. `Dog(Dog && other) = default;`
- Eliminate the function. `Dog(Dog && other) = delete;`

The default ctor is automatically generated when we declare no ctors.

For the Big Five, we covered the rules for when they are automatically generated. But you do not need to know these; just follow the Rule of Five.

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.

Typically, this happens when an object directly manages some resource—like dynamically allocated memory—that will need to be cleaned up.

Review
Managing Resources in a Class [1/2]

Exceptions may cause a function to exit, even where there is no return. Destructors of automatic objects are still called.

Dynamically allocated memory & objects need clean-up when we are done with them. If we never **deallocate**, then there is a **memory leak**.

To **own** memory/object = to be responsible for releasing (deallocating).

Ownership =
Responsibility
for Releasing

Prevent memory leaks with **RAII**.

- Memory/object 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
Managing Resources in a Class [2/2]

TO DO (last time)

- Write class `IntArray` in header `intarray.hpp`.
  - Constructor from size (explicit).
  - Destructor.
  - Bracket operator (const & non-const).
  - Member types `size_type`, `value_type`.
- Rewrite function `scaryFn` to use `IntArray`.

Done. See `intarray.hpp`.

For a program that uses `IntArray`, see `intarray_main.cpp`.

We cover this shortly.

Managing Resources in a Class

continued

Managing Resources in a Class
An RAII Class — Usage in a Function

Original `scaryFn`

```
void scaryFn(size_t size)
{
    int * buffer = new int[size];
    if (func1(buffer))
    {
        delete [] buffer;
        return;
    }
    if (func2(buffer))
    {
        delete [] buffer;
        return;
    }
    func3(buffer);
    delete [] buffer;
}
```

New `scaryFn`, using `IntArray`

```
void scaryFn(size_t size)
{
    IntArray buffer(size);
    if (func1(&buffer[0]))
        return;
    if (func2(buffer))
        return;
    func3(&buffer[0]);
}
```

This line supposes that `func2` has been rewritten to take an `IntArray` parameter.

The parameter cannot be passed by value, because `IntArray` has no copy/move ctors.

Managing Resources in a Class
An RAII Class — Usage in a Class

Class with an Array Member

```
class HasArray {
public:
    HasArray(size_t size)
        :_theArray(new int[size])
    {}

    ~HasArray()
    { delete [] _theArray; }

    …

    void out(size_t index) const
    { cout << _theArray[index]; }

private:
    int * _theArray;
};
```

Same idea, using `IntArray`

```
class HasArray {
public:
    HasArray(size_t size)
        :_theArray(size)
    {}

    // Auto-generated dctor

    …

    void out(size_t index) const
    { cout << _theArray[index]; }

private:
    IntArray _theArray;
};
```

Same

Managing Resources in a Class
Generalizing Ownership [1/3]

The concepts of ownership and RAII can be applied to resources other than dynamically allocated memory.

- An open file (who is responsible for closing it?)
- Network connections.
- Or anything else that needs clean-up when we are done with it.

Acquire a resource: get access and control.

Release a resource: clean it up and relinquish control.

So:

- If a resource is never released, then we have a **resource leak.**
- The **owner** of a resource is responsible for releasing it.
- **RAII**: an object owns a resource; so its destructor releases.
- Direct resource ownership is the usual reason to define/=delete the Big Five.

RAII is used by standard stream classes, to manage open files.

```
bool handleInput(const std::string & filename)
{
    std::ifstream inFile(filename);
    if (!inFile) return false;
    for (int i = 0; i < 10; ++i)
    {
        int inValue;
        inFile >> inValue;
        if (!inFile) return false;
        processInput(inValue);
    }
    return true;
}
```

Q. Where is the file closed?

A. In the dctor of `inFile`.

Strictly speaking, not here, since this exit is taken if the file could not be opened. But if you guessed this spot, then your heart is in the right place. ☺

Here or **here** …

… or *possibly* **here**, if `processInput` may throw an exception.

Managing Resources in a Class
Generalizing Ownership [3/3]

Class `IntArray` is just an exercise. The C++ Standard Library already includes smarter RAII array class templates (`std::vector`, `std::array`, and `std::basic_string`), as well as simpler ownership-only smart-pointer classes (`std::unique_ptr` and `std::shared_ptr`).

However, we can and do apply the ideas of ownership and RAII in real-world projects.

In situations where no existing resource-management classes fit our needs, we might need to write one or more, based on the principles covered here.

Containers & Iterators

Containers — Generic Containers

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 of generic container: a C++ built-in array.

```
MyType myArray[8];
```

Other generic container types are in the C++ Standard Library.

In particular, the **Standard Template Library** (**STL**), contains templates for many data structures that can hold arbitrary types, as well as algorithms that can deal with arbitrary types.

Containers — Troubles with C++ Built-In Arrays

STL containers are necessary for many reasons. One is that C++ built-in arrays have very few operations defined on them.

- There is no resizing and no “size” member function—no member functions at all, actually.
- There is no copy or assignment. When a built-in array is passed by value, it **decays** to a pointer to its first item.

```
int a[10];
func(a);
func(&a[0]); // Same as above
// func cannot tell the size of the array it receives
```

Containers — std::vector [1/3]

We would prefer a container type that is *first-class*.

A type is **first-class** if it can be tossed around with the ease of something like `int`—for example, new values can be created at runtime, they can be passed to and returned from functions, and they can be stored in containers).

One generic container found in the STL: `std::vector`.

- `vector` is a first-class array.
- It is declared in the standard header `<vector>`.
- This is a **class template**, not a class.

```
vector v1;          // DOES NOT COMPILE!
vector<int> v2;     // vector of int
```

Containers — std::vector [2/3]

Like any array, `vector` has lookup by index:

```
vector<int> v3(20);  // Much like int arr[20];
cout << v3[5] << endl;
v3[19] = 7;
```

A `vector` knows how to copy itself:

```
v3 = v2;
```

A `vector` knows its size.

```
cout << v3.size() << endl;
```

A default-constructed `vector` has size 0. But there are other ctors.

```
vector<Blug> v4(20);  // Holds 20 items of type Blug;
                      //  all are default-constructed
vector<double> v5(55, 7.);  // Holds 55 doubles, all 7.
```

And we can change the size of a `vector`:

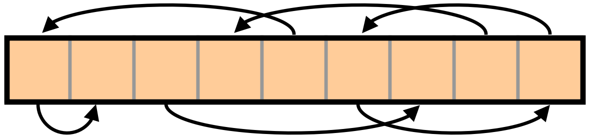
```
v5.push_back(6.1);  // Adds new item at end, value 6.1
v5.pop_back();      // Eliminates last item
v5.resize(20);      // v5 now has size 20
```

I call `std::vector` a **smart array**.

Containers — Kinds of Data

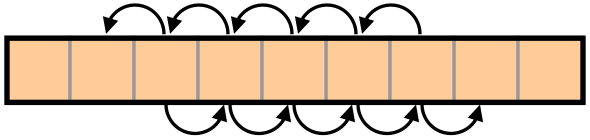
When we deal with containers, the following broad categories of data are important:

- **Random Access**
  - Random-access data can be dealt with in any order. We can efficiently skip from one item to any other. Example: `std::vector`.

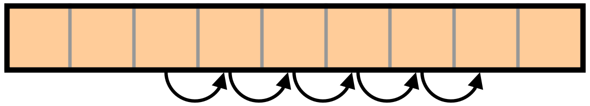


Random Access

- **Sequential Access**
  - Sequential-access data is data that can only be dealt with—or only dealt with efficiently—in a particular order. We begin with some item, then proceed to the next, etc.
  - Sequential access data may be **bidirectional**, accessible in both forward and backward order. Or it may be **forward-only**, accessible only in forward order.



Bidirectional Sequential Access



Forward-Only Sequential Access

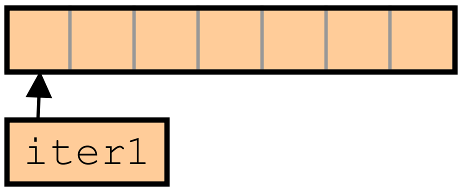
The STL includes a number of generic containers. Some are random-access; others are sequential-access.

- `std::vector`
- `std::basic_string`
- `std::array`
- `std::list`
- `std::forward_list`
- `std::deque`
- `std::map`
- `std::set`
- `std::unordered_map`
- `std::unordered_set`
- `std::multimap`
- `std::multiset`
- `std::unordered_multimap`
- `std::unordered_multiset`

All of these have interfaces that involve *iterators*.

Containers & Iterators
Iterators — Introduction [1/4]

An **iterator** refers to an item in a container.




```
vector<int> v(7);
vector<int>::iterator iter1 = begin(v);
```

STL containers have iterator member types.

`auto` would be nicer here.

Global function `begin` (`<iterator>`) calls member function `begin`, which returns an iterator to the first item in the container.

An iterator does *not* own the item it refers to.

Use the dereference operator (`*`) to access the item an iterator refers to. The item is available as an Lvalue.

```
v[0] = 3;
cout << *iter1;  // Prints "3"
*iter = 5;       // Set v[0] to 5
```

Iterators — Introduction [2/4]

STL containers actually have multiple iterator member types.

```
vector<int>::iterator it;
vector<int>::const_iterator cit;
    // Does not allow modification of referenced item

cout << *it;    // Okay
*it = 5;        // Okay
cout << *cit;   // Okay
*cit = 5;       // DOES NOT COMPILE!
```

Containers & Iterators
Iterators — Introduction [3/4]

Non-owning pointers are iterators for C++ built-in arrays.

```
int arr[8];
int * p = &arr[2];
*p = 7;  // Sets arr[2] to 7
```

The syntax used for iterators in C++ was based on the syntax for pointers, which is the same as the the pointer syntax in the C programming language.

Iterators — Introduction [4/4]

An iterator can be a **wrapper** around data, to make it look like a container.

```
#include <iterator>
using std::ostream_iterator;

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

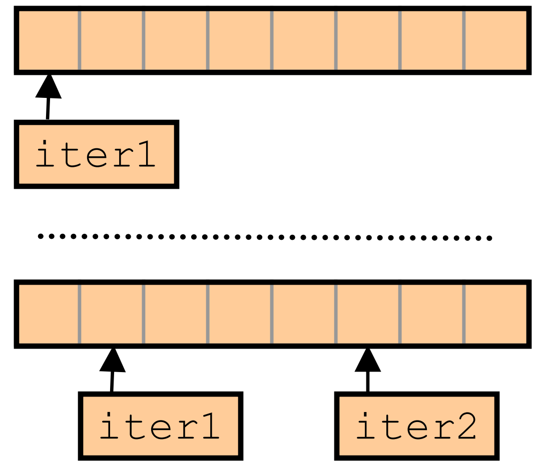
Now the following two lines do the same thing:

```
cout << 3 << "\n";
*coolIter++ = 3;  // Has same effect as previous line
```

Containers & Iterators
Iterators — Operations [1/3]

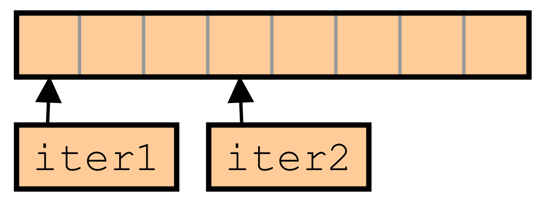
Adding to an iterator moves the iterator forward some number of steps to a new item in the same container.

```
++iter1;
auto iter2 = iter1 + 4;
```



Similarly, subtracting moves an iterator backward.

```
--iter1;
iter2 -= 2;
```



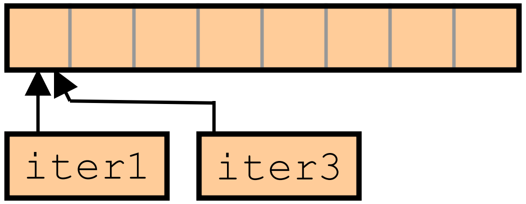
Containers & Iterators
Iterators — Operations [2/3]

Subtract two iterators *to the same container*, to find the **distance** between them.

```
auto dist = iter2 - iter1;  // dist is an integer
```

Copying an iterator gives a new iterator referring to the same item.

```
auto iter3 = iter1;
```



Checking equality of iterators tells whether they refer to the same spot in the container.

```
if (iter3 == iter1)
    …
```

Iterators — Operations [3/3]

Operations available on an iterator match the underlying data.

- Iterators for forward-only sequential-access data have the `++` operation. These are **forward iterators**.

```
++forwardIterator;
```

- Iterators for bidirectional sequential-access data also have the `--` operation. These are **bidirectional iterators**.

```
++bidirectionalIterator;
--bidirectionalIterator;
```

- Iterators for random-access data have all the iterator arithmetic operations. These are **random-access iterators**.

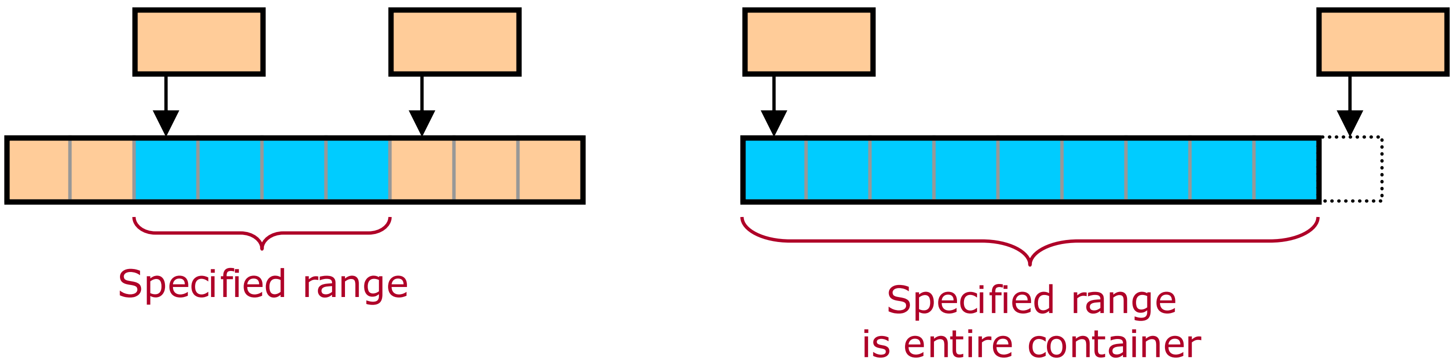
```
++randomAccessIter;
--randomAccessIter;
randomAccessIter += 7;
cout << randomAccessIter[5];
std::ptrdiff_t dist =
          raIter2 - raIter1;
```

Each boldface term is an **iterator category**.

Containers & Iterators
Iterators — Specifying Ranges

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.




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

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

Global function `end` (`<iterator>`) calls member function `end`, which returns an iterator to *just past* the last item in a container.

Iterators — Range-Based For-Loop [1/2]

Iterators are fundamental to the **range-based for-loop**, a flow-of-control construct introduced in the 2011 C++ Standard.

```
vector<int> data;

for (auto x : data)
    cout << x << " " << endl;
```

`x` becomes a copy of each item in container `data`.

The above is essentially the same as the following.

```
for (auto it = begin(data); it != end(data); ++it)
{
    auto x = *it;
    cout << x << " " << endl;
}
```

Containers & Iterators
Iterators — Range-Based For-Loop [2/2]

The variable in a range-based for-loop is treated much like a parameter. The usual parameter-passing methods are available.

We generally use by reference-to-const for containers of objects.

```
vector<Blug> data2;

for (const auto & x : data2)
    cout << x << endl;
```

Use by reference to allow alteration of items in the container.

```
for (auto & x : data2)
    x = bb;
```

Here, `x` is not a copy.

Containers & Iterators
TO BE CONTINUED …

Containers & Iterators will be continued next time.