Slides by Andrew DeOrio,
Jeff Ringenberg and
Brian Noble

EECS 280

Programming and Introductory Data Structures

Deep Copies, The Big Three, and Resizing

Review: dynamic arrays

- Last time, we built an `IntSet` with a flexible capacity

```
int main() {
  IntSet is(3); //IntSet with capacity for 3 ints
}
```

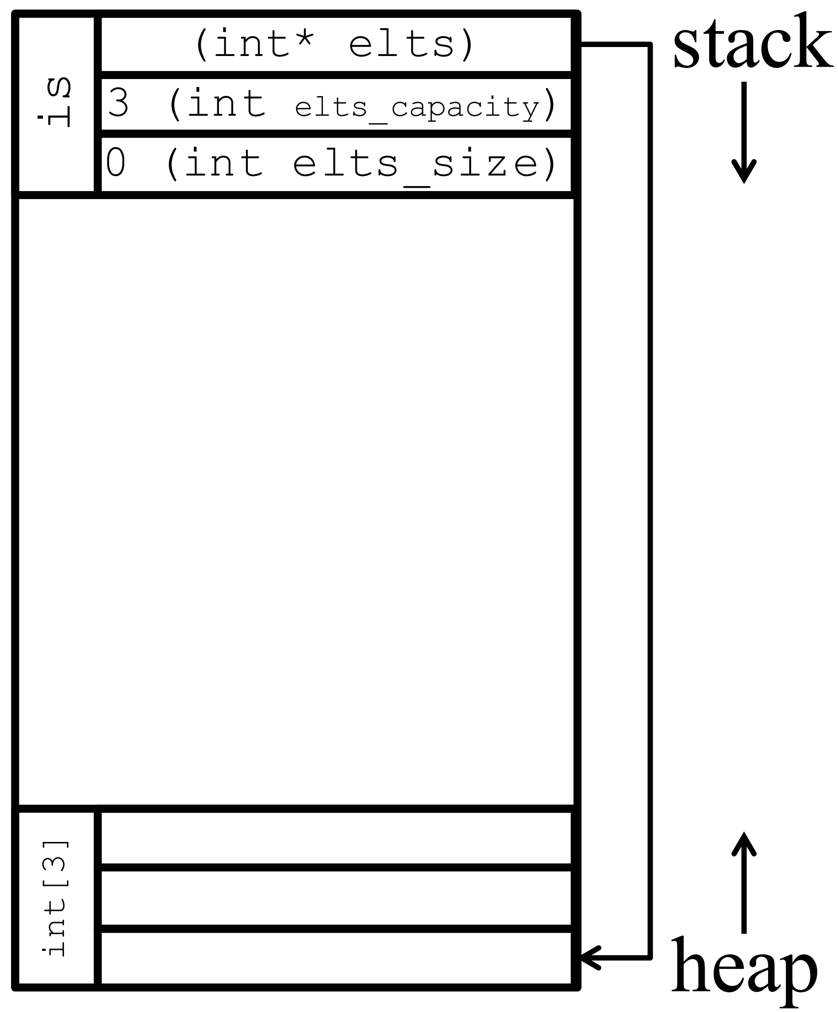
- We used a pointer to a dynamic array to store the set

```
class IntSet {
  int *elts;          //pointer to dynamic array
  int elts_size;      //current occupancy
  int elts_capacity; //capacity of array
public:
  IntSet();           //default constructor
  //...
};
```

Review: dynamic arrays

- The constructor allocated a dynamic array on the heap

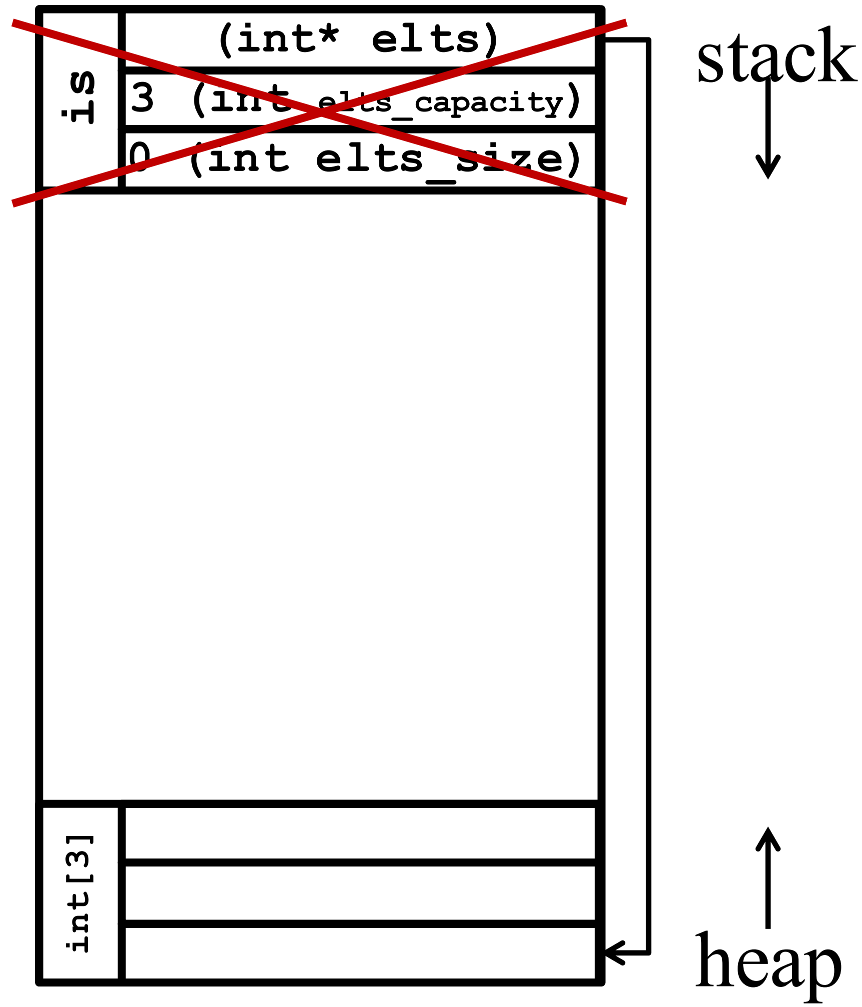
```
IntSet::IntSet(int capacity)
  : elts_size(0),
    elts_capacity(capacity) {
  elts = new int[capacity];
}

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



Review: leaking memory problem

- We had a problem when a local `IntSet` variable was destroyed
- This was because the lifetime of dynamic variables is managed by the programmer, **not automatically**

```
void foo() {
  IntSet is(3);
} //is goes out of scope,
  //but dynamic array does not

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



Review: destructor

- We fixed this problem by adding a destructor, which deleted the dynamically allocated memory

```
class IntSet {
public:
  ~IntSet();
  //...
};
```

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

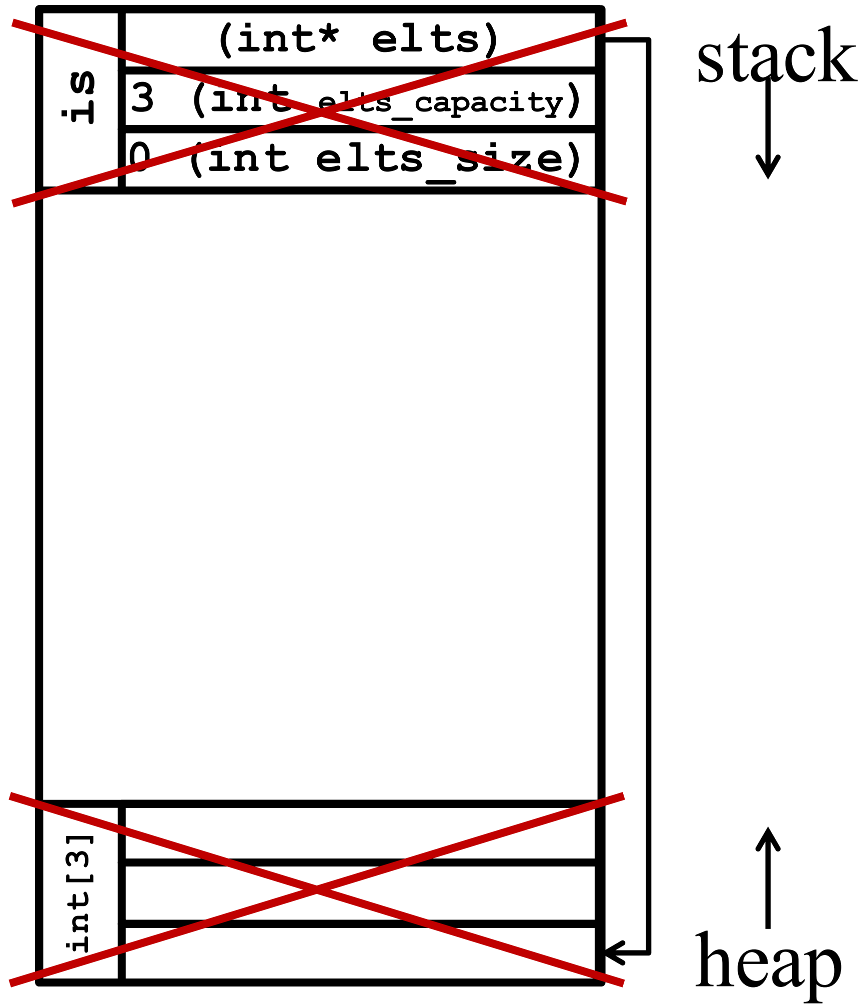
- A destructor runs automatically
  - Local variable: when it goes out of scope
  - Global variable: when the program ends
  - Dynamic variable: when it is deleted

Review: destructor

- A destructor runs automatically when a variable's lifetime ends

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

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

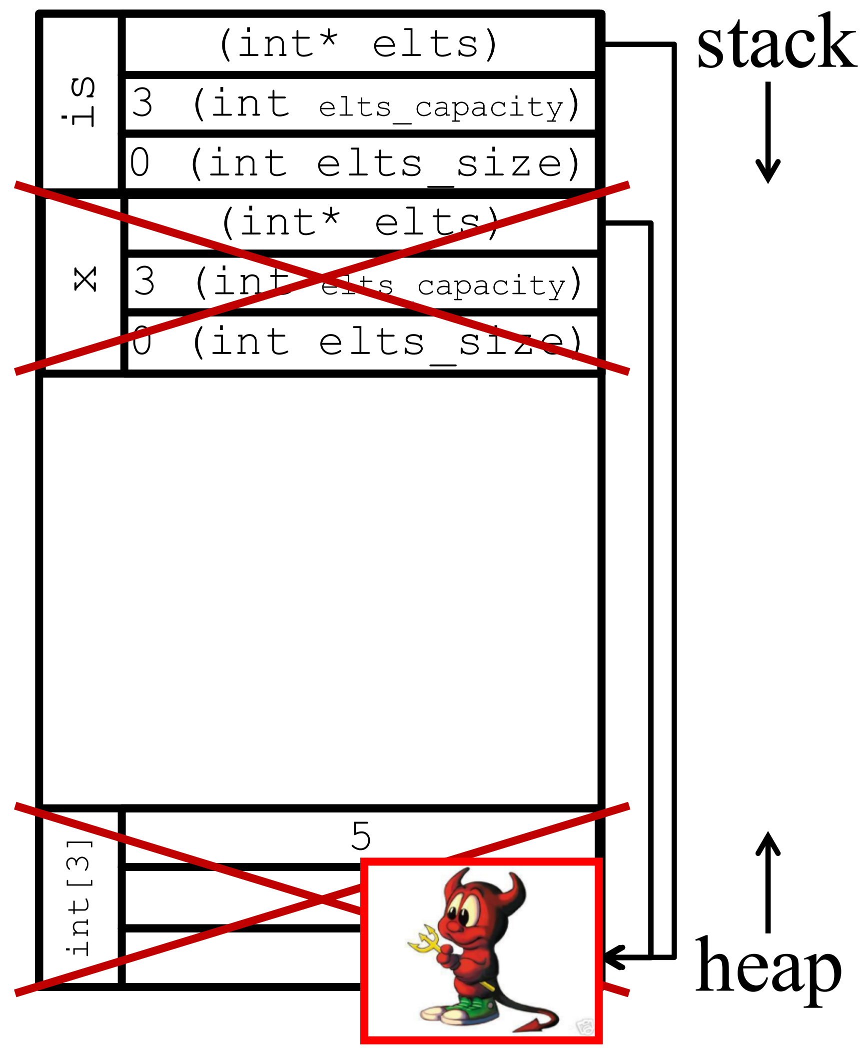
- Leak fixed!



Review: copy problem

- We had anothger problem with `IntSet` and pass-by-value
- The `IntSet` member variables were copied, but not the array on the heap

```
void foo(IntSet x) {
  //do something
}//shared array destructed!

int main() {
  IntSet is(3);
  is.insert(5);
  foo(is);
  is.query(5); //undefined!
}
```



Review: copy constructor

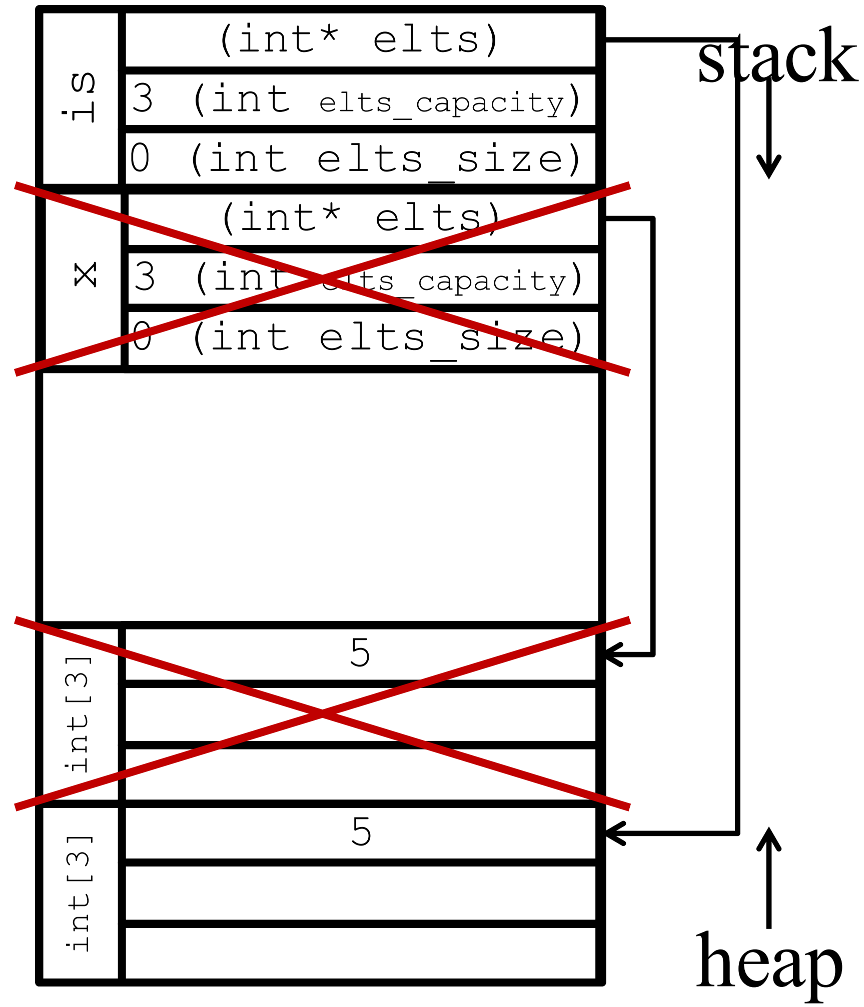
- Solution: copy constructor that copies the array on the heap, called a *deep copy*

```
class IntSet {
public:
  IntSet(const IntSet &other); //copy ctor prototype
  //...
};

IntSet::IntSet(const IntSet &other) {
  elts = new int[other.elts_capacity];
  elts_size = other.elts_size;
  elts_capacity = other.elts_capacity;
  for (int i = 0; i < other.elts_size; ++i)
    elts[i] = other.elts[i];
}
```

Review: copy problem

- Problem with `IntSet` and pass-by-value
- The `IntSet` member variables were copied, but the array on the heap was not

```
void foo(IntSet x) {
  //do something
}//copy is destructed

int main() {
  IntSet is(3);
  is.insert(5);
  foo(is);//copy ctor runs
  is.query(5); //OK!
}
```



Problems with assignment "="

- Draw the stack and the heap for this code:

```
int main() {
  IntSet is1(3);
  IntSet is2(6);
  is2 = is1;
}
```

- Does it cause errors? Leak memory?
- Hint: the assignment operator (equals "=" sign) does a shallow copy

Problems with assignment "="

- How do we fix this?

```
int main() {
   IntSet is1(3);
   IntSet is2(6);
   is2 = is1; //assignment of is1 to is2
}
```

- By default, the compiler will use a shallow copy
- Like the copy constructor, assignment must do a **deep copy** of the right hand side (`is1`) to the left hand side (`is2`)
- However, the class instance at the left hand side (`is2`) must first be destroyed, otherwise we have a memory leak!
- To implement this, we redefine the *assignment operator* for an `IntSet` by using *operator overloading*

Operator overloading

- *Operator overloading* lets us customize what happens when we use a built-in symbol

```
int main() {
  IntSet is1(3);
  IntSet is2(6);
  is2 = is1;
}
```

- Here, we want to change what the equals "=" sign does, by doing a deep copy instead of a shallow copy

Overloaded assignment operator

- Here's how we customize the assignment "=" operator

```
class IntSet {
  // data elements
  ...
  public:
  // Constructors

  //EFFECTS: assignment operator does a deep copy
  IntSet & operator= (const IntSet &rhs);
  //...
};
```

Overloaded assignment operator

- Using the “=“ symbol is actually calling a function

```
int main() {
  IntSet is1(3);
  IntSet is2(6);
  is2 = is1;
}
class IntSet {
 public:
  IntSet & operator= (const IntSet &rhs);
  //...
};
```

Overloaded assignment operator

- Why does `operator=` return a reference to an `IntSet`?

```
class IntSet {
 public:
  IntSet & operator= (const IntSet &rhs);
  //...
};
```

- So you can chain operations together, like this

```
int main() {
  IntSet is1(3), is2(6), is3(3);
  is3 = is2 = is1;
}
```

Implementing `operator=`

- Our overloaded assignment operator will share some code in common with the destructor and the copy constructor
- Destructor `~IntSet()`
  1. **Delete dynamic array**
- Copy constructor `IntSet(const IntSet &other)`
  1. Initialize member variables
  2. **Copy variables and dynamic array from `other IntSet`**
- Overloaded assignment operator
  `IntSet & operator=(const IntSet &rhs)`
  1. **Delete dynamic array**
  2. **Copy variables and dynamic array from `rhs IntSet`**

Implementing `operator=`

- Overloaded assignment operator
  `IntSet & operator=(const IntSet &rhs)`
  1. Delete dynamic array
  2. Copy variables and dynamic array from `rhs` `IntSet`

```
IntSet & IntSet::operator= (const IntSet &rhs) {
  delete[] elts; //delete dynamic array

  elts = new int[rhs.elts_capacity]; //copy variables
  elts_size = rhs.elts_size;         //from rhs IntSet
  elts_capacity = rhs.elts_capacity;

  for (int i = 0; i < rhs.elts_size; ++i) { //copy
    elts[i] = rhs.elts[i];                  //array
  }
  // …
}
```

Understanding `this`

```
IntSet & IntSet::operator= (const IntSet &rhs) {
  //...
  return *this;
}
```

- We need to return a reference to the current `IntSet` so that chaining works

```
int main() {
  IntSet is1(3), is2(6), is3(3);
  //fill is1, is2, is3
  is3 = is2 = is1;
}
```

Understanding `this`

- Every member function has "secret" input called `this`

```
class IntSet {
public:
  IntSet(IntSet *this);
  void insert(IntSet *this, int v);
  void remove(IntSet *this, int v);
  bool query(IntSet *this, int v) const;
  int size(IntSet *this) const;
  void print(IntSet *this) const;
  //...
};
```

- Detail: it would actually be `IntSet *const this` because you can't change the pointer

Understanding `this`

How it looks (real code)

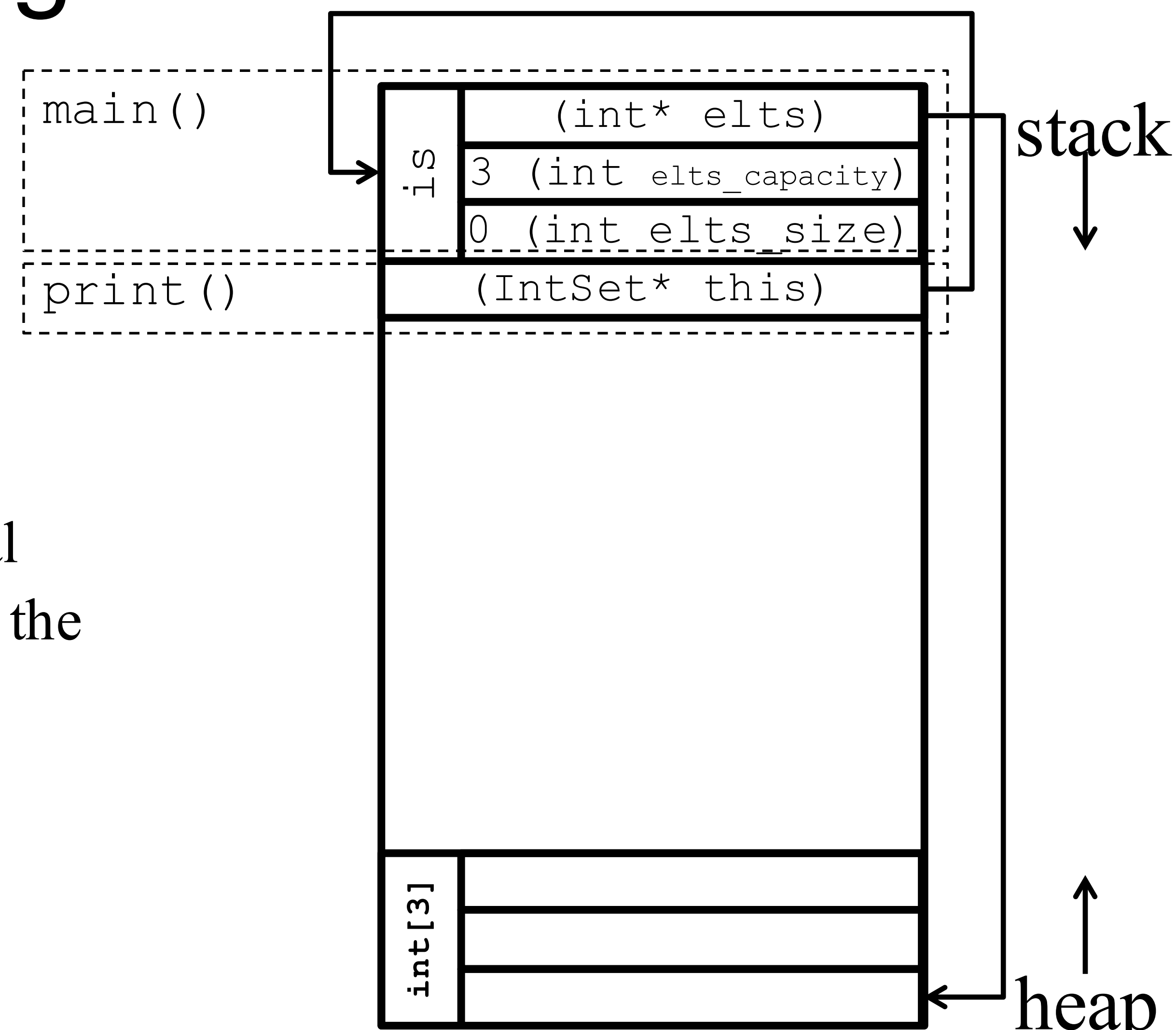
```
void IntSet::print() const {

  for (int i=0; i<elts_size; ++i)
    cout << elts[i] << " ";
}



int main() {
  IntSet is1;
  //fill is1
  is1.print();
}
```

How it works (pseudocode)

```
void IntSet::print(IntSet *this)
   const {
  for (int i=0; i<this->elts_size;
       ++i)
    cout << this->elts[i] << " ";
}

int main() {
  IntSet is1;
  //fill is1
  print(&is1);
}
```

Understanding `this`

```
int main() {
  IntSet is(3);
  is.print();
}
```



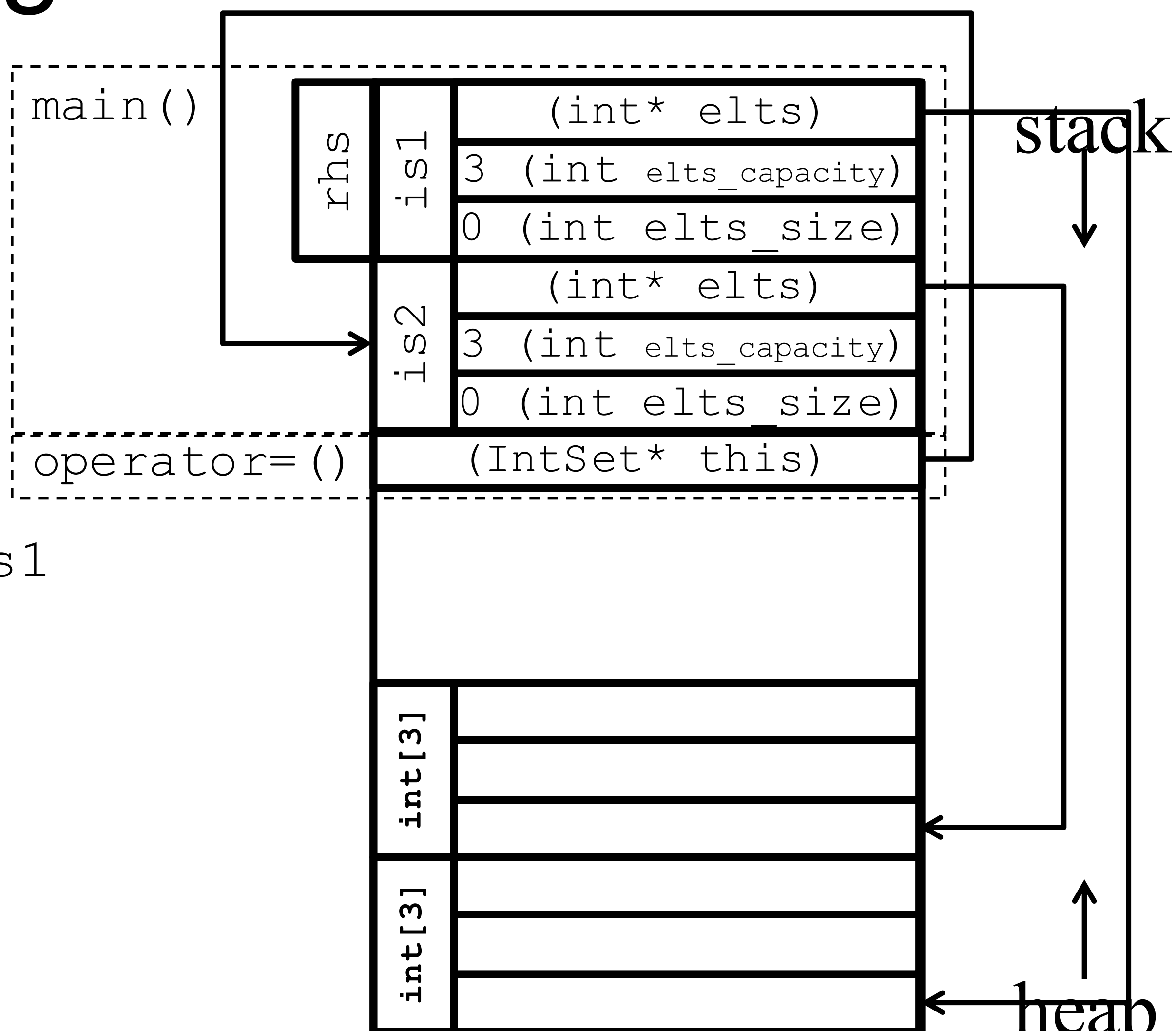
- Think of `this` as a local variable which points to the current instance

Understanding `this`

```
IntSet &
IntSet::operator=
(const IntSet &rhs){
  //...
  return *this;
}
```

- `rhs` is another name for `is1`
- `this` is a pointer to `is2`

```
int main() {
  IntSet is1(3);
  IntSet is2(3);
  is2 = is1;
}
```



Exercise

- Problem: what happens if we do this?
- Hint: draw a memory diagram

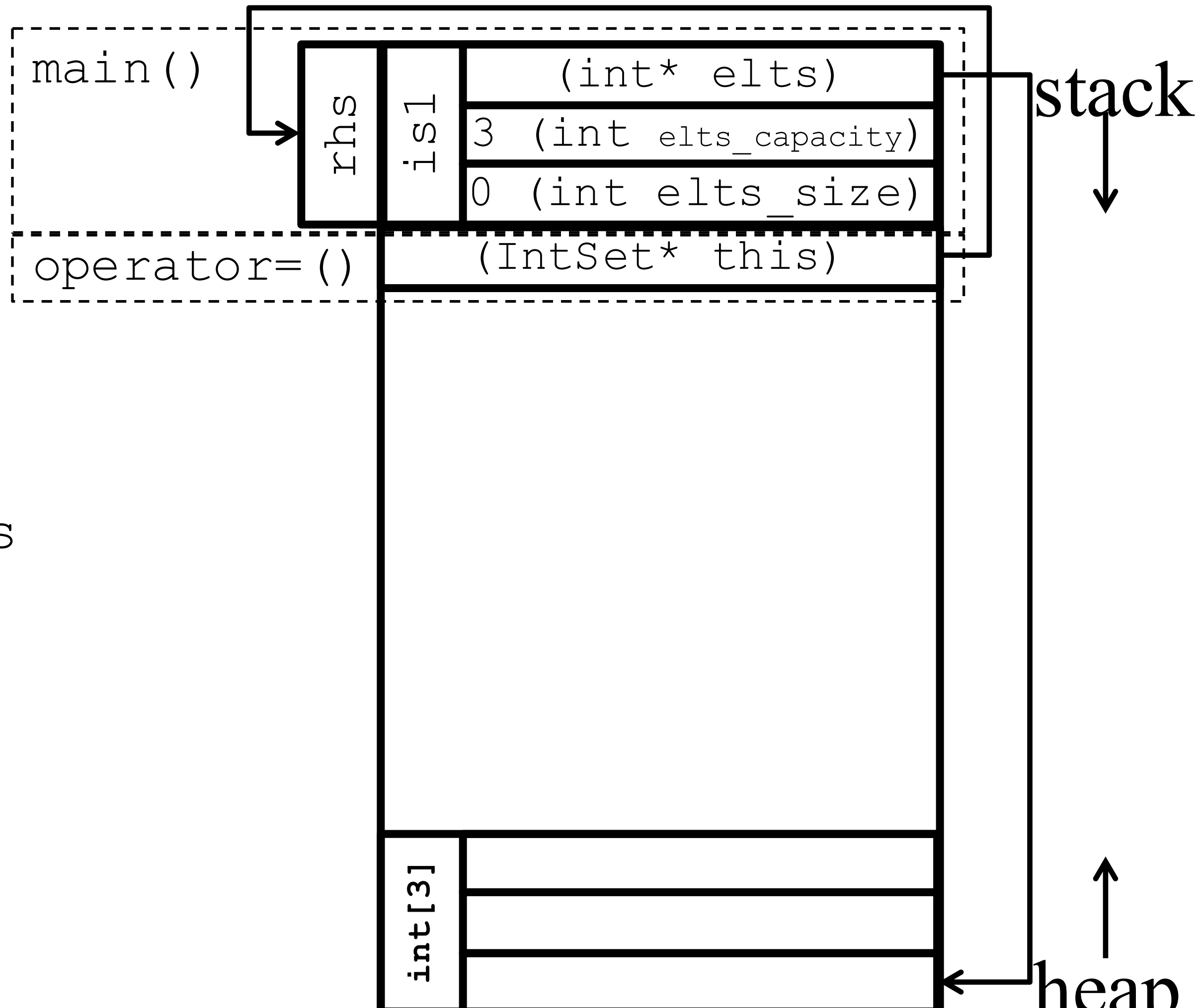
```
int main() {
  IntSet is(3);
  is.insert(5);
  is = is;
}
```

Exercise

```
IntSet &
IntSet::operator=
(const IntSet &rhs){
  //...
  return *this;
}
```

- `rhs` is another name for `is`
- `this` is a pointer to `is`

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



Completed assignment operator

```
IntSet & IntSet::operator= (const IntSet &rhs) {
  if (this == &rhs) return *this; //fix "is = is"
  delete[] elts; //remove all

  elts = new int[rhs.elts_capacity];
  elts_size = rhs.elts_size;
  elts_capacity = rhs.elts_capacity;

  for (int i = 0; i < rhs.elts_size; ++i)
    elts[i] = rhs.elts[i];

  return *this;
}
```

The Rule of the Big Three

- This lecture (and the last one) can be summarized with a simple rule: the Rule of *The Big Three*

- If you have any dynamically allocated storage in a class, you must provide:
  1. A destructor
  2. A copy constructor
  3. An overloaded assignment operator

- If you find yourself writing one of these, you almost certainly need all of them

Modifying `insert()`

- We have modified `IntSet` to allow a client to specify the capacity of an `IntSet`

- However, this doesn't really get around the “big instance” problem, since the client itself might not know how big the set will grow

- So, what we **really** want to do is to create an `IntSet` that can grow as big as it needs to

- To do this, we only need to modify the `insert()` method

Modifying `insert()`

- In this example, we'll be working with the version that allows clients to specify the **initial** size of the `IntSet`.
- We will use the unsorted representation, because it's a bit easier, and we can focus on the act of resizing, not the act of inserting

```
//REQUIRES: set is not full
void IntSet::insert(int v) {
  assert(elts_size < ELTS_CAPACITY);//full!
  if (indexOf(v) != elts_capacity) return;
  elts[elts_size++] = v;
}
```

Old version

Modifying `insert()`

- In this example, we'll be working with the version that allows clients to specify the initial size of the `IntSet`
- We're going to modify `insert()` so that it can grow the array

```
//REQUIRES: set is not full
void IntSet::insert(int v) {
  assert(elts_size < ELTS_CAPACITY);//full!
  if (indexOf(v) != elts_capacity) return;
  if(elts_size == elts_capacity) grow();
  elts[elts_size++] = v;
}
```

New version

Modifying `insert()`

- The `grow` method won't take any arguments or return any values.
- It should never be called from outside of the class, so make it private

```
class IntSet {
  //...
  // EFFECTS: enlarge the elts array,
  //          preserving current contents
  void grow();
public:
  // ...
};
```

Modifying `insert()`

- Grow will look a lot like the assignment operator

- It must perform the following steps:
  1. Allocate a bigger array
  2. Copy the smaller array to the bigger one
  3. Destroy the smaller array
  4. Modify `elts` and `elts_capacity` to reflect the new array

- Write this function

```
void IntSet::grow();
```

Modifying `insert()`

```
void IntSet::grow() {
1   int *tmp = new int[elts_capacity + 1];
    for (int i = 0; i < elts_size; ++i) {
2     tmp[i] = elts[i];
    }
3   delete [] elts;
4   elts = tmp;
    elts_capacity += 1;
}
```

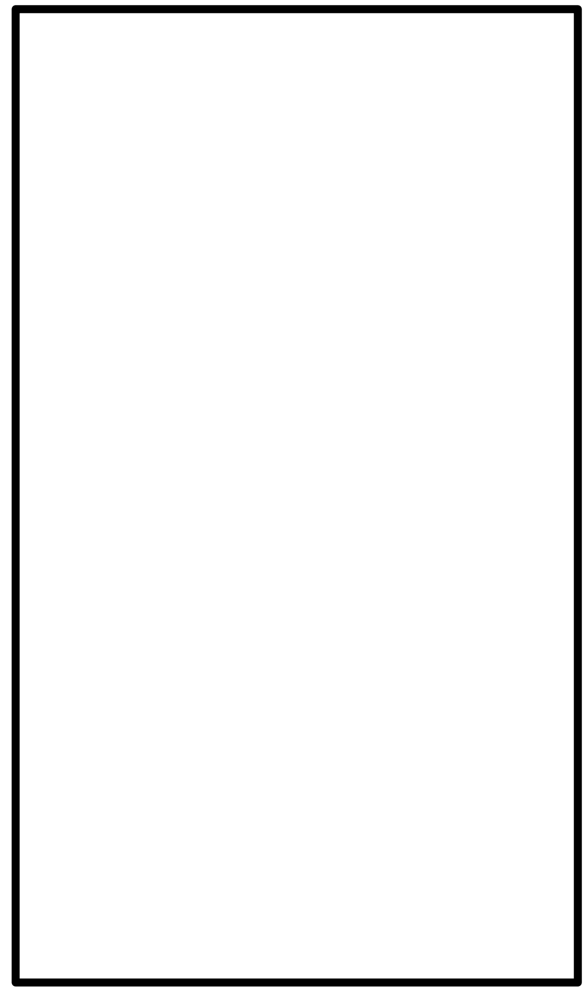
1. Allocate a bigger array
2. Copy the smaller array to the bigger one
3. Destroy the smaller array
4. Modify `elts` and `elts_capacity` to reflect the new array

`insert()` example

```
void IntSet::insert(int v) {
  if (indexOf(v) != elts_capacity) return;
  if(elts_size == elts_capacity) grow();
  elts[elts_size++] = v;
}
```

```
void IntSet::grow() {
  int *tmp = new int[elts_capacity + 1];
  for (int i = 0; i < elts_size; ++i)
    tmp[i] = elts[i];
  delete [] elts;
  elts = tmp;
  elts_capacity += 1;
}
```

```
int main() {
  IntSet is(1);
  is.insert(1);
  is.insert(2);
}
```

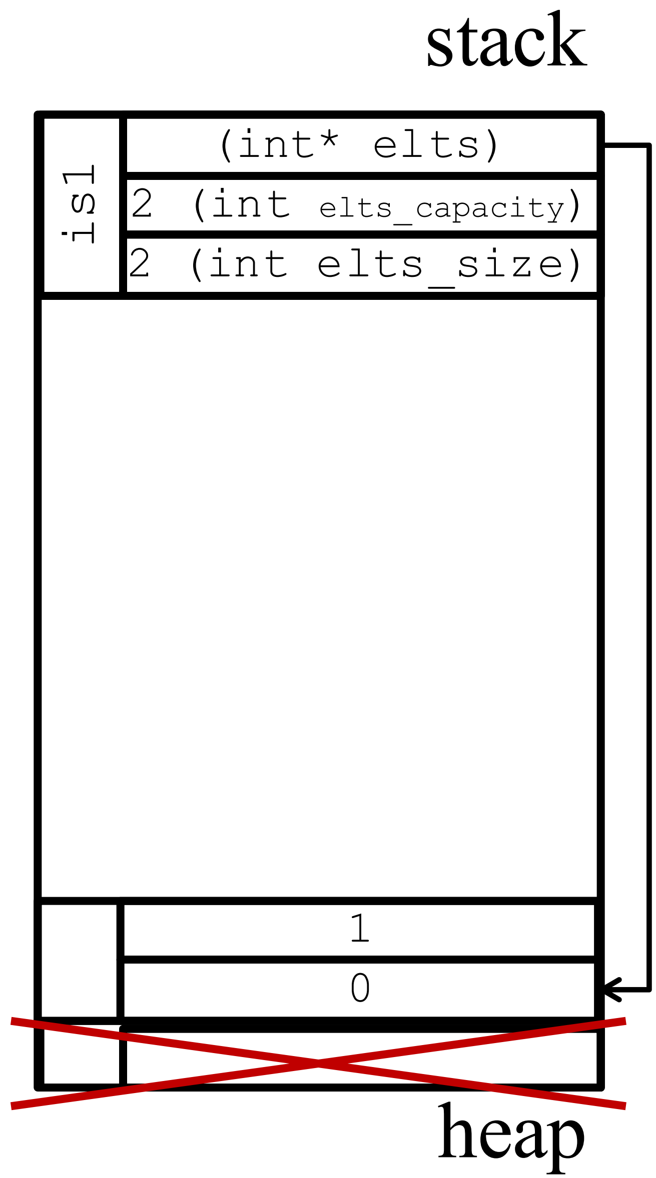


`insert()` example

```
void IntSet::insert(int v) {
  if (indexOf(v) != elts_capacity) return;
  if(elts_size == elts_capacity) grow();
  elts[elts_size++] = v;
}
```

```
void IntSet::grow() {
  int *tmp = new int[elts_capacity + 1];
  for (int i = 0; i < elts_size; ++i)
    tmp[i] = elts[i];
  delete [] elts;
  elts = tmp;
  elts_capacity += 1;
}
```

```
int main() {
  IntSet is(1);
  is.insert(0);
  is.insert(1);
}
```



Upper bound

- When is `grow()` called?
- How many array elements are copied?

```
int main() {
  IntSet is(1);
  is.insert(0);
  is.insert(1);
  is.insert(2);
  is.insert(3);
}
```

```
void IntSet::insert(int v) {
  if (indexOf(v) != elts_capacity) return;
  if(elts_size == elts_capacity) grow();
  elts[elts_size++] = v;
}
```

```
void IntSet::grow() {
  int *tmp = new int[elts_capacity + 1];
  for (int i = 0; i < elts_size; ++i)
    tmp[i] = elts[i]; //copy
  delete [] elts;
  elts = tmp;
  elts_capacity += 1;
}
```

Upper bound

- When is `grow()` called?
- How many array elements are copied?

```
int main() {
  IntSet is(1);
  is.insert(0);
  is.insert(1);  //grow(), 1 copy
  is.insert(2);  //grow(), 2 copies
  is.insert(3);  //grow(), 3 copies
}
```

Upper bound

- When is `grow()` called?
- How many array elements are copied?

```
int main() {
  IntSet is(1);
  int n = /* some large number */;
  for (int i=0; i<n; ++i) {
    is.insert(i);
  }
}
```

Upper bound

```
int main() {
  IntSet is(1);
  int n = /*...*/;
  for (int i=0; i<n; ++i)
    is.insert(i);
}
```

- Let's unroll this loop:

```
int main() {
  IntSet is(1);
  is.insert(0);
  is.insert(1);   //grow(), 1 copy
  is.insert(2);   //grow(), 2 copies
  //...
  is.insert(n-2); //grow(), n-2 copies
  is.insert(n-1); //grow(), n-1 copies
}
```

- How many copies?
- $1 + 2 + ... + (n-2) + (n-1)$

Upper bound

```
int main() {
  IntSet is(1);
  int n = /*...*/;
  for (int i=0; i<n; ++i)
    is.insert(i);
}
```

- How many copies?
- 1 + 2 + ... + (n-2) + (n-1)

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- = (n-1)(n)/2 = quadratic function

- This is called a tight upper bound on the number of copies over the lifetime of this function
- This means that as the size of the `IntSet` grows (`n`), the cost to build the `IntSet` grows much faster (n^2)

Optimizing `grow()`

- How can we make `grow()` better?

- Let's bet that we'll need more space in the future
- Before: we copy `n` things, but only buy room for one more slot
- After: copy `n` things, and buy room for `n` more slots

- The new version is only **slightly** different from the old version
- However, it has **very** different performance characteristics

Optimizing `grow()`

```
void IntSet::grow() {
  int *tmp = new int[elts_capacity * 2];
  for (int i = 0; i < elts_size; ++i) {
    tmp[i] = elts[i];
  }
  delete [] elts;
  elts = tmp;
  elts_capacity *= 2;
}
```

Instead of growing the array by one, we double it

Optimizing `grow()`

- When is `grow()` called?
- How many array elements are copied?

```
int main() {
  IntSet is(1);
  int n = /* some large number */;
  for (int i=0; i<n; ++i) {
    is.insert(i);
  }
}
```

Optimizing `grow()`

```
int main() {
  IntSet is(1);
  int n = /*...*/;
  for (int i=0; i<n; ++i)
    is.insert(i);
}
```

- Let's unroll this loop:

```
int main() {
  IntSet is(1);
  is.insert(0);
  is.insert(1);     //grow(), 1 copy, capacity=2
  is.insert(2);     //grow(), 2 copies, capacity=4
  is.insert(3);
  is.insert(4);     //grow(), 4 copies, capacity=8
  //...
  is.insert(n/4-1); //grow(), n/4-1 copies
  //...
  is.insert(n/2-1); //grow(), n/2-1 copies
  //...
  is.insert(n-1);   //grow(), n-1 copies
}
```

Optimizing `grow()`

```
int main() {
  IntSet is(1);
  int n = /*...*/;
  for (int i=0; i<n; ++i)
    is.insert(i);
}
```

- The total number of copies (`T`) is:

`T = (N-1)+(N/2-1)+(N/4-1)+(N/8-1)+...+ 2 + 1`

- We can drop the "-1" terms in each step:

`T < N + N/2 + N/4 + N/8 + ...`

Optimizing `grow()`

- Note that we will eventually terminate, since we can only copy an integral number of integers

- Start collapsing terms to see where we get:

```
T < N +  (N/2 + N/4) + N/8  + ...
  < N +  3N/4        + N/8  + ...
  < N + (3N/4        + N/8) + ...
  < N +  7N/8               + ...
```

Optimizing `grow()`

```
T < N +  (N/2 + N/4) + N/8  + ...
  < N +  3N/4        + N/8  + ...
  < N + (3N/4        + N/8) + ...
  < N +  7N/8               + ...
```

- In the worst case, this becomes:

```
  < N + (reallyBigNumber - 1)N/(reallyBigNumber)
```

- But `(reallyBigNumber-1)/reallyBigNumber` is almost 1, so:

```
T < N + N
  < 2N
```

- So, instead of copying almost (N^2-N)/2 elements, we copy fewer than 2N of them

Optimizing `grow()`

- There is a big difference between the two approaches!

# elements	(N^2-N)/2	2N
1	0	2
8	28	16
64	2016	128
512	130816	1024
2048	2096128	4096

- The "double" implementation is **much** better than the "by-one" implementation

Exercise: allocating classes

```
int main() {
  IntSet is1(1);
  is1.insert(42);

  IntSet is2(3);
  is2 = is1;
  is2.insert(43);
}
```

- Draw the stack and heap
- Hint: first label the lines where ctors, dtors, assignments, etc. happen