**A potpourri of class stuff**

dynamic memory, copy constructor, assignment, destructor, template classes

---

**Let's remember**

The rule should be:
- if you can use STL containers/algorithms to solve your problems, do so.
  - containers handle their own memory
  - algorithms are efficient, tested

Altogether a better approach

---

**but...**

It is useful to look under the "software covers" and see how things work.

We will go through some class design where we do our own memory management on a container.

More work, we must be careful

---

**copy/assign**
The **rule of three** is used for any object that dynamically allocates memory. In this case you probably:

- define a copy constructor
- define an assign
- define a destructor

**Rule:** if you need one (really need one), then you really need all three!

---

You do not have to write any of these member operations:

- 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 does not exist and won’t run.

In this way you force the user to use either a reference of a pointer.

---

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

We know this 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 assign operator

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

Stack assignment operator:
- also takes a const Stack reference
- returns a Stack reference
- why return anything?

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 if it is STL) to make a copy.
- except for pointers (copy of a pointer may not be what you want) that is good enough!

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 if it is STL) to make a copy.
- except for pointers (copy of a pointer may not be what you want) that is good enough!

Copy and assign do much the same thing
If we 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

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Copy_Constructors
- Implicit copying of objects in C++ is a rich source of bugs and of performance problems. It also reduces readability, as it becomes hard to track which objects are being passed around by value as opposed to by reference, and therefore where changes to an object are reflected.
- Decision:
  - Few classes need to be copyable. Most should have neither a copy constructor nor an assignment operator. In many situations, a pointer or reference will work just as well as a copied value, with better performance. For example, you can pass function parameters by reference or pointer instead of by value, and you can store pointers rather than objects in an STL container.
  - If your class needs to be copyable, prefer providing a copy method, such as CopyFrom() orClone(), rather than a copy constructor, because such methods cannot be invoked implicitly. If a copy method is insufficient in your situation (e.g., for performance reasons, or because your class needs to be stored by value in an STL container), provide both a copy constructor and assignment operator.

Better alternative
- Implicit copying of objects in C++ is a rich source of bugs and of performance problems. It also reduces readability, as it becomes hard to track which objects are being passed around by value as opposed to by reference, and therefore where changes to an object are reflected.
- Decision:
  - Few classes need to be copyable. Most should have neither a copy constructor nor an assignment operator. In many situations, a pointer or reference will work just as well as a copied value, with better performance. For example, you can pass function parameters by reference or pointer instead of by value, and you can store pointers rather than objects in an STL container.
  - If your class needs to be copyable, prefer providing a copy method, such as CopyFrom() orClone(), rather than a copy constructor, because such methods cannot be invoked implicitly. If a copy method is insufficient in your situation (e.g., for performance reasons, or because your class needs to be stored by value in an STL container), provide both a copy constructor and assignment operator.
**stack using vectors**

Composite class using vectors

Ex 16.1

---

**empty stack**

choice on what to do when we pop/top on an empty stack:

- could return "sentinel value:
  - bad, what should it be with template?
- 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 the class

---

```cpp
class Stack{
private:
    vector<char> vec_; // basic Stack operations:
    string top();
    void pop();
    void push(char s);
    bool empty();
    void clear();
public:
    Stack(); // constructor
    Stack(initializer_list<char> c) :
        vec_(c) {};
    char top();
    void pop(char s);
    void push(char s);
    // bool full(); // not directly accessible
    friend ostream& operator<<(ostream&*out,
        const Stack &s); // not directly accessible
    ostream& operator<<(ostream&out,
        const Stack &s); // not directly accessible
};
```

---

**Last In, First out (LIFO)**

Basic Stack operations:
- pop (top element off)
- top (value of top element)
- push (new top element)
- empty, full (Booleans)
- clear (remove all elements)
- ...

---

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

```cpp
void Stack::pop(char s){
    vec_.push_back(s);
}
```

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

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

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

```cpp
ostream& operator<<(ostream&out,
    const Stack &s){
    out << "(bottom)";
    copy(s.vec_.begin(), s.vec_.end(),
        ostream_iterator<char>(out, ","));
}
```

```cpp
ostream& operator<<(ostream &out,
    const Stack &s){
    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.

There are plusses and minuses to each.

```
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& out, const Stack &s);
};
```

Important to note:
- Allocate a fixed size array
- Take the default on copy, assign, destroy
  - **terrible idea!!!**

This says that we are being "clear", we are going to take the default behavior:
- We didn't have to say that, because if we don't say anything that is what it will do.
- Good to be clear
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_;}
ostream& operator<< (ostream &out, const Stack &s){
    out << "(bottom)";
    copy(s.ary_, s.ary_ + s.sz_,
        ostream_iterator<char>(out, ""));
    out << "Top is:" << s.top_;
    return out;
}

Stack stk1(5);
Stack stk2(stk1); // copy call

member by
copy
stk1
sz_
stk2
sz_
top_
top_
ary_
ary_

what type is ary_
what kind of copy is that?

stk1
sz_
top_
ary_

stk2
sz_
top_
ary_

memory hunk

So stk2 is a copy of stk1, except that both ary_ pointers now point to the same memory!

Very, very bad!

Repaired dynamic stack

Let's fix that pointer problem

Ex 16.3

copy constructor

Stack::Stack(const Stack &s){
    sz_ = s.sz_;  
    top_ = s.top_; 
    ary_ = new char[s.sz_];
    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.

remember, new object had no contents. Constructor!

destructor

Stack::~Stack(){
    cout << "Called dtor"<<endl;
    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 an object goes out of scope (or the like).
**Assignment**

- Assignment is very like copy, so there is likely 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.

```cpp
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_+s.sz_, ary_);
    }
    return *this;
}
```

Why this? Clean memory

**Copy and Swap, better assignment**

Ex 16.4

**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.
copy-and-swap idiom

This is very nice, makes everyone do their one job. Look carefully!

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

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

write a swap function that swaps the members of the copied rhs into the newly assigned lhs
- swap is efficient, swaps the members
  - if pointers, the pointers are swapped, not the memory
- rhs is a copy to be destroyed

This is a copy (not a ref)!
- call to copy constructor
- local scope so destructor called when scope ends

setup, do assignment

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

specific to Stacks, based on the arguments
want to use std::swap (a library function) inside
to do the actual movement of members.


top_ \rightarrow 0
ary_ \rightarrow \text{[image]}

\text{stk2}
\text{sz} \leftarrow 3
\text{top} \leftarrow 0
ary_ \rightarrow \text{[image]}

\text{stk}
\text{sz} \leftarrow 3
\text{top} \leftarrow 2
ary_ \rightarrow \text{[image]}

\text{stk2}
\text{sz} \leftarrow 3
\text{top} \leftarrow 0
ary_ \rightarrow \text{[image]}

\text{stk}
\text{sz} \leftarrow 3
\text{top} \leftarrow 2
ary_ \rightarrow \text{[image]}

\text{stk2}
\text{sz} \leftarrow 3
\text{top} \leftarrow 0
ary_ \rightarrow \text{[image]}

\text{stk}
\text{sz} \leftarrow 3
\text{top} \leftarrow 2
ary_ \rightarrow \text{[image]}
inside the call

Stack stk2(3)
stk2 = stk;
call the
operator=(Stack rhs)
copy of stk

Stack stk(3)
th is
* this
sz_ \leftarrow 3
top_ \leftarrow 0
ary_ \rightarrow a b
copy of stk

rhs
sz_ \leftarrow 3
top_ \leftarrow 2
ary_ \rightarrow a b
call the
operator=(Stack rhs)
copy of stk

Stack stk(3)
stk2 = stk;

Stack& Stack::operator=(Stack rhs)
copy of stk

* this
sz_ \leftarrow 3
top_ \leftarrow 0
ary_ \rightarrow a b
copy of stk

rhs
sz_ \leftarrow 3
top_ \leftarrow 2
ary_ \rightarrow a b
copy of stk

the swap

stack stk2(stk)

Stack stk(3)
stk2 = stk;

Stack& Stack::operator=(Stack rhs)
copy of stk

* this
sz_ \leftarrow 3
top_ \leftarrow 0
ary_ \rightarrow a b
copy of stk

rhs
sz_ \leftarrow 3
top_ \leftarrow 2
ary_ \rightarrow a b
copy of stk

* this
sz_ \leftarrow 3
top_ \leftarrow 0
ary_ \rightarrow a b
copy of stk

what gets destroyed?

* this
sz_ \leftarrow 3
top_ \leftarrow 2
ary_ \rightarrow a b

the memory of
the copy
destroyed. Yes!

templated classes
By writing the function as a template, we can write a *generic function*:

- a function which, even in C++ (which is type crazy), is generic for all types.

Remember: a template is a pattern to make a function. It is not a function.

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++
double result;
long i=1, j=2;
result = swap<double>(i, j);
```
Templated class
Composite template of stack
Ex 16.5

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 does, allow the type to vary (just like functions)

Templated Class

We write the template using the template variable everywhere we would normally put the actual type being used.

However, what will be different is that we have to force the selection of type, as we would with vectors, maps, etc.

Eventually, the template variable will be replaced with an actual type.
template <typename ElementType>
class Stack{
    private:
        vector<ElementType> vec;
    public:
        // stack operations
        ElementType top();
        void pop();
        void push(ElementType);
        bool empty();
        friend ostream& operator<<(ostream& out, const Stack<ElementType> &s);
};

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.

In the call, set the template
Stack<char> stk_c;
Stack<long> stk_l;

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

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

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 ElementType>
class Stack{
private:
   vector<ElementType> vec_;
public:
   // take defaults for the following 4. Vector handles it already.
   Stack()=default;
   Stack(const Stack&)=default;
   Stack& operator=(const Stack &s)=default;
   ~Stack()=default;
   // stack operations
   ElementType top();
   void pop();
   void push(ElementType);
   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 fns 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

Still Ex 16.5

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
• does inline substitution which may be problematic
hard

(Read the book if you like, pg 664).

```
template <typename ElementType>
    class Stack;

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

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

Now you can write the actual function (as always)

---

steps (FYI, not for exams)

1. forward declare that your class is a template
2. forward declare the friend object (with template info)
3. in the class, force the function template type the friend will use (after the friend fn 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

Ex 16.6
The full monty
template<typename ElementType>
    class Stack;

template<typename ElementType>
    class Stack{
    private:
        ElementType *ary_; // max size
        size_t sz_;        // where the next element is pushed
    public:
        Stack(size_t sz=10); // copy
        Stack(const Stack &s); // destructor
        ElementType top();
        void push(ElementType); // full()
        friend ostream& operator<<(ostream&, const Stack&);
    };

template<typename ElementType>
    Stack<ElementType>::Stack(size_t sz){
        ary_ = new ElementType[sz]();
        sz_ = sz;
        top_ = 0;
    }

template<typename ElementType>
    Stack<ElementType>::Stack(const Stack &s){
        sz_ = s.sz;
        top_ = s.top;
        ary_ = new ElementType[s.sz];
        copy(s.ary_, s.ary+s.sz_, ary_);
    }

template<typename ElementType>
    Stack<ElementType>& Stack<ElementType>::operator=(Stack &s){
        // swap & copy just do the swap individually here
        swap(this->sz_, s.sz_);
        swap(this->top_, s.top_);
        swap(this->ary_, s.ary_);
        return *this;
    }

    Stack<ElementType>::~Stack(){
        delete [] ary_;}

template<typename ElementType>
    ElementType Stack<ElementType>::top(){
        return ary_[top_-1];
    }

template<typename ElementType>
    void Stack<ElementType>::pop(){
        top_--;
    }

    bool Stack<ElementType>::empty(){
        return top_==0;
    }

    bool Stack<ElementType>::full(){
        return top_==sz;
    }
template <typename ElementType>
ostream& operator<<(ostream &out,
    const Stack<ElementType> &s){
    out << "(bottom)";
    copy(s.ary_, (s.ary_+s.sz_),
        ostream_iterator<ElementType>(out, ",");
    out <<": Top is:"<<s.top_;
    return out;
}