Slides by Andrew DeOrio,
Jeff Ringenberg and
Brian Noble

EECS 280

Programming and Introductory Data Structures

Deep Copies

IntSet.h

Review: IntSet

```
class IntSet {
  //OVERVIEW: mutable set of ints with bounded size
 public:
  IntSet();
  void insert(int v);
  void remove(int v);
  bool query(int v) const;
  int size() const;
  void print() const;
  static const int ELTS_CAPACITY = 100;
 private:
  int elts[ELTS_CAPACITY];
  int elts_size;
  int indexOf(int v) const;
};
```

IntSet.cpp

Review: IntSet

```
int IntSet::size() const {
  return elts_size;
}

int IntSet::indexOf(int v) const {
  for (int i = 0; i < elts_size; ++i) {
    if (elts[i] == v) return i;
  }
  return ELTS_CAPACITY;
}

bool IntSet::query(int v) const  {
  return indexOf(v) != ELTS_CAPACITY;
}
// ...
```

main.cpp

Review: using IntSet

```
#include "IntSet.h"
int main () {
  IntSet is;
  is.insert(7);
  is.insert(4);
  is.insert(7);
  is.print();
  is.remove(7);
  is.print();
  return 0;
}
```

```
g++ IntSet.cpp main.cpp
./a.out
{ 7 4 }
{ 4 }
```

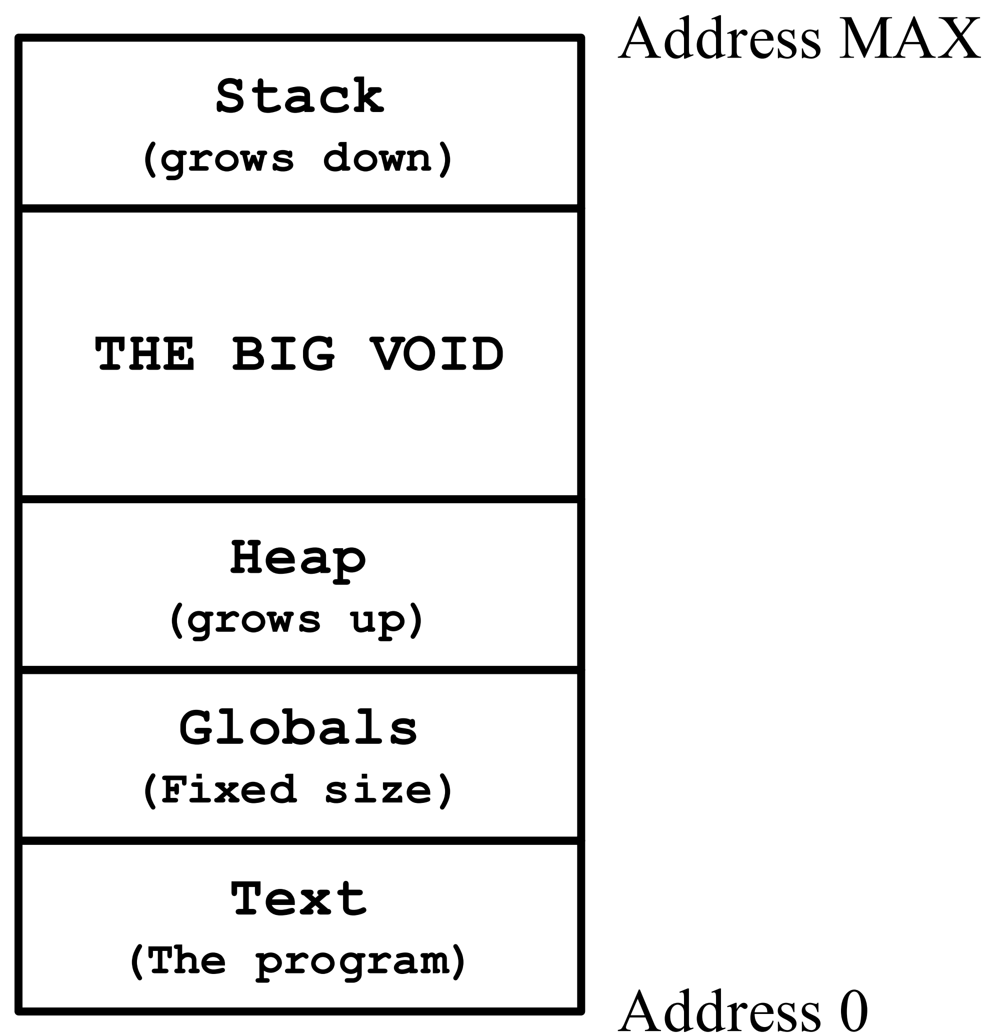
Review: global vs. local vs. dynamic

	Global	**Local**	**Dynamic**
Where in code?	Outside function	Inside function (block) or args	Anywhere you use `new`
When created	Beginning of program	Beginning of function (block)	You call `new`
When destroyed	End of program	End of function (block)	You call `delete`
Size	static	static	dynamic
Location	Globals	Stack	Heap

Review: stack and heap

- When functions are called, stack frames are created on the *stack*, which grows downward.

- When dynamic variables are allocated with `new`, they come from the *heap*, which grows upward.



Review: dynamic arrays

- Some dynamic variables have sizes known to the compiler

```
int *ip = new int(5); //int w/ value "5"
int *array = new int[5]; //array w/ 5 elts
```

- Recall that we can create dynamic variables whose size is not known to the compiler using **dynamic arrays**

```
int size = get_size_from_user(); //"200"
int *p = new int[size];
```

- Recall that we can destroy dynamic arrays using the array delete operator

```
delete[] p;
```

Review: destroying dynamic arrays

```
int * p = new int[5];
delete[] p;
```

- If you allocate an `array-of-T`, you **absolutely must** use the `delete[]` operator, and **not** the "plain" delete operator.
- Mixing them leads to undefined behavior

```
int * p = new int[5];
delete p; //bad!
```

- This is because the language runtime system has to keep track of how large arrays are – since the compiler doesn't know, it has to be dynamic information.

Classes with dynamic arrays

- Now, we can build a version of `IntSet` that allows the client to specify how large the capacity of the set should be
- Rather than hold an array explicitly, we have a pointer `elts` that will (eventually) point to a dynamically-created array
- `elts_size` still tells us how many elements are stored

```
class IntSet {
  int *elts;          //pointer to dynamic array
  int elts_size;      //current occupancy
  int elts_capacity; //capacity of array
  static const int ELTS_CAPACITY_DEFAULT = 100;
public:
  //...
};
```

Classes with dynamic arrays

- `elts_capacity` tells us the size of the allocated array
- `ELTS_CAPACITY_DEFAULT` tells us the initial size

```
class IntSet {
  int *elts;             //pointer to dynamic array
  int elts_size;         //current occupancy
  int elts_capacity; //capacity of array
  static const int ELTS_CAPACITY_DEFAULT = 100;
public:
  //...
};
```

Classes with dynamic arrays

- We'll base our changes on the unsorted implementation, and most methods don't need to change
- The constructor changes, and allocates a default size array

```
IntSet::IntSet()
  : elts_size(0),
    elts_capacity(ELTS_CAPACITY_DEFAULT) {
  elts = new int[ELTS_CAPACITY_DEFAULT];
}
```

Alternate constructor

- In addition to the default, we can write an alternate constructor
- It has the same name as the default, but a different type signature:

```
class IntSet {
  //...
public:
  //EFFECTS: creates an IntSet with default capacity
  IntSet();

  //EFFECTS:  creates an IntSet with specified capacity
  //REQUIRES: capacity > 0
  IntSet(int capacity);

  //...
};
```

Alternate constructor

- The alternate constructor creates an array of the specified size

```
IntSet::IntSet(int capacity)
  : elts_size(0), elts_capacity(capacity) {
  elts = new int[capacity];
}
```

Function overloading

- Recall that this is called *function overloading:* two different functions with the same name, but different prototypes
  - Since the compiler knows the argument types, it can select the correct constructor when a new object is created
- Different from *function overriding*, where a derived class has a function with the same name and prototype as the parent

```
IntSet is1;  //No arguments
             //default ctor runs

IntSet is2(200); //Integer argument
                 //alternate ctor runs
```

Building a new `IntSet`

- Notice that the two constructors are nearly identical:
- The only difference is whether we use `capacity` or `ELTS_CAPACITY_DEFAULT`

```
IntSet::IntSet()
  : elts_size(0),
    elts_capacity(ELTS_CAPACITY_DEFAULT) {
  elts = new int[ELTS_CAPACITY_DEFAULT];
}
```

```
IntSet::IntSet(int capacity)
  : elts_size(0), elts_capacity(capacity) {
  elts = new int[capacity];
}
```

Default arguments

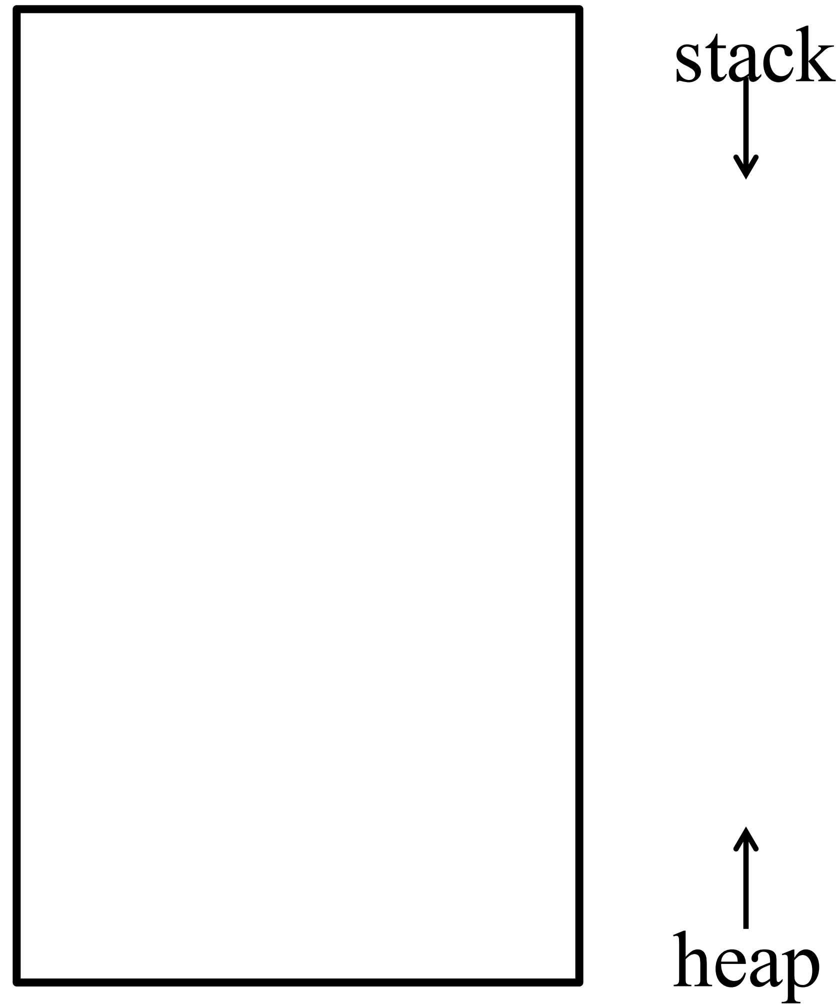
- Solve this duplication with *default argument*
- Define **only one** constructor, but make its argument optional

```
class IntSet {
public:
  //EFFECTS:  creates an IntSet with specified capacity
  //REQUIRES: capacity > 0
  IntSet(int capacity = ELTS_CAPACITY_DEFAULT);
  //...
}

IntSet::IntSet(int capacity)
  : elts_size(0), elts_capacity(capacity) {
  elts = new int[capacity];
}
```

Exercise: allocating classes

```
void foo() {
  IntSet is(3);
}
```

- Draw the stack and heap



Copy problems

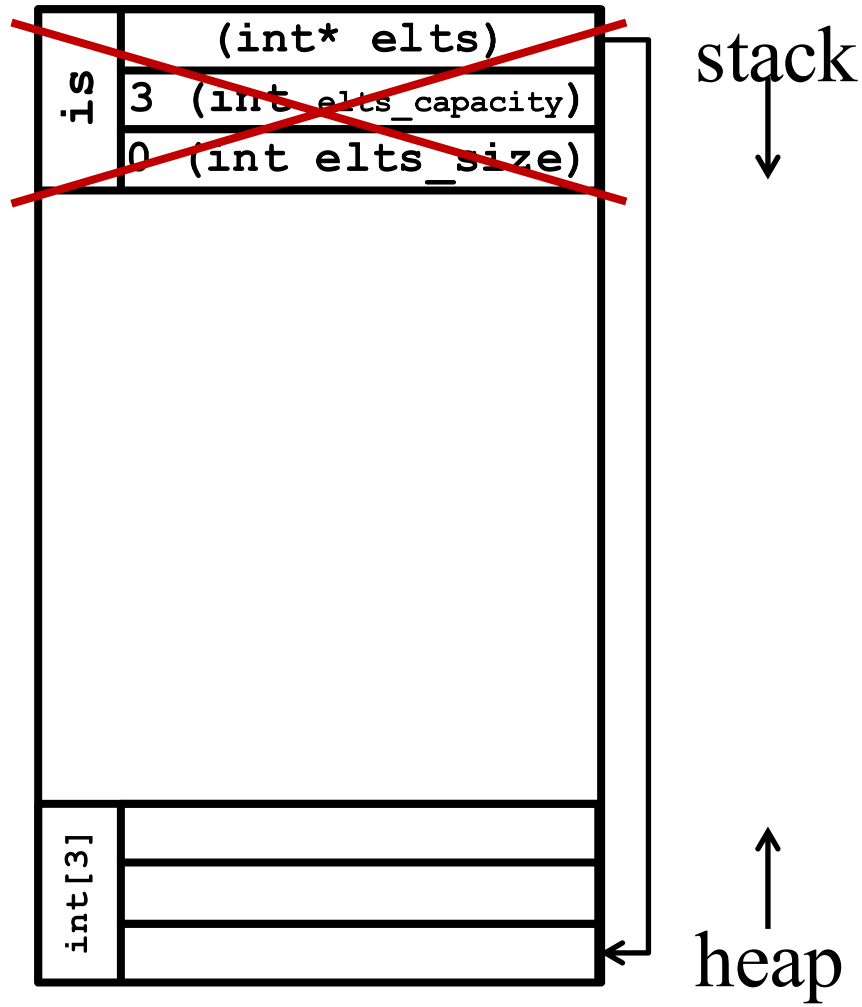
- Problem: what happens if we have a local `IntSet` inside of a function and the function returns?
- Why isn't this a problem with the static version of `IntSet`?

```
void foo() {
  IntSet is(3);
}
int main() {
  foo();

  //or worse!
  //for (int i=0; i<1000000; ++i)
  //  foo();
}
```

Copy problems

```
void foo() {
  IntSet is(3);
} //foo returns

int main() {
  foo();
}
```

- Array is still on the heap!
- Leak!



Destructors

- To solve this memory leak, we have to arrange to de-allocate the integer array whenever the "enclosing" `IntSet` is destroyed.
- We do this with a *destructor* and it is the opposite of a constructor
- The constructor ensures that the object is in fact a legal instance of its class and the destructor's job is to do the opposite
- If a class allocates dynamic storage, then the destructor is responsible for deallocating it

Destructors

```
class IntSet {
public:
  //...

  //EFFECTS: destroys this IntSet
  ~IntSet();

  //...
};

IntSet::~IntSet() {
  delete[] elts;
}
```

We have to use the array-based `delete` operator, not the "standard" `delete` operator, because the thing we are `delete[]`ing was created by `new[]`.

When does the destructor run?

- Destructors run automatically when an object is destroyed
- Local variable: when it goes out of scope

```
int foo() {
  IntSet is; //ctor runs
}//dtor runs
```

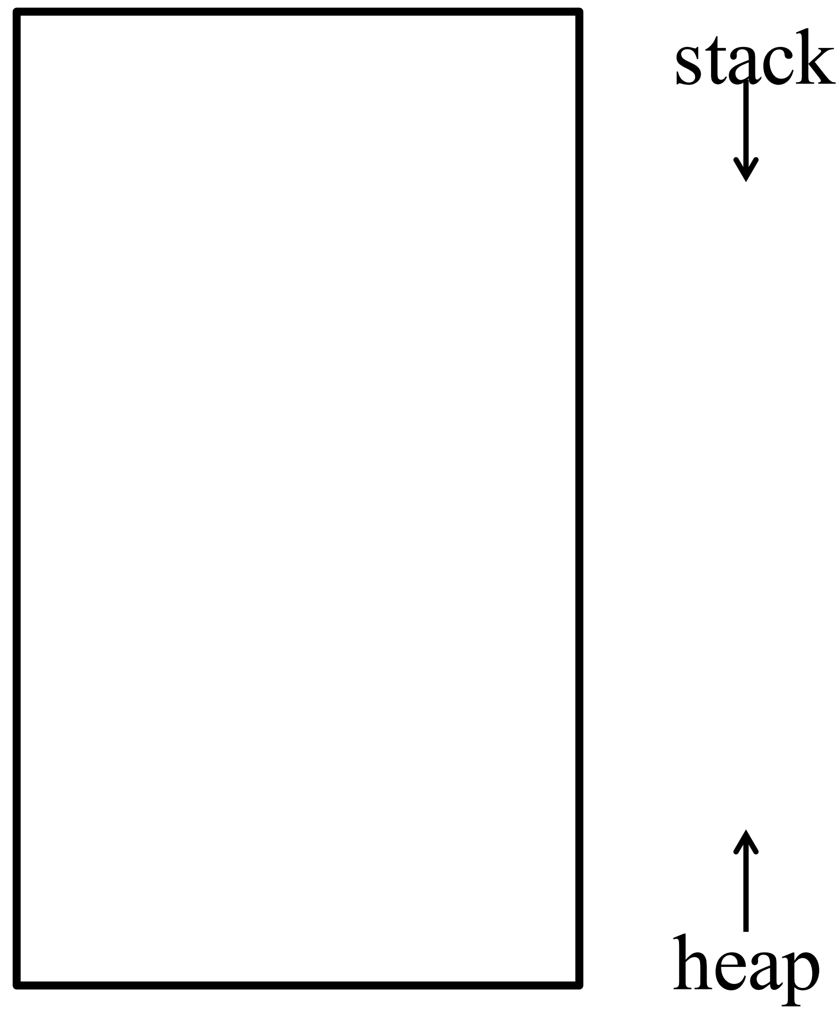
- Global variable: when the program ends

```
IntSet is; //ctor runs
int main() {
}
//dtor runs
```

Exercise: deallocating classes

```
void foo() {
  IntSet is(3);
}
```

- Draw the stack and heap
- How much memory is leaked?
  - Assume 4B int and 8B int*



When does the destructor run?

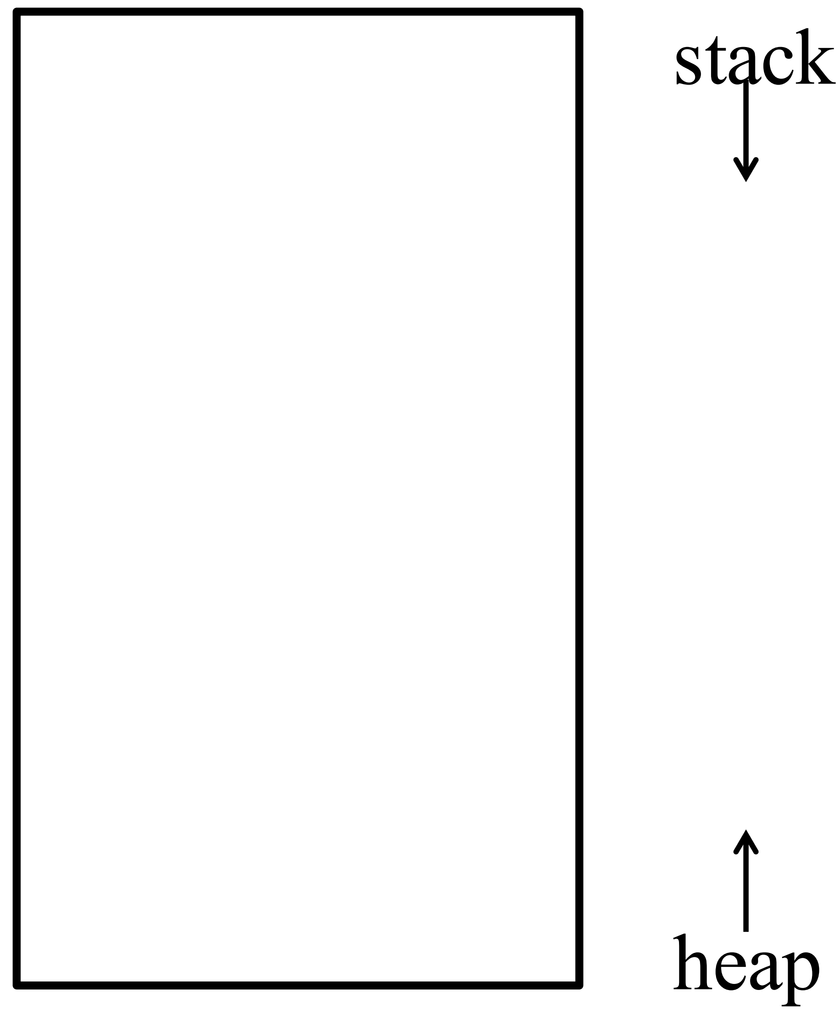
- Destructors run automatically when an object is destroyed
- Dynamic variable: when it is deleted

```
int main() {
  IntSet *ptr = new IntSet; //ctor runs
  delete ptr; ptr=0;  //dtor runs
}
```

Exercise: allocating classes

```
int main() {
  new IntSet(3);
}
```

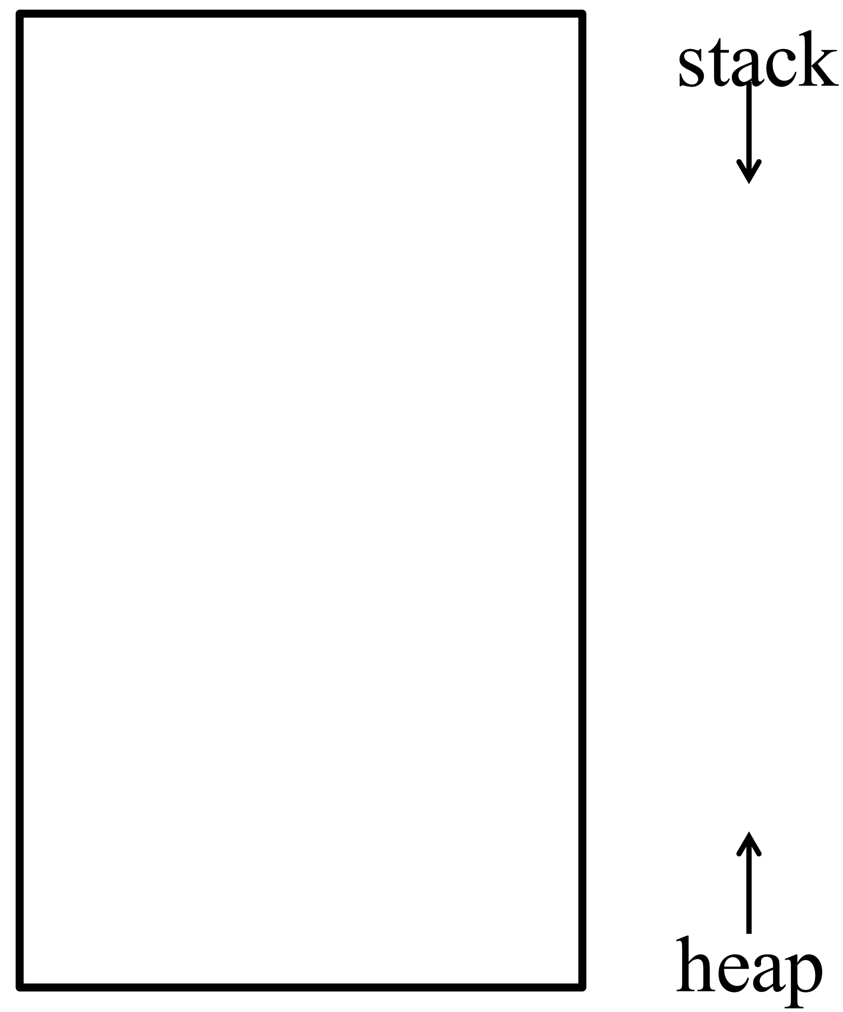
- Draw the stack and heap
- How much memory is leaked?
  - Assume 4B int and 8B int*
- Fix the leak



Exercise: copying classes

```
void foo(IntSet x) {
  //do something
}
int main() {
  IntSet is(3);
  is.insert(5);
  foo(is);
  is.query(5);
}
```

- Draw the stack and heap
- Hint: classes are passed by value



Dangling pointers

- Code with dangling pointer problem (prev. slide)

```
void foo(IntSet x){/*...*/}
int main() {
  IntSet is(3);
  is.insert(5);
  foo(is);
  is.query(5);
}
```

- Another example with the same problem

```
int main() {
  IntSet is(3);
  is.insert(5);
  {
    IntSet x = is;
  }
  is.query(5);
}
```

Member variables are copied from `is` to `x`, but both share the `elts` array on the heap. When `x` goes out of scope and is destroyed, `is.elts` dangles.

Copy object w/ dynamic memory

- So, what's the problem?

- The semantics of by-value arguments and assignment specify that we should copy the contents of `is` to `x`, but unfortunately, that's not what happens since the two end up sharing the `elts` array

- What we really want is to copy the array in addition to the elements themselves

Dangling pointers

- This code uses the copy constructor to copy `is` to `x`

```
void foo(IntSet x){/*...*/}
int main() {
  IntSet is(3);
  is.insert(5);
  foo(is);
  is.query(5);
}
```

- This code uses the assignment operator to copy `is` to `x`

```
int main() {
  IntSet is(3);
  is.insert(5);
  {
    IntSet x = is;
  }
  is.query(5);
}
```

Both the copy constructor and assignment operator make copies of a class instance. Let's start by fixing the copy constructor

Copy constructor

- Declare a copy constructor for our `IntSet` class

```
class IntSet {
public:
  //EFFECTS: create an IntSet that is a copy of other
  IntSet(const IntSet &other);

  //...
};
```

- Why pass by reference? It's more efficient
- Why pass by `const` reference? The copy constructor should *not* change the `other` `IntSet`

Copy constructor

- Like other constructors, the copy constructor must create a new `IntSet`, starting from a formless blob of memory
- Unlike other constructors, the copy constructor must copy everything from the `other IntSet`

- The default constructor had one task
  1. Initialize the member variables
- The copy constructor has two tasks
  1. Initialize the member variables
  2. Copy everything from the `other IntSet`

Copy constructor

```
IntSet::IntSet(const IntSet &other) {

  //1. Initialize member variables
  elts = new int[other.elts_capacity];
  elts_size = other.elts_size;
  elts_capacity = other.elts_capacity;

  //2. Copy everything from the other IntSet
  for (int i = 0; i < other.elts_size; ++i)
    elts[i] = other.elts[i];
}
```

Copy constructor

```
IntSet::IntSet(const IntSet &other) {

  //1. Initialize member variables
  elts = new int[other.elts_capacity];
  elts_size = other.elts_size;
  elts_capacity = other.elts_capacity;

  //2. Copy everything from the other IntSet
  for (int i = 0; i < other.elts_size; ++i)
    elts[i] = other.elts[i];
}
```

An `IntSet` method has access to the `private` members of both `this` instance and the `other` instance

Deep copies

- Contrast this copy constructor with the default method of copying, which does only copies member variables, including pointers

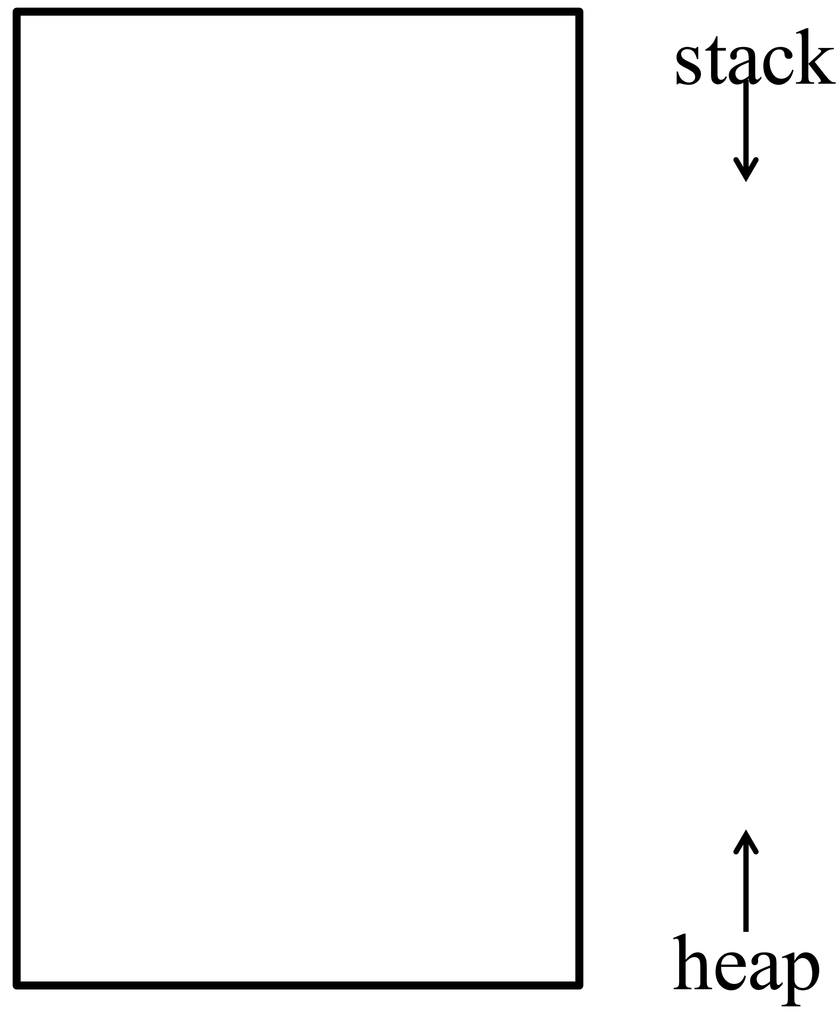
- The copy constructor we've written follows pointers and copies the things they point to, rather than just copying the pointers

- This is called a *deep copy*, as opposed to the default behavior which called a *shallow copy*

Exercise: copy constructor

```
void foo(IntSet x) {
  //do something
}
int main() {
  IntSet is(3);
  is.insert(5);
  foo(is);
  is.query(5);
}
```

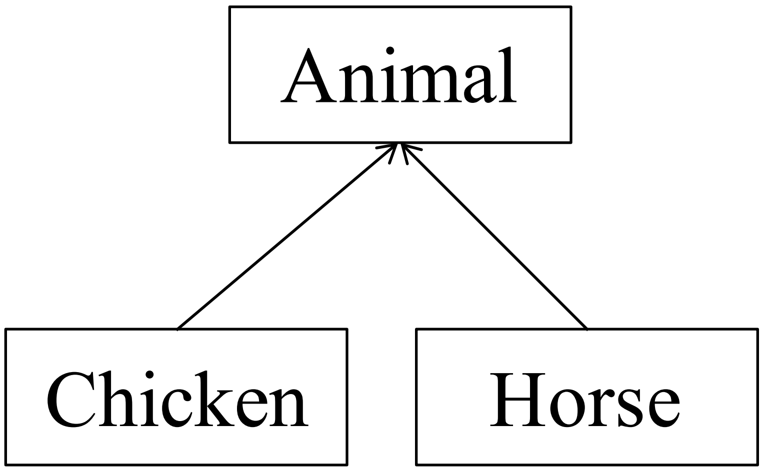
- Draw the stack and heap
- Identify where the copy constructor is used



Destructors and polymorphism

- We've talked about constructors and polymorphism
- When you create a object, all the constructors run, starting with the base class
- Now, let's see what happens when we mix destructors with polymorphism

Destructors and polymorphism




```
class Animal {
  virtual void talk() {}
};

class Chicken : public
Animal {
public:
  virtual void talk()
  { cout << "cluck\n"; }
};

class Horse : public
Animal {
public:
  virtual void talk()
  { cout << “neigh\n"; }
};
```

Destructors and polymorphism

```
int main() {
  Animal *a = ask_user(); //input: "Chicken"
  //do something with a
  delete a; a=0;
}
```

- Now the destructor runs.
- Group discussion: which destructor?

Destructors and polymorphism

```
class Animal {
public:
  virtual void talk() {}
  virtual ~Animal() {}
};
```

- Polymorphic objects need virtual destructors
- Put another way: if you have a virtual function and a destructor, then the destructor probably needs to be virtual too.

```
class Animal {
  virtual void talk() {}
  virtual ~Animal {}
};
```

Destructors and po

```
Animal *a = ask_user(); //input: "Chicken"
// do something with a
delete a; a=0;
```

- For variable `a`, Actual type (`Chicken`) is different from apparent type (`Animal`)
- Since dtor is virtual, correct dtors (`~Chicken(), then ~Animal()`) are selected at runtime.