Containers & Iterators continued
Invisible Functions II
Thoughts on Project 2

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, September 9, 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
- (part) Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions
- A little about Linked Lists

Major Topics: S.E. Concepts

- ✓ Invariants
- ✓ Testing
- Abstraction

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 can be transferred, shared, and chained.

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 or =delete each of the Big Five in an RAII class.

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

Review
Containers & Iterators

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**).

The STL includes `std::vector`, a smart array template.

```
#include <vector>
using std::vector;

vector<int> iv(20);
iv.push_back(4);
cout << iv.size() << endl;  // Prints "21"
cout << iv[20] << endl;     // Prints "4"
```

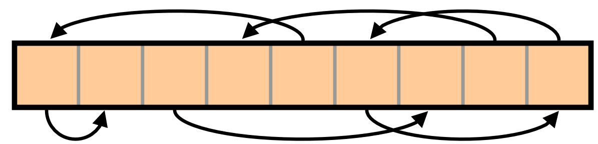
Containers & Iterators

continued

Containers & Iterators
Containers — Kinds of Data

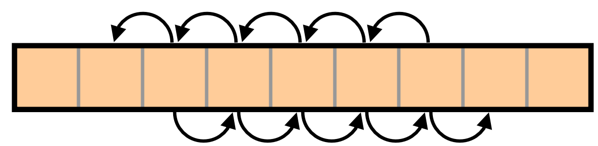
When we deal with containers, the following broad categories of data are important:

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

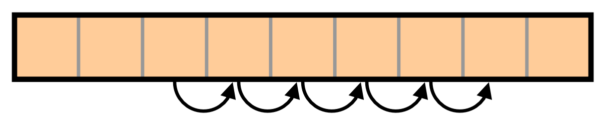


Random Access

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



Two-Way Sequential Access



One-Way Sequential Access

Containers & Iterators
Containers — STL Generic Containers

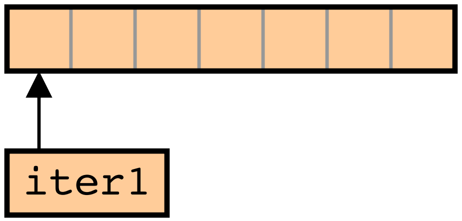
The STL includes a number of generic containers. Some are random-access; others are sequential-access.

- `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.



STL containers have iterator member types.

```
vector<int> v(8);
vector<int>::iterator iter1 = begin(v);
```

`auto` would be helpful 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 << *iter;   // 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!
```

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 derived from 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");
```

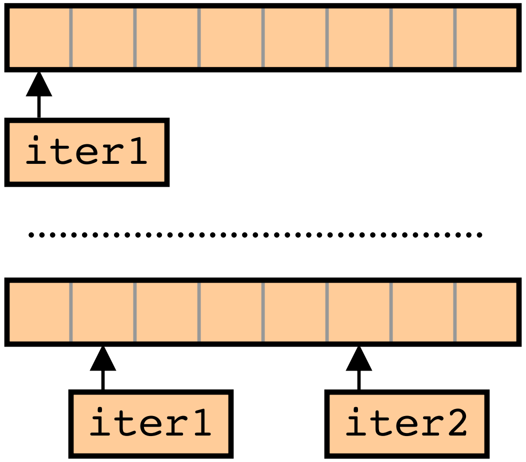
Now the following two lines do the same thing:

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

Iterators — Operations [1/3]

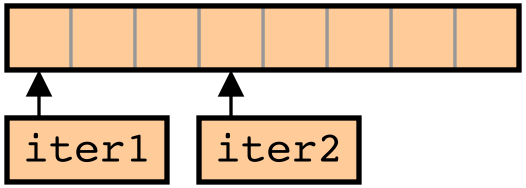
Adding to an iterator moves the iterator forward some number of steps to a new item in the same container.

```
++iter1;

auto iter2 = iter1 + 4;
```



Similarly, subtracting moves an iterator backward.

```
--iter1;

iter2 -= 2;
```



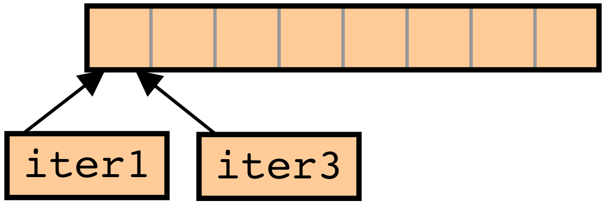
Iterators — Operations [2/3]

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)
    …
```

Containers & Iterators
Iterators — Operations [3/3]

Operations available on an iterator match the underlying data.

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

```
++forwardIterator;
```

- Iterators for two-way 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**.

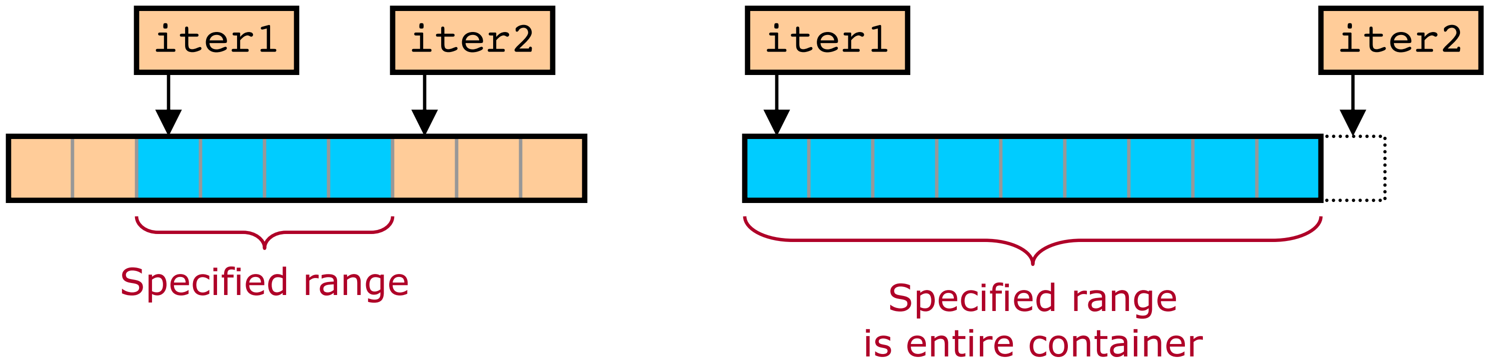
```
++randomAccessIter;
--randomAccessIter;
randomAccessIter += 7;
cout << randomAccessIter[5];
std::ptrdiff_t dist =
          raIter2 - raIter1;
```

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

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.



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

```
#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
```

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 pretty much 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
Generic Algorithms [1/3]

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!
```

Generic Algorithms [2/3]

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 [3/3]

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

- `std::equal`: check if two ranges have 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
CODE

TO DO

- Run some code using iterators and STL algorithms.

See `iterators.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 op
Dog(Dog && other);                    // Move ctor
Dog & operator=(Dog && rhs);          // Move assignment op
```

All five are sometimes automatically generated. But when we write them ourselves, we need to consider *how* we would write them.

Invisible Functions II
The Big Five [2/2]

The **Rule of Five**:

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

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**:

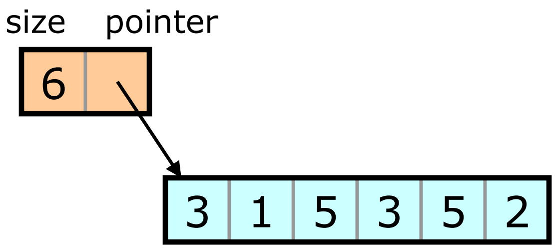
Where possible, do not explicitly define any of the Big Five. Resources should be managed by data members that are objects of RAII classes.

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.

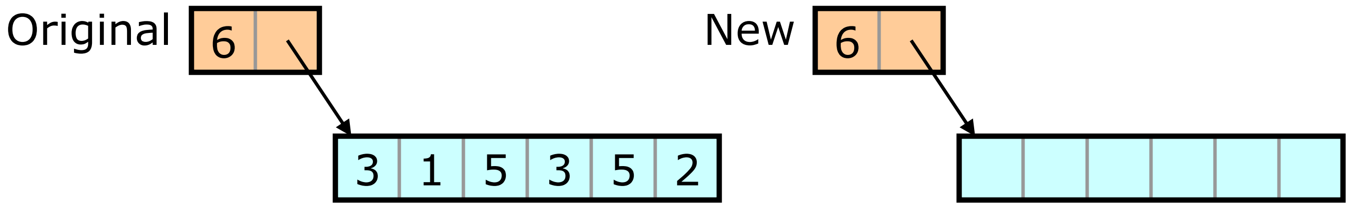
Suppose we have an array object. Typically, this will have a pointer to a block of memory containing the array data, along with an integer whose value is the size of the array.



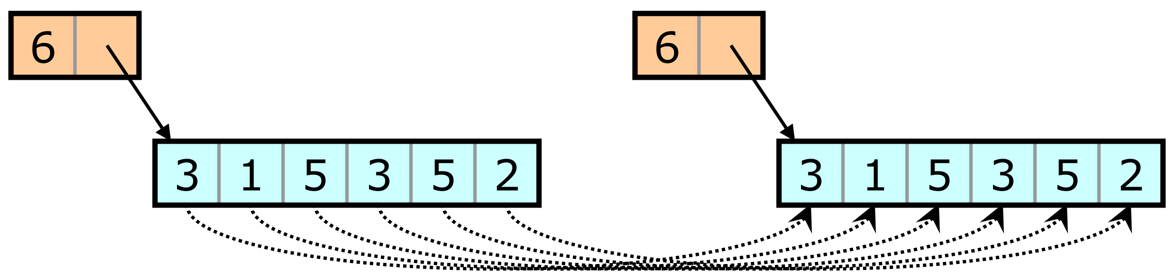
Now we want to create a new object just like it.

- 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**.

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

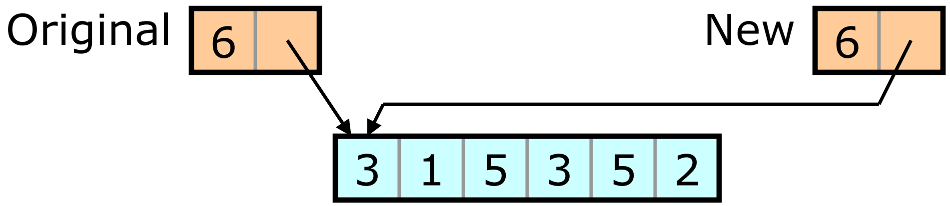


Then we copy each array item to the new memory.

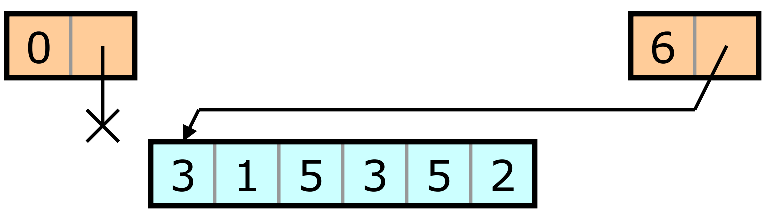


If the array is large, then this can be time-consuming. If the array items are complicated, then it is possible for an item to copy unsuccessfully, and we will have to deal with the error.

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 be discussing how 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 clean up 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.

```
// 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.

We will discuss exceptions on another day

Invisible Functions II
Writing Them — Swap [1/2]

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 typically 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 — Swap [2/2]

For the convenience of other classes, we *might* write a noexcept global function `swap`. This just calls the `mswap` member.

If `mswap` is private, then our global `swap` must be a friend.

```
    friend void swap(Foo & a, Foo & b) noexcept;
private:
    void mswap(Foo & other) noexcept
    { … }
};  // End class Foo

void swap(Foo & a, Foo & b) noexcept
{
    a.mswap(b);
}
```

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 `noexcept`.

```
Foo & operator=(const Foo & rhs)      // Copy assignment
{
    Foo copy_of_rhs(rhs);
    mswap(copy_of_rhs);
    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.

Thoughts on Project 2

Thoughts on Project 2
Overview of Ideas

In Project 2 you implement a “moderately smart” array (`MSArray`). This will require applying some recently covered ideas.

- Integer Types
  - What type will you use for the size of an `MSArray`? For array indices?
- Managing Resources in a Class
  - Are you doing dynamic allocation correctly? When you allocate something, is it always freed?
  - `MSArray` should use RAII. This affects how you write it and how you document it.
- Containers & Iterators
  - `MSArray` is a generic container. Its member functions `begin` and `end` return iterators.
- Invisible Functions II
  - `MSArray` directly manages a resource. You will need to define all of the Big Five. (So Project 2 is *not* a place to apply the Rule of Zero!)

Thoughts on Project 2
Documentation [1/2]

`MSArray` and all global functions will be templates.

When we define a template, the things between the angle brackets are **template parameters**.

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

Template parameters

```
template <typename ABC, typename XYZ>
```

This semester, all templates must have documented **requirements on types** that specify what must be true about the template parameters. Typically, these will say that the type must have certain member functions.

```
// Requirements on Types:
//     XYZ must have a copy ctor.
template <typename ABC, typename XYZ>
void fff(const ABC & n1, XYZ n2);
```

Thoughts on Project 2
Documentation [2/2]

We still need to document **preconditions** for all functions that have them, and **class invariants** for all classes.

```
// Invariants: …
// Requirements on Types: ValType must have …
template <typename ValType>
class MSArray { … };
```

Maybe

```
// Pre: …
// Requirements on Types: ValType must have …
template <typename ValType>
bool operator==( … )
```

What must be true about `ValType` for this template to work?

Typically: List member or global functions that must be defined for `ValType`.

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?*).

```
… begin()
{ return …; }
…
… end()
{ return begin() + …; }
```

Pointer type

Access to Internal Data

A `const MSArray` has non-modifiable data. If a function gives 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) the return method.

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

Thoughts on Project 2
Writing the Big Five

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!