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
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(size_t sz)
    Stack(initializer_list<char> c) : vec_(c){};

    char top();
    void pop();
    void push(char);
    bool empty();  // not a problem with vectors
    void clear();

    friend ostream& operator<<(ostream &out, Stack const &s);
};

char Stack::top() {
    if (vec_.size() == 0)
        throw underflow_error("top, empty stack");
    return vec_.back();
}

void Stack::pop() {
    if (vec_.size() == 0)
        throw underflow_error("pop, empty stack");
    vec_.pop_back();
}

void Stack::push(char s) { vec_.push_back(s); }

bool Stack::empty() { return vec_.empty(); }

void Stack::clear() { vec_.clear(); }

ostream &operator<<(ostream &out, Stack const &s) {
    out << "(bottom) ";
    copy(s.vec_.begin(), s.vec_.end(), ostream_iterator<char>(out, ",");
    out << " (top)";
    return out;
};
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 BadStack {
private:
    char *ary_ = nullptr;
    int sz_ = 0;
    int top_ = -1;
public:
    BadStack() = default;
    explicit BadStack(size_t sz);
    BadStack(initializer_list<char> c);
    BadStack(BadStack const &) = default;
    BadStack &operator=(BadStack const &) = default;
    ~BadStack() = default;
    char top();
    void pop();
    void push(char);
    bool empty();
    bool full();
    void clear();

    friend ostream &operator<<(ostream &, BadStack const &);
};
ostream &operator<<(ostream &out, BadStack const &s);
BadStack::BadStack(size_t sz) {
    sz_ = sz;
    ary_ = new char[sz]{};
    top_ = -1;
}

char BadStack::top() {
    if (top_ < 0) throw underflow_error("top, empty stack");
    return ary_[top_];
}

void BadStack::pop() {
    if (top_ < 0) throw underflow_error("pop, empty stack");
    --top_;
}

void BadStack::push(char element) {
    if (top_ >= (sz_ - 1)) throw overflow_error("push, full stack");
    ary_[++top_] = element;
}

bool BadStack::empty() { return top_ < 0; }

bool BadStack::full() { return top_ >= (sz_ - 1); }
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_?
Stack1
sz_
top_
ary_

Stack2
sz_
top_
ary_
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(Stack const &s) {
    sz_ = s.sz_;  
    top_ = s.top_;  
    ary_ = new char[s.sz_];  
    // ary_ = s.ary_   BAD IDEA, just copies the pointers!!!  
    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

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

```cpp
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.

```cpp
Stack &Stack::operator=(Stack s) {
    swap(*this, s);
    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)

this
sz_ = 3
top_ = 0
ary_

Copy of stk1
rhs
sz_ = 3
top_ = 2
ary_

a
b
The swap

```
<table>
<thead>
<tr>
<th>rhs</th>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz_ = 3</td>
<td>sz_ = 3</td>
</tr>
<tr>
<td>top_ = 2</td>
<td>top_ = 0</td>
</tr>
<tr>
<td>ary_</td>
<td>ary_</td>
</tr>
<tr>
<td>a b</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>rhs</th>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz_ = 3</td>
<td>sz_ = 3</td>
</tr>
<tr>
<td>top_ = 0</td>
<td>top_ = 2</td>
</tr>
<tr>
<td>ary_</td>
<td>ary_</td>
</tr>
<tr>
<td>a b</td>
<td></td>
</tr>
</tbody>
</table>
```
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.
Invocation

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

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 strings
  - 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
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);
};
In the call, set the template

Stack<char> stk_c;
Stack<int> stk_i;

- 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

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

Create the actual class

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

Stack requires a template type

```cpp
Stack<char> s;
```
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!
template<typename T>

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

Each member templated

Return type

templated stack

templated header one for each member
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:
    Stack() = default;
    Stack(initializer_list<T> lst) : vec_(lst){};
    Stack() = default;
    Stack(Stack const &) = default;
    Stack &operator=(Stack const &) = default;
    ~Stack() = default;

    T top();
    void pop();
    void push(T);
    bool empty();

};
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)

```cpp
template <typename T>
class Stack;

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

friend ostream& operator<< <T> (ostream& out, Stack const & 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<int> stk;  // has operator<< <int> as a friend
  Stack<char> stk; // has operator<< <char> as a friend
Templated, dynamic memory stack

- Example 19.6
- The full monty