CSE 1325: Object-Oriented Programming

Lecture 21 / Chapters 20-21

Standard Template Library (STL)

Containers and Iterators

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

Or by appointment

Overview: Containers and Iterators

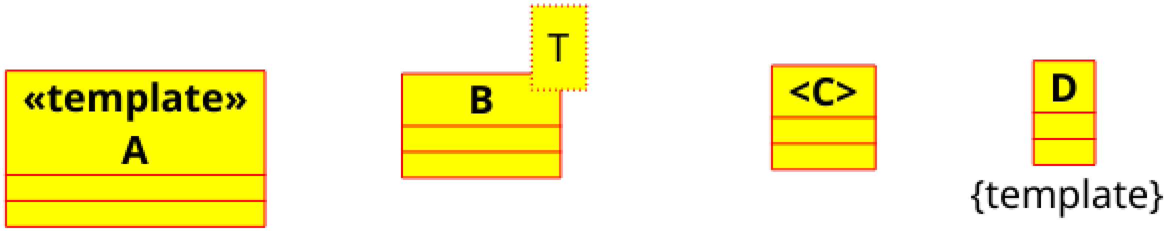
- Containers
  - Sequence Containers
  - Container Adapters
  - Associative Containers
  - Unordered Associative Containers
- Iterators

Cargo container ship photo cc0 Public Domain
https://www.publicdomainpictures.net/en/view-image.php?image=104269&picture=cargo-container-ship

Lecture 20

Quick Review

- Writing algorithms in terms of types that are specified as parameters during instantiation or invocation is called **Generic Programming**.
- A **template** is a C++ construct representing a function or class in terms of generic types.
- **True** or False: Both the declaration AND definition of a template class must be placed in the header file.
- The **Standard Template Library** provides a good variety of well-implemented, mostly non-numerical algorithms focused on organizing code and data as C++ templates.
- A template is represented in the UML as **B**.



Review

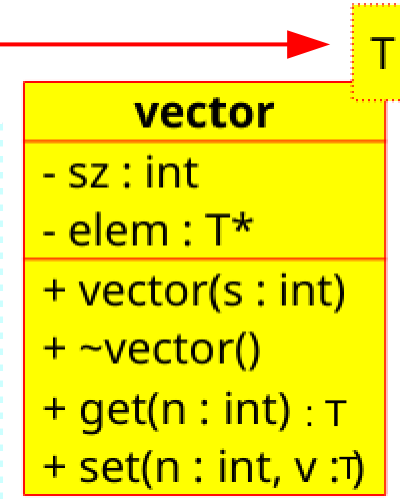
Template Summary

- **Template** – A C++ construct representing a function or class in terms of generic types
  - This enables the algorithm to be written independent of the types of data to which it applies
  - A type must be specified when the template class is instanced or the template function is called
  - Templates rely on “Duck Typing”: The type must supply whatever methods are access by the template
- Potentially reusable algorithms that are suitable should generally be defined as templates

Review

Vector v0.4 – Now a Template

UML Template Class indicator




```
template <class T>
class vector {
   int sz;        // the size
   T* elem;       // a pointer to the elements
public:
   vector(int s) : sz(s), elem{new T[s]} { }
   ~vector() {delete[ ] elem; }
   T get(int n) const { return elem[n]; }  // access: read
   void set(int n, T v) { elem[n]=v; }     // access: write
};
```

- To convert a class to a template:
  - Add template specifier before the class declaration
  - Replace the type with T

```
ricegf@pluto:~/dev/cpp/201801/21$ make vector
g++ -std=c++14    vector.cpp    -o vector
ricegf@pluto:~/dev/cpp/201801/21$ ./vector
ricegf@pluto:~/dev/cpp/201801/21$
```

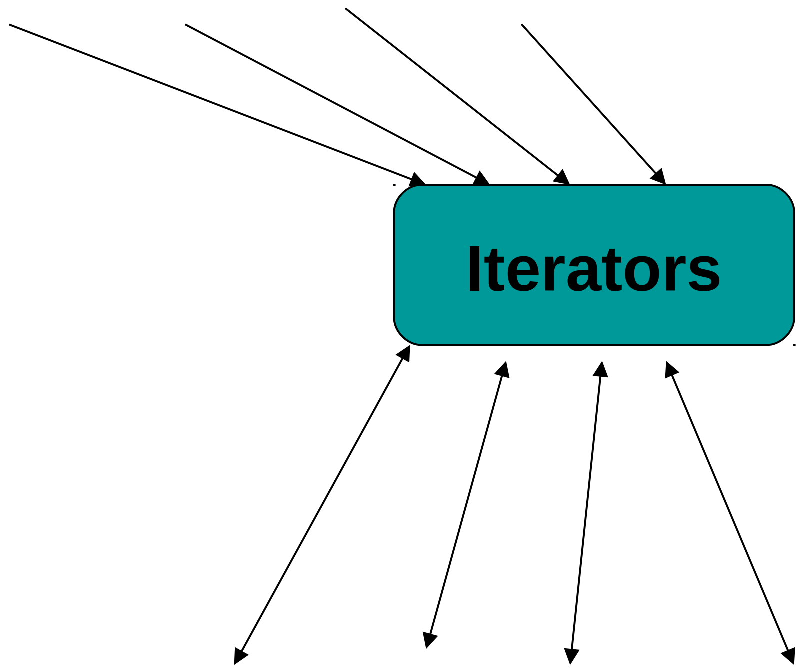
The Standard Template Library (STL)

- The STL provides many well-implemented, mostly non-numerical **algorithms** focused on organizing code and data as C++ templates
  - search, count, for_each, copy, transform, fill, reverse, sort, merge
- The STL also provides many **containers** for managing data
  - vector, array, stack, queue, set / map (multi and unordered variants)
- Each container includes nested **iterators** to enable the algorithms to manipulate the containers
- The STL also provides many **functors** (a class with operator() defined, thus an instance of a functor can be called just like a function)

Review
Basic STL Model

Algorithms

Sort, find, search, copy, ...



vector, list, map, unordered_map, ...

Containers

- Separation of concerns
  - Algorithms manipulate data, but don’t know about containers
  - Containers store data, but don’t know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

The STL also defines some **Functors**, but we’ll ignore those...

STL Container Overview

- The STL contains the following containers of interest
  - **Sequence Containers** store data linearly
    - vector, array, list, forward_list, and deque (string is a non-STL sequence container)
  - **Container Adapters** are façades to Sequence Containers
    - stack, queue, and priority_queue
  - **(Ordered) Associative Containers** provide fast lookup via keys by maintaining the data in sort order
    - set, map, multiset, and multimap
  - **Unordered Associative Containers** are unsorted, and thus provide somewhat faster insertion but slower lookups via a hash
    - unordered_set, unordered_map, unordered_multiset, unordered_multimap
- Let's take a brief look at each

Sequence Containers
vector and array

- **Vector** provides a resizable, flexible version of the C / C++ standard array
  - Very fast lookup (e.g., foo[42])
  - Relatively slow insert / delete except at end
- We already know vector fairly well
- **Array** is basically a fixed-size vector
  - All memory is allocated when instanced
  - While vector has 24 additional bytes of overhead beyond the data buffer, array has none
  - Array is very useful for memory-constrained systems and for embedded programming

http://www.cplusplus.com/reference/vector/vector/
http://www.cplusplus.com/reference/array/array

Sequence Containers

deque (**d**ouble-**e**nded **que**ue)

- **Deque** provides a version of vector that supports fast additions at *both ends* of the underlying array
  - Very fast lookup (e.g., foo[42])
  - Relatively slow insert / delete except at the beginning or end
- Deques can push_back and pop_back like a vector, but also push_front and pop_front
- Supports operator[ ] just like vector (indexes always start at 0)

http://www.cplusplus.com/reference/deque/deque

Sequence Containers

list and forward_list

- Linked lists provide chains of data objects connected via one or more pointers
  - Relatively slow lookup (requires iterators, q.v.)
  - Very fast insert / delete anywhere
- Two types of lists are supported
  - **List** provides a double-linked list that can be navigated forward (++) or backward (–)
  - **Forward_list** provides a single-linked list that can be navigated forward (++) only, using slightly less memory with faster insertion / deletion compared to list
- Lists do NOT support operator[ ]

http://www.cplusplus.com/reference/list/list
http://www.cplusplus.com/reference/forward_list/forward_list/

Sequence Containers
string

- OK, this is NOT a “real” STL container, but it works very much like a vector of char
  - Relatively fast lookup (e.g., foo[42])
  - Very slow insert / delete
- Actually, string is just a typedef for basic_string<char>
  - Other string types are not uncommon, e.g., basic_string<wchar>
  - More on this next week

http://www.cplusplus.com/reference/string/string/

Sequence Adapters
stack and queue

- **Stack** provides a Last-In, First-Out (LIFO) stack
  - Very fast single-ended push and pop only
  - By default, stack is a façade for a **deque**
  - Can be specified to wrap a **vector** or **list** instead, e.g., stack<int, std::vector<int>> lifo_of_ints;
- **Queue** provides a First-In, First-Out (FIFO) stack
  - Very fast opposite-ended push and pop only
  - By default, queue is a façade for a **deque**
  - Can be specified to wrap a **list** instead, e.g., stack<int, std::list<int>> fifo_of_ints;

http://www.cplusplus.com/reference/stack/stack/
http://www.cplusplus.com/reference/queue/queue/

Sequence Adapters
priority_queue

- **Priority_queue** provides a stack for which the *highest priority* item is always popped next
  - Fast push and constant time lookup for pop
  - By default, priority_queue is a façade for a **vector**
  - Can be specified to wrap a **deque** instead, e.g., stack<int, std::list<int>> fifo_of_ints;
- As with sort, you may need to provide a comparator method for priority_queue to use

http://www.cplusplus.com/reference/queue/priority_queue/

Associative Containers
set and map

- **set** is a collection of keys, sorted by keys
  - Essentially a vector of objects with duplicates automatically removed, and always sorted
- **map** is a collection of key-value pairs sorted by keys
  - Essentially a vector of objects with (almost) any type as the key, and always sorted
  - Elements are accessed like a vector, e.g., foo[“bar”]
- As with sort and priority_queue, you may need to provide a comparator method for ordered containers
  - Associative Containers also allow non-default allocators, which we won’t cover in this class

http://www.cplusplus.com/reference/set/set/
http://www.cplusplus.com/reference/map/map/

We’ll discuss sets and maps in more detail next class!

Associative Containers
multiset and multimap

- **multiset** and **multimap** are similar to set and map, but permit duplicate keys
  - For example, a multiset of chars would allow you to store every character in a document separately, then use the count method to determine how many
  - Access is a bit trickier – foo[“bar”] is ambiguous – so iterators are needed (coming shortly)
- As with set and map, you may need to provide a comparator method for ordered containers
  - You may again provide a non-default allocator

http://www.cplusplus.com/reference/set/multiset/
http://www.cplusplus.com/reference/map/multimap/

Unordered Associative Containers

Unordered_*

- **Unordered_set, unordered_map, unordered_multiset, unordered_multimap,** similar to the ordered versions
  - Because they are not sorted, insertions are slightly faster but lookups are slightly slower
  - I've never actually needed one of these
- As with set and map, you may need to provide a comparator method for ordered containers
  - Unordered Associative Containers also allow non-default allocators and hashes, which we won't cover in this class

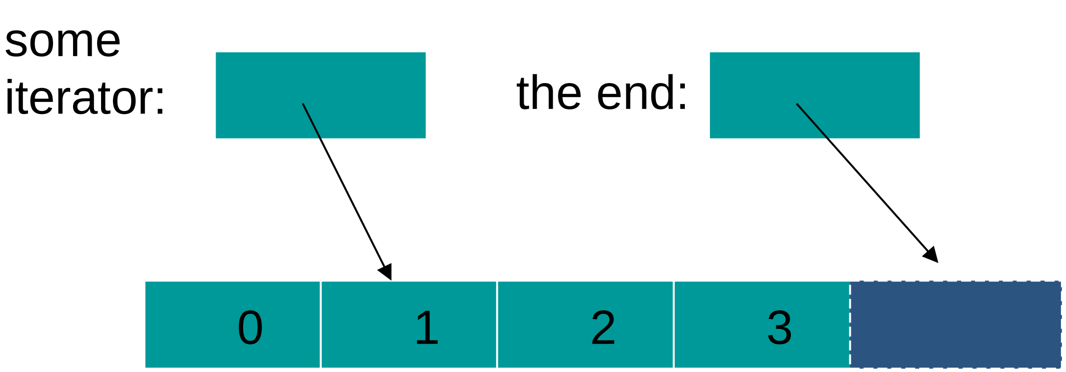
http://www.cplusplus.com/reference/unordered_set/
http://www.cplusplus.com/reference/unordered_map/

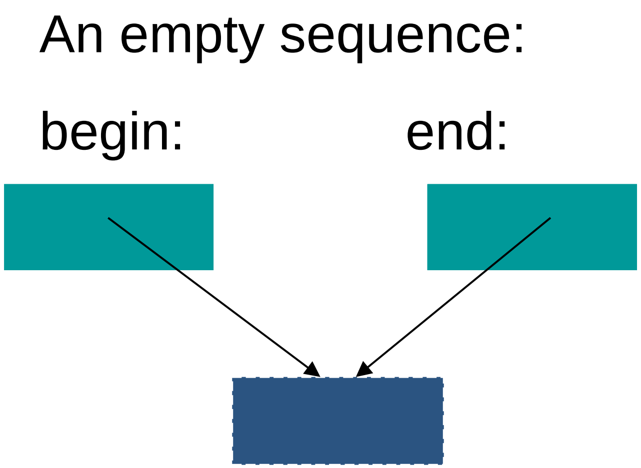
Iterators

- **Iterator**: A pointer-like instance of a nested class used to access items managed by the outer class instance
  - An iterator can always be incremented via ++, dereferenced with *, and compared to other iterators
  - The outer class provides pre-instanced iterators using methods such as begin() and end()
- Two complementary nested iterator classes are typically provided by a container
  - Iterator allows the dereferenced item to be modified
  - const_iterator prevents modification to the dereferenced item

Algorithms and Iterators

- An iterator object points to (refers to, denotes) an element of a container, e.g., a sequence
- The end of the sequence is “one past the last element”
  - *Not* “the last element”
  - That’s necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn’t an element
    - You can compare an iterator pointing to it
    - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”





Iterating Through a Vector

With and Without Iterators

```
ricegf@pluto:~/dev/cpp/201801/21$ make iteration
g++ -std=c++14    iteration.cpp   -o iteration
ricegf@pluto:~/dev/cpp/201801/21$ ./iteration
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
ricegf@pluto:~/dev/cpp/201801/21$
```

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
  vector<int> v = {1, 2, 3, 4, 5};

  // Old school - 3-term counted loop
  for(int i=0; i<v.size(); ++i) cout << v[i] << " ";

  // Verbose school (but more techically accurate) - 3-term counted loop
  for(vector<int>::size_type i=0; i<v.size(); ++i) cout << v[i] << " ";

  // Verbose school - 1-term for-each loop
  for(vector<int>::value_type i : v) cout << i << " ";

  // Classic school - 1-term for-each loop with known type
  for(int& i: v) cout << i << " ";

  // Auto school - 1-term for-each loop where the compiler figures out the type
  for(auto& i : v) cout << i << " ";

  // Explicit school - iterator is a nested class inside vector that returns
  // a pointer to a vector item (or the address past the last item v.end())
  for(vector<int>::iterator p = v.begin(); p!=v.end(); ++p) cout << *p << " ";
}
```

The 1-term for-each loops depend on iterators

Common Iterator Methods

- All iterators must provide:
  - copy constructor and destructor
  - operator=, operator++ and operator*
- Input iterators add:
  - operator== and operator!=
- Output iterators add:
  - Dereferencing as an lvalue (e.g., *x = t)
- Forward iterators add:
  - Default constructor
- Bidirectional iterators add:
  - operator--
- Random access iterators add:
  - operator+, operator-, operator< et al, operator+= and operator-=, operator[ ]

Containers that support iterators usually provide (x is item, p is iterator):

- begin() - returns p pointing to first item
- end() - returns p pointing one past last item
- front(), back() - first / last item
- size() - number of elements
- empty() - true if no elements
- push_back(x) / push_front(x) – insert at end / begin
- insert(p, x) – insert x immediately before p
- pop_back() / pop_front() – delete at end / begin
- erase(p) – remove item at p

These are <u>conventions</u> – the compiler doesn't care

Writing Iterators for Custom Classes

- Writing an iterator from scratch is fairly complex
  - But we usually don't need to write from scratch
- Four common scenarios
  - **Pointers** – A pointer is by definition a (non-class) iterator, so if one accesses your data, just return the pointer
  - **Nested container** (e.g., vector inside a wrapper class) – Just delegate to the nested container's iterator
  - **Iterator++** (e.g., a range-checked array accessor) – This also relies on delegation to the extended iterator, but can be tricky to implement given compiler limitations
  - **Written from scratch** – Hard, but rarely necessary

A Nested Container Example

- Let's produce an "order book" that lists orders for a sales associate in a store
  - The book will identify the sales associate's name, and also include a vector of orders
  - Iterating through the book should return the orders

```
#include <iostream>
#include <vector>
using namespace std;

class Order {
  public:
    Order(string customer, string model) : _customer{customer}, _model{model} { }
    string customer() {return _customer;}
    string model() {return _model;}
  private:
    string _customer;
    string _model;
};
```

An Order simply associates a customer with a model number

Defining an Order Book: A Nested Container Example

- In addition to the usual structure, we delegate 4 members to vector

```
class Order_book {
  private:
    string _sa;                      // Sales Associate who owns this book
    typedef vector<Order> Orders;
    Orders orders;                   // This SA's list of orders
  public:
    Order_book(string sales_associate) : _sa{sales_associate} { }
    string sales_associate() {return _sa;}
    void add_order(Order order) {orders.push_back(order); }
    typedef Orders::iterator iterator;
    typedef Orders::const_iterator const_iterator;
    iterator begin() {return orders.begin();}
    iterator end() {return orders.end();}
};
```

- That's all there is to it!

A Nested Container Example

- Our order book is now an enhanced vector (adding info on the sales associate) that still iterates
  - This is sometimes called “composite inheritance”

```
int main() {
  Order_book book{"Prof Rice"};
  book.add_order(Order{"Larry", "Crazy Cukes"});
  book.add_order(Order{"Curly", "Zany Zeros"});
  book.add_order(Order{"Moe", "Quick Quirks"});

  cout << "ORDER BOOK: " << book.sales_associate() << endl;
  for(Order o: book) {
    cout << o.customer() << " ordered " << o.model() << endl;
  }
}
```

```
ricegf@pluto:~/dev/cpp/201801/21$ make order_book
g++ -std=c++14    order_book.cpp   -o order_book
ricegf@pluto:~/dev/cpp/201801/21$ ./order_book
ORDER BOOK: Prof Rice
Larry ordered Crazy Cukes
Curly ordered Zany Zeros
Moe ordered Quick Quirks
ricegf@pluto:~/dev/cpp/201801/21$
```

Inheritance vs Composites

- Extending a class via composition often yields the same or similar benefits as inheritance
  - This has led to some debate: “Composition over Inheritance” vs “True Inheritance”
  - We can leave that to the theoreticians (until you become one) – use what works best per application
- With STL classes, you have no choice
  - **Never inherit from an STL class** – use composition
  - You can violate this rule after years of experience – but by then you probably won’t want to...

Class Constructors and Destructors

- C++ classes require (up to) 4 essential operations
  - Default and Non-default constructors (defaults to: nothing)
    - No default if any non-default constructors are declared
  - Copy constructor (defaults to: copy the member)
    - Needed if the class includes pointers or dynamic memory allocation
  - Copy assignment (defaults to: copy the members)
    - Used when an instance is to the left of the “=” operator
  - Destructor (defaults to: nothing)
    - Used to “clean up” e.g., dynamically allocated memory when an object is deleted (destroyed)

Constructors and destructors are described in detail in chapters 17-19

Review

Default Constructor Only

```
#include <iostream>
using namespace std;

// previous slide's code goes here

class Date {
  public:
    void set_date(int y, Month m, int d) {year=y; month=m; day=d;}
    void print_date() {cout << month_to_string(month) << " " <<
                        day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday;
    my_birthday.print_date();  // Oops – we haven't provided any data yet!
    my_birthday.set_date(1950, Month::dec, 30);  // OK
    my_birthday.print_date();  // Better
}
```

```
ricegf@pluto:~/dev/cpp/201701/16$ c++ -std=c++11 enum_class_io_no_constructor.cpp
ricegf@pluto:~/dev/cpp/201701/16$ ./a.out
Unknown 0, -1162798832
December 30, 1950
ricegf@pluto:~/dev/cpp/201701/16$
```

Review

Non-Default Constructors

```
#include <iostream>
using namespace std;

// previous slide's code goes here

class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
    Date() : Date(1970, Month::jan, 1) { }
    void set_date(int y, Month m, int d) {year=y; month=m; day=d;}
    void print_date() {cout << month_to_string(month) << " " <<
                        day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date unix_epoch;  // or unix_epoch{} - Default constructor
    unix_epoch.print_date();
    Date my_birthday{1950, Month::dec, 30};  // Non-default constructor
    my_birthday.print_date();
}
```

Multiple constructors (with or without a default) is fine and quite common

```
ricegf@pluto:~/dev/cpp/201801/21$ make date
g++ -std=c++14    date.cpp   -o date
ricegf@pluto:~/dev/cpp/201801/21$ ./date
January 1, 1970
December 30, 1950
ricegf@pluto:~/dev/cpp/201801/21$
```

Initialization vs Assignment

This is important for understanding copy constructors vs copy assignment

- Initialization causes *a new object* to have the same value as an existing object
  - `Robot r1 = r2; // initialization`
  - `Robot r1{r2};  // initialization – exactly the same`
- Assignment causes *an existing object* to have the same value as an existing object
  - `Robot r1;`
  - `r1 = r2; // assignment`
- These are two distinct operations in C++

Default Copy Constructors and Copy Assignment

```
ricegf@pluto:~/dev/cpp/201801/21$ make cc_and_cao2
g++ -std=c++14    cc_and_cao2.cpp    -o cc_and_cao2
ricegf@pluto:~/dev/cpp/201801/21$ ./cc_and_cao2
bar0 = 0
bar1 = 1
bar2 = 1
bar0 now = 1
ricegf@pluto:~/dev/cpp/201801/21$
```

```
#include <iostream>
using namespace std;

class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {} // Non-default constructor
    Foo() : Foo(0) {}           // Default constructor
    int val() {return _val;}
};

int main() {
  Foo bar0;            // Default constructor
  cout << "bar0 = " << bar0.val() << endl;
  Foo bar1{1};         // Non-default constructor
  cout << "bar1 = " << bar1.val() << endl;
  Foo bar2{bar1};      // Default copy constructor for initialization
  // Foo bar2 = bar1; // Exactly the same thing: bar2 has same values as bar1
  cout << "bar2 = " << bar2.val() << endl;
  bar0 = bar1;         // Default copy assignment for assignment
  cout << "bar0 now = " << bar0.val() << endl;
}
```

Copy Constructor

- For initialization, C++ actually invokes a special constructor – the copy constructor
  - If you have not specified one, you'll get the default – all of your variables will be copied directly across
- The copy constructor is thus invoked when:
  - You initialize a new object to an existing object
    - `Foo bar2 = bar1;  // Identical to Foo bar2{bar1};`
  - You pass an object as a non-reference parameter
    - `analyze(bar1); // A copy of bar1 is created for analyze`
  - You return an object from a function
    - `return bar1;  // A copy of bar1 is created and returned`

Declaring a Copy Constructor

- A copy constructor is just a constructor that accepts a const reference to the object to be copied
  - I *know* this isn't useful here – we'll come back to when writing a copy constructor is useful shortly

```
#include <iostream>
using namespace std;

class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                // Non-default constructor
    Foo() : Foo(0) {}                          // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}  // Copy constructor
    int val() {return _val;}
};
```

Copy Assignment

- For assignment, C++ invokes the assignment operator to overwrite the left-hand object's values
  - If you have not specified one, you'll get the default – all of your variables will be copied directly across
- The copy assignment is thus invoked when:
  - You assign to an existing object the value of another (*or the same*) existing object
    - `bar2 = bar1;  // bar2 was existing, so we overwrite it`

Declaring a Copy Assignment

- A copy assignment is a definition of the = operator to handle copying members as needed
  - Not useful here, either – almost there!

```
class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                      // Non-default constructor
    Foo() : Foo(0) {}                                // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}  // Copy constructor
    Foo& operator=(const Foo &rhs) {               // Copy assignment
      if (this != &rhs) _val = rhs.val();
      return *this;
    }
    int val() const {return _val;}
};
```

Destructors

- The destructor is invoked when the object itself is being deleted
  - *Really* useless here! Next slide, promise!

```
class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                    // Non-default constructor
    Foo() : Foo(0) {}                              // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}  // Copy constructor
    Foo& operator=(const Foo &rhs) {          // Copy assignment
      if (this != &rhs) _val = rhs.val();
      return *this;
    }
    ~Foo() {}                                      // Destructor
    int val() const {return _val;}
};
```

Practical Use and the Rule of Three

- So when would copy constructors, copy assignment, and destructors actualy be *useful*?
  - Typically, when your class allocates memory from the heap, keeping a pointer to its address
    - To copy the object, you also need to "deep copy" the allocated heap memory to another heap memory area
    - To assign the object, you need to deallocate the existing heap memory before doing the "deep copy" to allocate the new heap memory
    - To delete the object, you also need to delete the heap memory

Rule of Three: If you define <u>one</u> of the above,
you probably need to define <u>all three</u> of the above!

For more detail, I recommend the following paper. This isn't required for the exam.
http://www.keithschwarz.com/cs106l/winter20072008/handouts/170_Copy_Constructor_Assignment_Operator.pdf

Rule of 3 Implementation of Foo

- Allocating and deallocating private variables
- Copy and assignment allocate new heap
  - Rather than pointing to the *same* heap!

```
class Foo {
    int* _val;
  public:
    Foo(int* val) : _val{val} {}                            // Non-default constructor
    Foo() : Foo(new int{0}) {}                              // Default constructor
    Foo(const Foo &rhs) : _val{new int{*rhs.get()}} {} // Copy constructor
    Foo& operator=(const Foo &rhs) {                        // Copy assignment
      if (this != &rhs) _val = new int{*rhs.get()};
      return *this;
    }
    ~Foo() {delete _val;}                                   // Destructor
    int* get() const {return _val;}                         // Getter
    void set(int* v) {*_val = *v;}                          // Setter
};
```

Testing the Rule of 3

- Unfortunately, the C++ standard provides no way to determine if a pointer has been deleted, so...

```
int main() {
    Foo foo1{new int{42}};                                  // Non-default constructor
    // Test non-default constructor
    if (*foo1.get() != 42) std::cerr << "foo1 set to 42 but is now "
                                     << *foo1.get() << std::endl;

    Foo foo2{foo1};                                         // Copy constructor
    Foo foo3;                                               // Default constructor
    // Test default constructor
    if (*foo3.get() != 0) std::cerr << "foo3 default constructor set value to "
                                     << *foo3.get() << std::endl;
    foo3 = foo1;                                            // Copy assignment operator

    // This should NOT modify foo2 or foo3!
    int i = 17;
    foo1.set(&i);

    // Test copy constructor
    if (*foo2.get() != 42) std::cerr << "foo2 changed from 42 to "
                                     << *foo2.get() << std::endl;
    // Test copy assignment operator
    if (*foo3.get() != 42) std::cerr << "foo3 changed from 42 to "
                                     << *foo3.get() << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/201801/21$ make cc_and_caol
g++ -std=c++14    cc_and_caol.cpp   -o cc_and_caol
ricegf@pluto:~/dev/cpp/201801/21$ ./cc_and_caol
ricegf@pluto:~/dev/cpp/201801/21$
```

Quick Review

- The four basic types of STL containers in C++ are
  (a) Memory Managed Containers (b) Unmanaged Containers
  (c) Sequence Containers (d) Sequence Adapters
  (e) Associative Containers (f) Associative Adapters
  (g) Unordered Associative Containers
- _______ ________ are a façade for __________ __________,
- A(n) ________ is a pointer-like instance of a nested class used to access items managed by the outer class instance/
- True or False: With iterators, the end of the sequence is “one past the last element”, not “the last element”
- Two types of iterators are typically provided, ______ and _____ ________.
- Name some common iterator and container methods in the STL.
- True or False: It is a good practice to derive your classes from STL classes.
- What is a copy constructor? Copy assignment operator?
- Define the Rule of 3.

For Next Class

- (Optional) Read Chapters 20-21 in Stroustrop
  - Do the Drills!
- Skim Chapters 23-24 for next class
  - We’ll discuss text manipulation (including regular expressions in more detail) and numerics
- **Sprint #2 now in progress: Our first GUI!**