

COPYING & MOVING

LECTURE 10-3

JIM FIX, REED COLLEGE CS2-S20

LOGISTICS

▶ Office hours:

- Mine: Tues 3-4pm, Weds 11am-12pm, Thurs & Fri 12-1pm
 - Tutors: Tues 4-7pm, Weds 12-2pm, Thurs 1-4pm, Fri 1-3pm
 - Each day is a contiguous block of office hours. One Zoom link for each day.
- ▶ "Download/upload" **midterm on circuits & MIPS** next Thurs. Due Fri.
- I will post a practice exam this weekend.
- ▶ **Homework 10** will be assigned this weekend.
- on subclassing and templates

TODAY'S PLAN

- ▶ A BUG IN A SIMPLE MOD OF MY STACK-BASED **dc**
- ▶ EXAMINE TWO TEST PROGRAMS:
 - A SIMPLE CLASS WITH A VALUE MEMBER
 - A SIMPLE CLASS WITH A HEAP-ALLOCATED MEMBER
- ▶ LOOK AT:
 - COPY CONSTRUCTOR, COPY ASSIGNMENT
 - MOVE CONSTRUCTOR, MOVE ASSIGNMENT

SMALL CHANGE

- Let's make a small change to my stack-based calculator **dc.c**

```
void output_top(Stck s) {  
    if (!s.is_empty()) {  
        std::cout << s.top() << std::endl;  
    }  
}
```

```
int main() {  
    ...  
    Stck s {100};  
    std::string entry;  
    do {  
        output_top(s);  
        // parse and handle entry  
        ...  
    } while (entry != q);  
}
```

SMALL CHANGE

- Let's make a small change to my stack-based calculator **dc.c**

```
void output_top(Stck s) {  
    if (!s.is_empty()) {  
        std::cout << s.top() << std::endl;  
    }  
}
```

```
int main() {  
    ...  
    Stck s {100};  
    std::string entry;  
    do {  
        output_top(s);  
        // parse and handle entry  
        ...  
    } while (entry != q);  
}
```

SMALL CHANGE

- ▶ Let's make a small change to my stack-based calculator **dc.c**

```
void output_top(Stck s) {  
    if (!s.is_empty()) {  
        std::cout << s.top() << std::endl;  
    }  
}
```

```
int main() {  
    ...  
    Stck s {100};  
    std::string entry;  
    do {  
        output_top(s);  
        // parse and handle entry  
        ...  
    } while (entry != q);  
}
```

A MYSTERY

- Here is what happens when I recompile and run it...

```
$ ./dc
You've just run my version of the Unix calculator utility
'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ The second will instead have an instance variable of type **int***

```
class R {  
private:  
    int* a;  
public:  
    R(void);  
    R(int x0);  
    ~R(void);  
    friend R operator+(int i, R&& r);  
}
```


TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ They each can get **built two ways**:

```
class V {  
private:  
    int x;  
public:  
    V(void) : x {0} { };  
    V(int x0) : x {x0} { };  
    ~V(void);  
    friend V operator+(int i, V&& v);  
}
```

TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ They each can get **built two ways**:

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void);  
    friend R operator+(int i, R&& r);  
}
```

- ▶ But we'll look at destructors, copying, moving.

TWO SAMPLE CLASSES

- ▶ Today we examine some trickier aspects of C++ storage management.
- ▶ We'll reference two simple class definitions.
- ▶ They each can get **built two ways**:

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void);  
    friend R operator+(int i, R&& r);  
}
```

- ▶ But we'll look at destructors, copying, *moving*.

TRACKING CONSTRUCTION

- *In the sample folder*, have a second version of each that also store an ID.

```
class V {  
private:  
    static int next_id;  
    int id;  
    int x;  
    void give_id(void) { id = ++next_id; }  
public:  
    V(void) : x {0} { give_id(); };  
    V(int x0) : x {x0} { give_id(); };  
    ~V(void);  
    friend V operator+(int i, V&& v);  
}  
  
int V::next_id = 0;
```

- We'll do this to track what's going on.

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ Here This second line below gives its standard use:

```
V v1 {42}; // This calls the V(int) constructor.  
V v2 {v1}; // This calls the copy constructor.
```

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ Here This second line below gives its standard use:

```
V v1 {42}; // This calls the V(int) constructor.  
V v2 {v1}; // This calls the copy constructor.
```

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

it must match this signature

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ Here This second line below gives its standard use:

```
V v1 {42}; // This calls the V(int) constructor.  
V v2 {v1}; // This calls the copy constructor.
```


THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ **NOTE:** the copy construct gets applied in several other situations:
 - When a function is passed a **V** parameter *by value*

THE COPY CONSTRUCTOR

- ▶ A copy constructor is one that is used to construct an instance from another.
- ▶ Here is an example for the "value class" **V**:

```
V(const V& ov) : x {ov.x} { }
```

- ▶ Here we are simply copying the contents of another **V** instance **ov**
- ▶ **NOTE:** the copy construct gets applied in several other situations:
 - When a function is passed a **V** parameter *by value*
 - When a function returns a **V** by value

COPY CONSTRUCTOR APPLICATIONS

- ▶ When a **V** is constructed using a **V**:

```
V v2 {v1};
```

- ▶ When a function is passed a **V** parameter by value:

```
int get_value(V v) { ... }  
...  
int i = get_value(v1);
```

- ▶ When a function returns a **V** by value:

```
V get_V(...) {  
    V my_v;  
    ...  
    return my_v;  
}  
...  
V their_v = get_V(...);
```

- ▶ When an assignment is actually a **V** initialization:

```
V v2 = V {v1};
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member component is *copy assignment*
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ It gets used most times that there is a **V** assignment:

```
V v1 {42};  
V v2 {87};  
...  
v2 = v1;
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member component is *copy assignment*
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ It gets used most times that there is a **V** assignment:

```
V v1 {42};  
V v2 {87};  
...  
v2 = v1;
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member component is *copy assignment*
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ It gets used most times that there is a **V** assignment:

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
v3 = v2 = v1;
```

- ▶ It has this weird signature returning the assigned object as a reference because some C programmers like to **chain assignments**.

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member component is *copy assignment*
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ There are cases where it might not get used...

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
V v4 = V {v3}; // This, we saw, uses the copy constructor.  
V v5 = V {101}; // This will use the V(int) constructor.  
v3 = V {789}; // And this will use move assignment
```

THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member component is *copy assignment*
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ There are cases where it might not get used...

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
V v4 = V {v3}; // This, we saw, uses the copy constructor.  
V v5 = V {101}; // This will use the V(int) constructor.  
v3 = V {789}; // And this will use move assignment
```


THE COPY ASSIGNMENT OPERATOR

- ▶ A similarly behaving member component is *copy assignment*
- ▶ Here is an example for the "value class" **V**:

```
V& operator=(const V& ov) { x = ov.x; return *this; }
```

- ▶ There are cases where it might not get used...

```
V v1 {42};  
V v2 {87};  
V v3 {99};  
...  
V v4 = V {v3}; // This, we saw, uses the copy constructor.  
V v5 = V {101}; // This will use the V(int) constructor.  
v3 = V {789}; // And this will use move assignment
```

- ▶ WHY? Because **V{789}** is immediately discarded.

MOVE ASSIGNMENT

- ▶ Here is an example definition of a *move assignment operator*

```
V& operator=(V&& ov) { x = ov.x; return *this; }
```

- ▶ Here again is a typical situation when it gets used

```
V v3 {99};  
...  
v3 = V {789};
```

- ▶ Since **V{789}** is a temporary object, it doesn't take up resources (i.e. no slot in the stack frame).
 - The object **v3** is seen to be "taking over its resources."
 - The temporary **V** is seen as "moving out", and **v3** is seen as "moving in."

MOVE ASSIGNMENT

- ▶ Here is an example definition of a *move assignment operator*

```
V& operator=(V&& ov) { x = ov.x; return *this; }
```

- ▶ Here again is a typical situation when it gets used

```
V v3 {99};  
...  
v3 = V {789};
```

- ▶ It has a weird annotation of its argument.
- ▶ This is an *R-value reference*
 - L-value expressions are ones that can appear on the LHS of an assignment
 - R-value expressions are ones that only appear on the RHS of an assignment...

MOVE ASSIGNMENT

- ▶ Here is an example definition of a *move assignment operator*

```
V& operator=(V&& ov) { x = ov.x; return *this; }
```

- ▶ Here again is a typical situation when it gets used

```
V v3 {99};  
...  
v3 = V {789};
```

- ▶ It has a weird annotation of its argument.
- ▶ This is an *R-value reference*
 - L-value expressions are ones that can appear on the LHS of an assignment
 - R-value expressions are ones that only appear on the RHS of an assignment... like **V{789}**

MOVE CONSTRUCTOR

- ▶ There is also a *move constructor*

```
V(V&& ov) : x {ov.x} { }
```

- ▶ Here is a typical situation when it gets used:

```
V make_a_V(int x0) {  
    return V {x0};  
}
```

...

```
v3 = make_a_V(789);
```

MOVE CONSTRUCTOR

- ▶ There is also a *move constructor*

```
V(V&& ov) : x {ov.x} { }
```

- ▶ Here is a typical situation when it gets used:

```
V make_a_V(int x0) {  
    return V {x0};  
}
```

...

```
v3 = make_a_V(789);
```

- ▶ Note again the use of an R-value reference annotation.

AN EXAMPLE USE OF &&

- ▶ I tried to demonstrate these things in the sample code for this lecture.
 - So far, in **samples/copy_move/cm_value_debug.cc**
 - Run **make** to build an executable **./cmvd** and look at its output.
- ▶ There's an additional definition:

```
class V {  
    ...  
    friend int operator+(int i, V&& v);  
};  
int operator+(int i, V&& v) { return i+v.x; }
```

AN EXAMPLE USE OF &&

- ▶ I tried to demonstrate these things in the sample code for this lecture.
 - So far, in **samples/copy_move/cm_value_debug.cc**
 - Run **make** to build an executable **./cmvd** and look at its output.
- ▶ There's an additional definition:

```
class V {  
    ...  
    friend int operator+(int i, V&& v);  
};  
int operator+(int i, V&& v) { return i+v.x; }
```

- ▶ Here is where **it is used**:

```
V v4 = V{1 + V {3}};
```


AN EXAMPLE USE OF &&

- ▶ I tried to demonstrate these things in the sample code for this lecture.
 - So far, in `samples/copy_move/cm_value_debug.cc`
 - Run **make** to build an executable `./cmvd` and look at its output.
- ▶ There's an additional definition:

```
class V {  
    ...  
    friend int operator+(int i, V&& v);  
};  
int operator+(int i, V&& v) { return i+v.x; }
```

- ▶ Here is where **it is used**:

```
V v4 = V{1 + V {3}};
```

- Note the use of an **R-value reference** in its definition.

CONTAINER CLASSES VERSUS COPY/MOVE

- Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- Note that I allocate the array upon construction with a value.
- Note that I wrote the default constructor to set a null pointer instead.

CONTAINER CLASSES VERSUS COPY/MOVE

- Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- Note that I allocate the array upon construction with a value.
- Note that I wrote the default constructor to set a null pointer instead.

CONTAINER CLASSES VERSUS COPY/MOVE

- Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- Note that I allocate the array upon construction with a value.
- Note that I wrote the default constructor to set a null pointer instead.

CONTAINER CLASSES VERSUS COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead...
 - ➡ ... so that I could write move constructors that don't leak memory.

CONTAINER CLASSES VERSUS COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead.
- ▶ Note that I give back the array storage in the destructor, if not null.

CONTAINER CLASSES VERSUS COPY/MOVE

- ▶ Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- ▶ Note that I allocate the array upon construction with a value.
- ▶ Note that I wrote the default constructor to set a null pointer instead.
- ▶ Note that I give back the array storage in the destructor, if not null.

CONTAINER CLASSES VERSUS COPY/MOVE

- Recall my array-based class **R**, a companion to class **V**

```
class R {  
private:  
    int* a;  
public:  
    R(void) : a {nullptr} { };  
    R(int x0) : a {new int[1]} { a[0] = x0};  
    ~R(void) { if (a != nullptr) delete [] a; }  
}
```

- Note that I allocate the array upon construction with a value.
- Note that I wrote the default constructor to set a null pointer instead.
- Note that I give back the array storage in the destructor, if not null.
- What should the copy/move members do?

COPY CONSTRUCTOR AND ASSIGNMENT

- Here are the copy operations for class **R**

```
R::R(const R& r) : a {new int[1]} {  
    a[0] = r.a[0];  
}
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = new int[1];  
    a[0] = r.a[0];  
    return *this;  
}
```

- They each perform a deep copy of the data structure.

COPY CONSTRUCTOR AND ASSIGNMENT

- Here are the copy operations for class **R**

```
R::R(const R& r) : a {new int[1]} {  
    a[0] = r.a[0];  
}
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = new int[1];  
    a[0] = r.a[0];  
    return *this;  
}
```

- They each perform a *deep copy* of the data structure.

COPY CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the copy operations for class **R**

```
R::R(const R& r) : a {new int[1]} {  
    a[0] = r.a[0];  
}
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = new int[1];  
    a[0] = r.a[0];  
    return *this;  
}
```

- ▶ They each perform a deep copy of the data structure.
- ▶ But we also have to **deallocate** the destination's old storage.

MOVE CONSTRUCTOR AND ASSIGNMENT

► Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

► They can perform a shallow copy of the source object's data.

MOVE CONSTRUCTOR AND ASSIGNMENT

► Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

► They can perform a *shallow copy* of the source object's data.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

- ▶ They can perform a shallow copy of the source object's data.
- ▶ We still need to **give back the destination's old array** upon reassignment.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

- ▶ They can perform a shallow copy of the source object's data.
- ▶ We still need to give back the destination's old array upon reassignment.
- ▶ And it is standard practice to "clear out" the source of the move.

MOVE CONSTRUCTOR AND ASSIGNMENT

- ▶ Here are the move operations for class **R**

```
R::R(R&& r) {  
    a = r.a;  
    r.a = nullptr;  
}  
R& R::operator=(R&& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    r.a = nullptr;  
    return *this;  
}
```

```
R::~~R(void) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
}
```

- ▶ They can perform a shallow copy of the source object's data.
- ▶ We still need to give back the destination's old array upon reassignment.
- ▶ We *clear out* the source of the move in preparation for its destruction.

SHALLOW COPY CONSTRUCTOR AND ASSIGNMENT BUGGY!

- ▶ Here instead are shallow copy operations for class **R**

```
R::R(const R& r) : a {r.a} { }
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    return *this;  
}
```

- ▶ With these, we would have instances of **R** *sharing* the same array **a**.
- ▶ The destructor would eventually "double delete" that shared pointer.
- ▶ **NOTE** that shallow copying is sometimes desirable...

SHALLOW COPY CONSTRUCTOR AND ASSIGNMENT BUGGY!

- ▶ Here instead are shallow copy operations for class **R**

```
R::R(const R& r) : a {r.a} { }
```

```
R& R::operator=(const R& r) {  
    if (a != nullptr) {  
        delete [] a;  
    }  
    a = r.a;  
    return *this;  
}
```

- ▶ With these, we would have instances of **R** *sharing* the same array **a**.
- ▶ The destructor would eventually "double delete" that shared pointer.
- ▶ **NOTE** that shallow copying is sometimes desirable...
...and the STL "smart pointers" will allow us to do sharing, in a smart way.

LET'S REVISIT THE MYSTERY BUG FROM THE START

► RECALL: my change to **dc.c**

```
void output_top(Stck s) {  
    if (!s.is_empty()) {  
        std::cout << s.top() << std::endl;  
    }  
}
```

```
int main() {  
    ...  
    Stck s {100};  
    std::string entry;  
    do {  
        output_top(s);  
        // parse and handle entry  
        ...  
    } while (entry != q);  
}
```

LET'S REVISIT THE MYSTERY BUG FROM THE START

► **RECALL:** what happened when I ran it...

```
$ ./dc
You've just run my version of the Unix calculator utility 'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

LET'S REVISIT THE MYSTERY BUG FROM THE START

► **RECALL:** what happened when I ran it...

```
$ ./dc
You've just run my version of the Unix calculator utility 'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

Q: So what went wrong????

LET'S REVISIT THE MYSTERY BUG FROM THE START

► **RECALL:** what happened when I ran it...

```
$ ./dc
You've just run my version of the Unix calculator utility 'dc'.
It uses a stack to track intermediate calculations.
Enter a command just below (h for help):
p
[ ]
dc(23213,0x7fff7c877000) malloc: *** error for object
0x7fa93ae00000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

Q: So what went wrong????

A: I don't define a copy constructor for `Stck`. The default one does a shallow copy. When I pass `s` by value to `output_top` it is passed by value. The two stack objects share a pointer to the array data. It exits, the destructor gets called. It deletes the elements pointer.

ONE POSSIBLE FIX

```
void output_top(Stck s) { // pass s by ref, no copy made
    if (!s.is_empty()) {
        std::cout << s.top() << std::endl;
    }
}

int main() {
    ...
    Stck s {100};
    std::string entry;
    do {
        output_top(s);
        // parse and handle entry
    } while (entry != q);
    ...
}
```

SUMMARY

- ▶ COPY CONSTRUCTORS
- ▶ COPY ASSIGNMENT
- ▶ MOVE CONSTRUCTORS
- ▶ MOVE ASSIGNMENT
- ▶ ... are each used by the C++ compiler in various ways.
- ▶ If you are rolling your own data structures, then you need to become an expert and understand their subtleties.
- ▶ *Probably best to learn what's provided by the C++ Standard Template Library*

MODERN C++ WE COVER

- ▶ BASIC OBJECT-ORIENTATION: CLASSES, METHODS, CON-/DE-STRUCTORS ✓
- ▶ INHERITANCE **next week 10**
- ▶ TEMPLATES **week 10 or 11**
- ▶ SOME NITTY-GRITTY STUFF
 - OPERATOR OVERLOADING ✓
 - REFERENCES & ; **const** ; COPY/MOVE CONSTRUCTORS/ASSIGNMENT **10? 11?**
- ▶ THE C++ STANDARD TEMPLATE LIBRARY **week 11**
 - **vector**, **map**, **unordered_map**, ... **week 11**
- ▶ **lambda week 12**
- ▶ SMART POINTERS, "RAII": **shared_ptr** AND **weak_ptr week 12**

MODERN C++ WE COVER

- ▶ BASIC OBJECT-ORIENTATION: CLASSES, METHODS, CON-/DE-STRUCTORS ✓
- ▶ INHERITANCE **week 10** ✓
- ▶ TEMPLATES **week 10** ~~or 11~~ ✓
- ▶ SOME NITTY-GRITTY STUFF
 - OPERATOR OVERLOADING ✓
 - REFERENCES & ✓; **const** ✓; COPY/MOVE CONSTRUCTORS/ASSIGNMENT ✓
10? 11?
- ▶ THE C++ STANDARD TEMPLATE LIBRARY **week 11**
 - **vector**, **map**, **unordered_map**, ... **week 11**
- ▶ **lambda** ~~**week 12**~~ **week 11**
- ▶ SMART POINTERS, "RAII": **shared_ptr** AND **weak_ptr** **week 12** **11?**

NEXT WEEK

- ▶ BASIC OBJECT-ORIENTATION: CLASSES, METHODS, CON-/DE-STRUCTORS ✓
- ▶ INHERITANCE **week 10** ✓
- ▶ TEMPLATES **week 10 or 11** ✓
- ▶ SOME NITTY-GRITTY STUFF
 - OPERATOR OVERLOADING ✓
 - REFERENCES & ✓; **const** ✓; COPY/MOVE CONSTRUCTORS/ASSIGNMENT / **10? 11?**
- ▶ THE C++ STANDARD TEMPLATE LIBRARY **week 11**
 - **vector, map, unordered_map, ...** **week 11**
- ▶ **lambda** **week 12** **week 11**
- ▶ SMART POINTERS, "RAII": **shared_ptr** AND **weak_ptr** **week 12** **11?**