Rule of Three Review

Let’s remember

- If you can use STL containers / algorithms to solve your problem, do so
  - Containers handle their own memory
  - Algorithms are efficient, tested
- Altogether a better approach

But...

- It is useful to look “under the hood” to see how things work
- We will go through some class design where we do our own memory management on a container
- More work, must be careful

Rule of Three

- The **rule of three** is used for any object that dynamically allocates memory. In this case, you probably need
  - Copy constructor
  - Assignment operator
  - Destructor
- **Rule**: If you need one (really need one) then you need all three

Defaults are fine for non-dynamic memory

- You **do not have** to write any of these member operation
  - If you do not, C++ provides them for you (destructor, copy, assign)
  - If you define only one, C++ will define the other two (but remember the rule of three)
- Unless you are doing dynamic memory, you don't **need** this, but you can do it if there is a good reason

=delete

- Like `explicit` which controls when a conversion gets called you can set a method (like a copy) to be `=delete`, meaning it doesn’t exist and won’t run
- In this way you force the user to use either a reference or a pointer

Form of copy constructor

- `Stack::Stack(const Stack &s) {}`

- We know it is a copy constructor because:
  - It is a constructor
  - It takes as a parameter a reference to the same class

  - Why does it have to be a reference?
  - What does it return?

Form of assignment operator

- `Stack& Stack::operator=(const Stack &s) {}`

- Stack assignment operator:
  - Also takes a const Stack reference
  - Returns a Stack reference
    - Why return anything?

Member to member copy

- C++ by default does mostly the right thing: **member to member copy**
  - For each data member in the class, a copy is made (calling the copy constructor of that class) to make a copy
  - Except for pointers (copy of a pointer may not be what you want) that is usually good enough

Copy and assign do much the same thing

- If you want to control how things get copied, then we probably want to control how things get assigned
  - They pretty much do the same thing
  - They could be exactly the same except for chaining behavior of assign

Composite Class

Stacks using vectors

- Composite class using vectors
- Example 19.1

Last in, First out

- Basic stack operations
  - pop (top element off)
  - top (value of top element)
  - push (new top element)
  - empty / full (boolean)
  - clear (remove all elements)

Empty Stack

- Choice of what to do when we pop / top on an empty stack
  - Could return a “sentinel” value
    - Bad, what should it be in a templated class?
  - Could create our own error type
    - Best, but beyond us at this point
  - Could throw an existing error
    - Compromise, simpler to do but error is not tied to this class

```
class Stack {
private:
	vector<char> vec_;
public:
	Stack() = default;
	Stack(initializer_list<char> c) :
vec_(c) {};
	char top();
	void pop();
	void push(char);
	bool empty();
	// bool full();
	void clear();
	friend ostream& operator<<
		(ostream&, const Stack&);
}
ostream& operator<<(ostream&, const
Stack&);
```

```
char Stack::top() {
	if (vec_.empty())
		throw underflow_error("top, empty stack");
	return vec_.back();
}
void Stack::push(char s) {
	vec_.push_back();
}
void Stack::pop() {
	if (vec_.empty())
		throw underflow_error("pop, "empty stack");
	vec_.pop_back();
}
bool Stack::empty() {
	return vec_.empty();
}
void Stack::clear() {
	vec_.clear();
}
```

Composite Class

- This is a “composite class” a class built by using the operations of other classes in the implementation
- Inheritance in CSE 335 is another way to achieve the same effect
- There are pluses and minuses to each

Bad Dynamic Memory Class

Stack via dynamic memory

- Doing it by yourself
- Example 19.2

Remember =default

- This says that we are being “clear” we are going to take the default behavior
  - However, we don’t have to say it, because if we don’t say anything that is what it will do
  - But, it is good to be clear

```
class Stack {
private:
    char *ary_;
    size_t sz_;
    size_t top_;
public:
    explicit Stack(size_t sz = 10);
    Stack(const Stack&) = default;
    Stack& operator=(const Stack&) = default;
    ~Stack() = default;
    char top();
    void pop();
    void push(char);
    bool empty();
    bool full();
    friend ostream& operator<< (ostream&, const Stack&);
}
```

Allocate a fixed-size array
Take the default on copy, assign, destroy
Terrible idea!

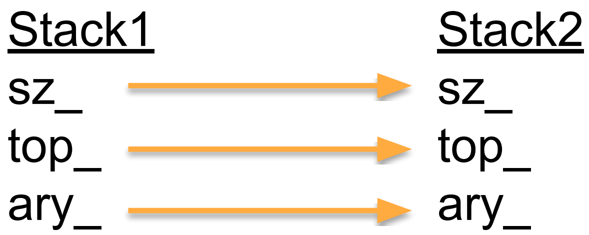
```
Stack::Stack(size_T sz) {
    ary_ = new char[sz]();
    sz_ = sz;
    top_ = 0;
}
char Stack::top() {
    return ary_[top_ - 1];
}
void Stack::push(char element) {
    ary_[top_++] = element;
}
void Stack::pop() {
    top_--;
}
```

```
bool Stack::empty() {
    return top_ == 0;
}
bool Stack::full() {
    return top_ == sz_;
}
```

Took the default on the “three”

- Think about what should happen now under a copy scenario
  - `sz_` gets copied to the new object
  - `top_` gets copied to the new object
  - `ary_` gets copied to the new object
    - What does that mean?
    - What type is `ary_`?

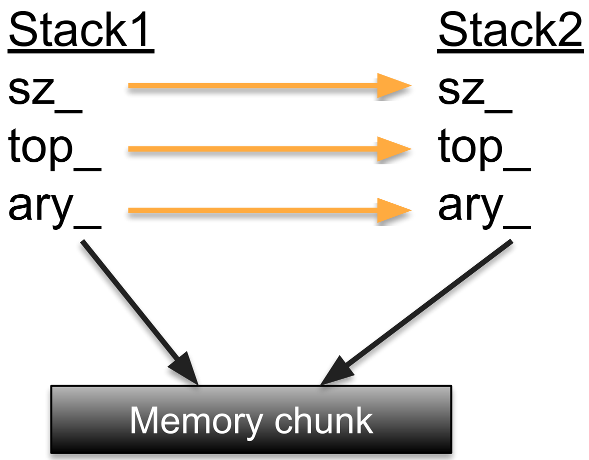
Copying



ary_ is a pointer

- What do you get when you copy one pointer to another?
- You get two pointers that point to the same memory location!
- Oops

What kind of copy is that?



Stack2 is now a copy of Stack1
except that both `ary_` point to the same memory

Very bad!

Fixing the Dynamic Memory

Repaired dynamic stack

- Let’s fix that pointer problem
- Example 19.3

Copy constructor

```
Stack::Stack(const Stack &s) {
    sz_ = s.sz_;
    top_ = s.top_;
    ary_ = new char[s.sz_];
    std::copy(s.ary_, s.ary_+s.sz_, ary_);
}
```

- pass by reference
- Copy over the built-in types
- Allocate new memory
- Copy contents of argument stack to the newly created stack

Destructor

```
Stack::~Stack() {
  delete [] ary_;
}
```

- Not good enough to just remove each member. Your object will leak!
  - If you `new` dynamic memory then you have to `delete` it as well
  - Destructor called when the object goes out of scope (or the like)

Assignment

- Assignment is very like copy, so there is like some code we can carve out as one
- There are some issues however
  - In assignment, the lhs has a pointer to dynamic memory. We have to delete that to avoid leaks
  - We have a use-case that could be a problem: self-assignment

A way, not the best

```
Stack& Stack::operator=(const Stack& s) {
    if (this != &s) {
        delete [] ary_;
        sz_ = s.sz_;
        top_ = s.top_;
        ary_ = new char[s.sz_];
        copy(s.ary_, s.ary_ + sz_, ary_);
    }
    return *this;
}
```

Why this?

Clean memory

Repeat of copy

Copy and Swap Idiom

Copy and swap, better assignment

- Example 19.4

Not modular

- Each element should do one job and we should reuse that. This op= does not
  - Copy constructor should do the copy, write it once and use it
  - Destructor should delete the memory, write it once and use it

swap function, friend

```
void swap(Stack &s1, Stack &s2) {
    std::swap(s1.top_, s2.top_);
    std::swap(s1.sz_, s2.sz_);
    std::swap(s1.ary_, s2.ary_);
}
```

This is a pointer swap
Is this ok?

Specific to Stacks based on the arguments
Want to use std::swap (a library function) inside to do the actual movement of members

Copy-and-swap idiom

- This is very nice, makes everyone do their one job.

```
Stack& Stack::operator=(Stack rhs) {
    swap(*this, rhs);
    return *this;
}
```

This is a copy (not a ref)!
Call to copy constructor
Destructor called when scope ends

Swaps the members of the rhs into the newly assigned lhs
Swap is efficient, swaps the members
The pointers are swapped, not the memory
rhs is a copy to be destroyed

Setup, do assignment

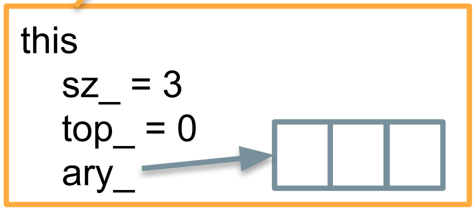
```
Stack stk1(3);
stk1.push('a');
stk1.push('b');
Stack stk2(3);
stk2 = stk1;
```

Inside the call

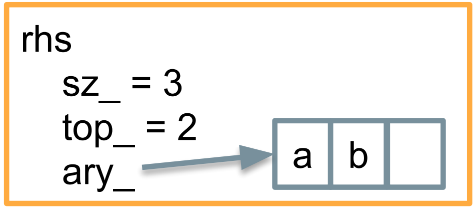
```
Stack stk2(3);
stk2 = stk;
```

Call the copy ctr

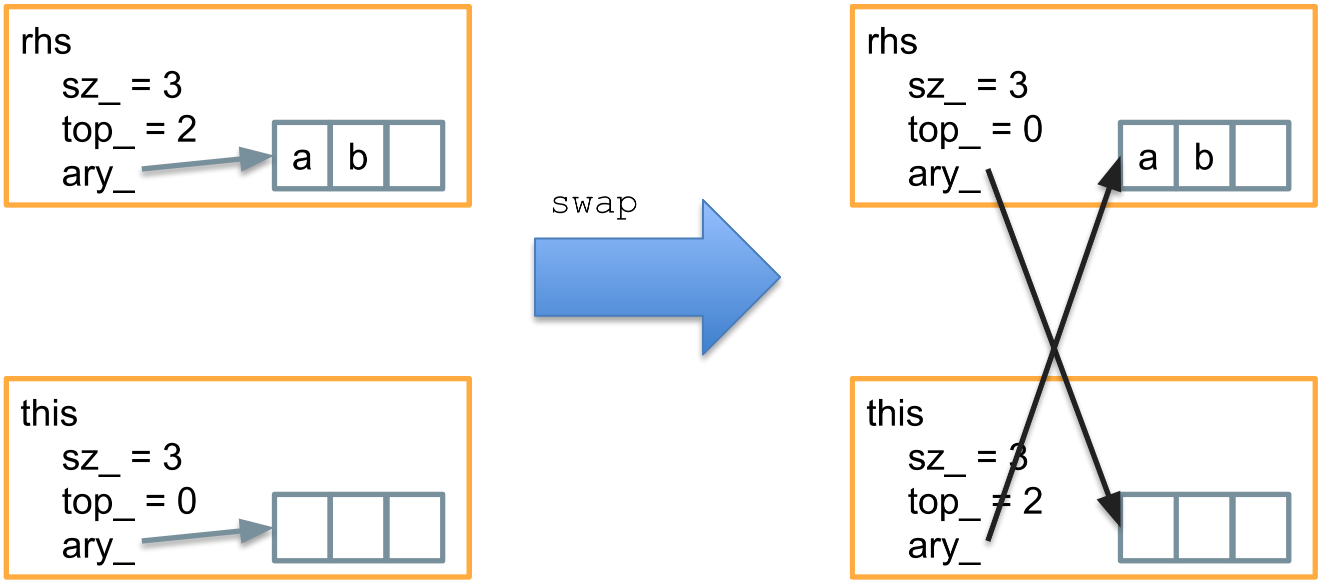
```
Stack& Stack::operator=(Stack rhs)
```



Copy of stk1



The swap



rhs
sz_ = 3
top_ = 2
ary_
a
b
this
sz_ = 3
top_ = 0
ary_
swap
rhs
sz_ = 3
top_ = 0
ary_
a
b
this
sz_ = 3
top_ = 2
ary_

Templates and Classes

Remember template functions?

```
template<typename my_type>
void swap(my_type &first, my_type &second) {
    my_type temp;
    temp = first;
    first = second;
    second = temp;
}
```

Generic function

- By writing the function as a template we can write a *generic function*
  - A function which, even in C++ (which is very type strict) is generic **for all types**
- Remember: a template is a pattern to make a function, It is not a function

Force the type

- Typically the compiler deduces the type for substitution in the template from the provided arguments
- You can force (though you must be careful) the type used, but it has to work with the args and the created function

Picking the template types

- Invocation

```
long i = 1, j = 2;
swap<double>(i, j);
```

template type directly indicated

Will see this again and again. We specify in the invocation the type we want used in the template

Templates and Classes

- Composite template of stack
- Example 19.5

Templated Class

- It is inconvenient to write a container that can only store one type
  - Stack of longs
  - Stack of ints
  - Stack of chars
- Better if we capture what the class Stack doesn't allow the type to vary (just like functions)

Same line as with functions

- `template <typename TemplateVar>`
- However, what will be different is that we have to **force** the selection of type as we would with vectors, maps, etc.

Put template var where type would go

- We write the template using the template variable everywhere we would normally put the actual type being used
- Eventually, the template variable will be replaced with an actual type

Templated Stack

```
template<typename T>
class Stack {
private:
    vector<T> vec_;
public:
    T top();
    void pop();
    void push(T);
    bool empty();
    friend ostream& operator<<(ostream& out, const Stack<T> &s);
};
```

Template variable

In the call, set the template

```
Stack<char> stk_c;
Stack<long> stk_l;
```

- As with functions, we say in `< >` what type we work with and the template engine substitutes the given type with the template variable, making the new class.

Instantiate new class

sub char for T

Stack requires a template type

```
template<typename T>
class Stack {
private:
    vector<T> vec_;
public:
    T top();
…
```

```
Stack<char> s;
```

Create the actual class

```
class Stack {
private:
    vector<char> vec_;
public:
    char top();
…
```

A template is not a class

- A template is a way to make a class where the type is “independent”
  - By substitution, we can create many versions of the class, each with the template type set to a particular actual type
- A template is not a class, it is a pattern!

Each member templated

```
template<typename T>
T Stack<T>::top() {
    return vec_[vec_.size() - 1];
}
```

templated header one for each member

Return type

templated stack

No .cpp file for a template

- Everything in a templated container goes in the header. No .cpp file
- This is because all of the code needs to be in one place so that the appropriate substitution can occur to create the actual class from the template.

```
template<typename T>
class Stack {
private:
    vector<T> vec_;
public:
    // Take defaults.  Vector handles it already
    Stack()=default;
    Stack(const Stack &s)=default;
    Stack& operator=(const Stack &s)=default;
    ~Stack()=default;
    // Stack operations
    T top();
    void pop();
    void push(T);
    bool empty();
    // friends are special, see following slides
};
```

No templates of class in class def

- You will note that you do not need to provide the template vars in the class definition itself for the class.
- Inside the class template, **and only there**, the compiler treats a class reference as if it were templated

Instantiation of member functions as needed

- Remember, a template is not a class. It is a pattern to instantiate a class
- Thus each member function is only instantiated as needed (when used in a calling program somewhere).

Templated Friends

The problem

- There is a problem matching up the template of the friend function with the templating of the class
- There are two ways to do it, easy and hard

Easy

- Do the friend **inline** in the class declaration
  - In this way, the template substitution gets done correctly
  - Get a new friend for every template instantiation

Hard

- (read the book if you like, pg 664)

```
template <typename T>
class Stack;

template <typename T>
ostream& operator<<(ostream&, const Stack<T> &);

friend ostream& operator<< <T> (ostream& out, const Stack& s);
```

- Now you can write the actual function (as always)

Steps

- Forward declare that your class is a template
- Forward declare the friend object with template info
- In the class, force the function type the friend will use (after the friend function name)

One-to-one type friends

- By declaring this way, we get

```
Stack<long> stk; // has operator<< <long> as a friend
Stack<char> stk; // has operator<< <char> as a friend
```

Templated, dynamic memory stack

- Example 19.6
- The full monty