Ve 280

Programming and Introductory Data Structures

Destructor; Deep Copy

Outline

- Overloaded Constructor and Default Argument
- Destructor
- Shallow Copy versus Deep Copy
- Copy Constructor
- Assignment Operator

Review

- Build a new IntSet with dynamic array
- In addition to the default constructor, we have an alternate constructor

```
class IntSet {
  int *elts;    // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
 public:
  IntSet();   // default constructor
    // EFFECTS: create a MAXELTS capacity set
  IntSet(int size); // constructor with
                    // explicit capacity
    // REQUIRES: size > 0
    // EFFECTS: create a size capacity set
};
```

function overloading

Dynamic Arrays

Building a new `IntSet`

```
IntSet::IntSet(int size):
   elts(new int[size]),
   sizeElts(size),
   numElts(0) {
}
```

```
IntSet::IntSet():
   elts(new int[MAXELTS]),
   sizeElts(MAXELTS),
   numElts(0) {
}
```

- Notice that the two constructors are nearly identical:
  - The only difference is whether we use `size` or `MAXELTS`.
  - Otherwise the code is duplicated.
- This is bad: when we find ourselves writing the same code over and over, we should try to use parametric generalization.

Dynamic Arrays

Building a new constructor

- One way to solve this problem of duplicate definitions is to use **default argument**.
- We can define **just one** constructor, but make its argument **optional**.
- First, we have to re-declare the constructor in IntSet:

```
class IntSet {
  int *elts;    // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  IntSet( int size = MAXELTS );
    // EFFECTS: create a set with specified
    //    capacity.  It defaults to MAXELTS if
    //    not supplied.
};
```

Default Argument

- `int add(int a, int b, int c = 1)`
  - The default value of c is 1.

- Using default arguments allows you to call the function with different number of arguments.

```
add(1, 2) // a = 1,b =2,c = 1 (default value)
add(1, 2, 3) // a = 1, b = 2, c = 3
```

- There could be multiple default arguments in a function, but they must be the last arguments.

```
int add(int a, int b = 0, int c = 1) // OK
int add(in a, int b = 1, int c) // Error
```

Dynamic Arrays

Building a new constructor

- Then, we implement the constructor in a same way as before.

```
IntSet::IntSet(int size):
   elts(new int[size]), sizeElts(size),
   numElts(0)
{
}
```

Don’t add “**= MAXELTS**”!

Outline

- Overloaded Constructor and Default Argument
- Destructor
- Shallow Copy versus Deep Copy
- Copy Constructor
- Assignment Operator

Problem

- There is a problem with what we've built so far.
- What happens if we have a local `IntSet` inside of a function and the function returns?
- Answer: **Memory leak**! Because link to the `elts` array in `IntSet` is lost.

```
class IntSet {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  ...
};
```

Question

- Is this a problem with the "static" version of `IntSet`? Why?

```
void foo() {
   IntSet is2;
   // Do work with is2 in some way
}
```

```
class IntSet {
   int elts[MAXELTS];
   int numElts;   // current occupancy

public:
   ...
};
```

Dynamic Arrays

How to solve the leak

- To solve this memory leak, we have 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 a legal instance of its class and the destructor's job is to destroy the object.

- In a class where its methods (including the constructor) allocate **dynamic storage**, the destructor is responsible for **de-allocating** it.

The Destructor

```
class IntSet {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  IntSet(int size = MAXELTS);
    // EFFECTS: create a set with size capacity;
    //          capacity is MAXELTS by default.
  ~IntSet(); // Destroy this IntSet
  ...
};

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

Note that we have to use the array-based `delete` operator, not the "standard" `delete` operator

The Destructor

```
class IntSet {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  IntSet(int size = MAXELTS);
    // EFFECTS: create a set with size capacity;
    //          capacity is MAXELTS by default.
  ~IntSet(); // Destroy this IntSet
  ...
};

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

When the IntSet is destroyed, the elements in the array will first be deleted.

The Destructor

```
class IntSet {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  IntSet(int size = MAXELTS);
    // EFFECTS: create a set with size capacity;
    //          capacity is MAXELTS by default.
  ~IntSet(); // Destroy this IntSet
  ...
};
```

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

Note: the destructors for any ADTs declared locally within a block of code are called automatically when the block ends.

Dynamic Arrays

Dynamic `IntSet`

- The new definition of IntSet can be created/destroyed dynamically, just like anything else:

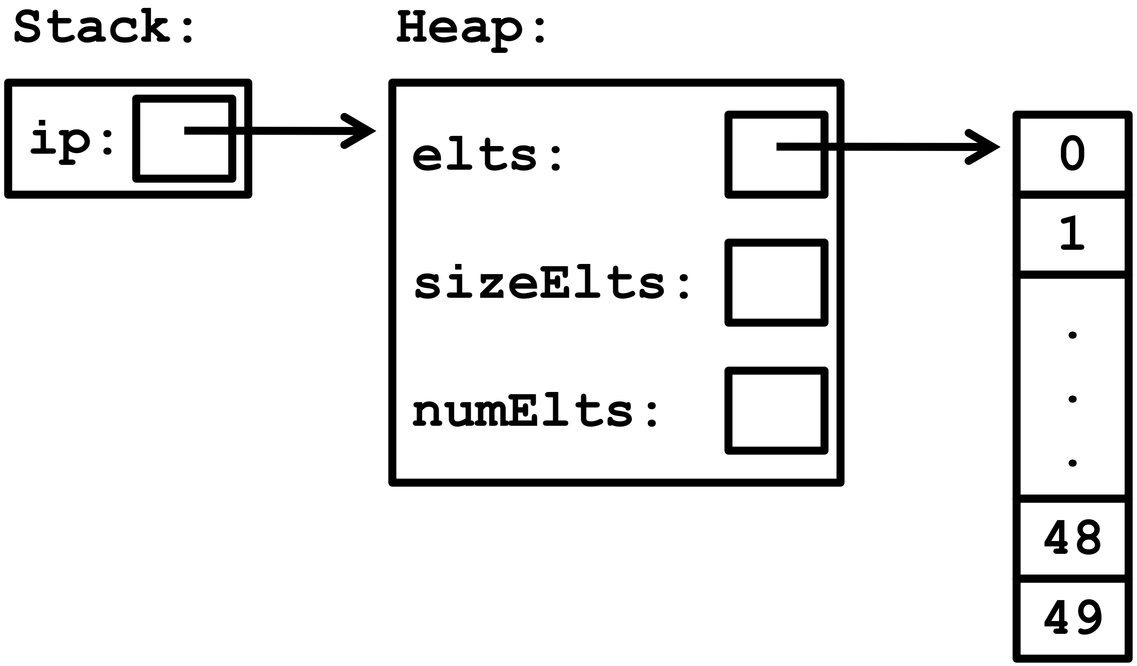
```
// a non-standard size
IntSet *ip = new IntSet(50);

... // do stuff

delete ip; // Destroys the IntSet.
```

```
IntSet *ip = new IntSet(50);
```

Dynamic Arrays

Dynamic `IntSet` creation

- After the `IntSet` pointer is created, we get:
  - Allocate space to hold the IntSet (a pointer and two integers)
  - Call the constructor on that object (allocates space for the array of 50 integers)

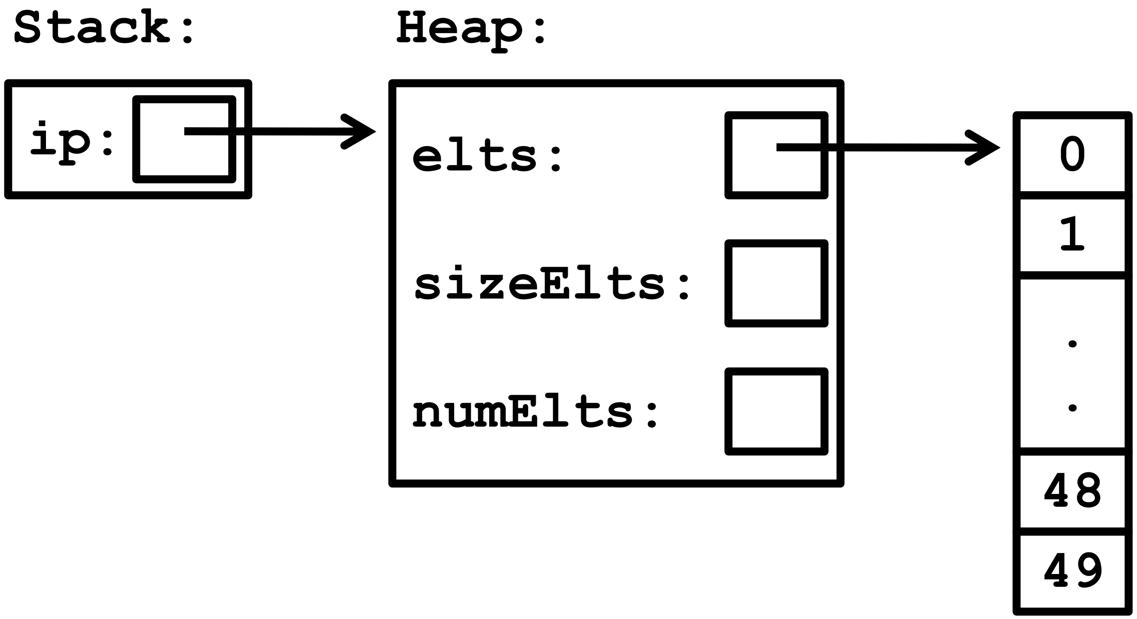


Dynamic Arrays

Dynamic `IntSet` deletion

```
delete ip;
```

- When you call delete on an instance of a class with a destructor
  - **First** the destructor is called (deallocates the array)
  - **Then** the object itself is deleted



Outline

- Overloaded Constructor and Default Argument
- Destructor
- Shallow Copy versus Deep Copy
- Copy Constructor
- Assignment Operator

Dynamic Arrays

Group Exercise

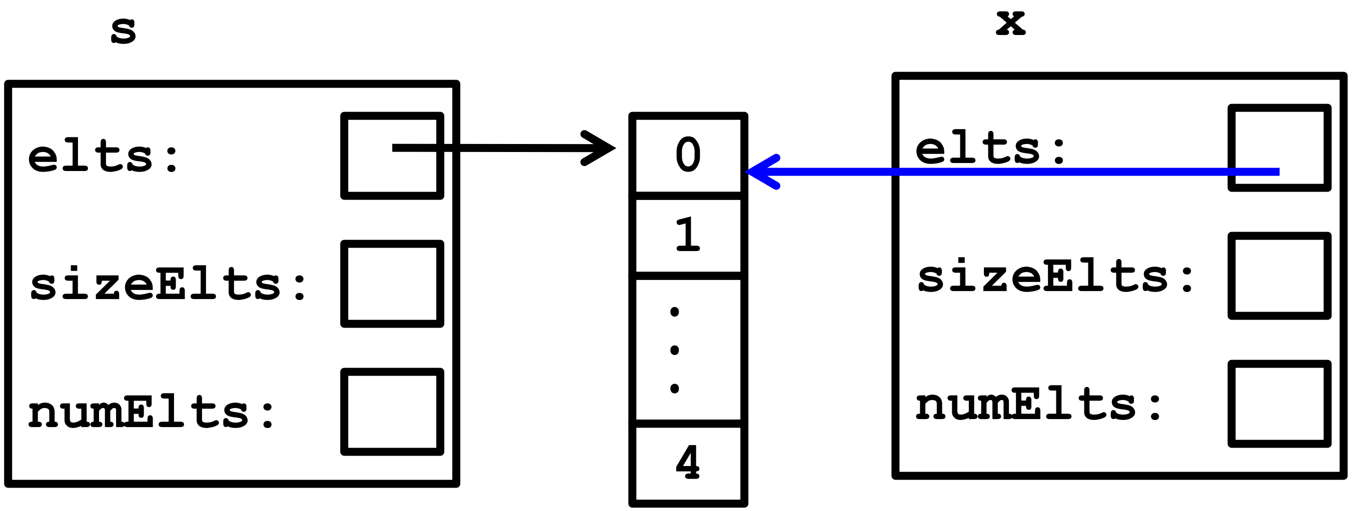
- **Question**: What happens in the following code?
- **Hint**: Classes are passed by-value, just like structs. They are also bitwise-copied, just like structs!

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

Dynamic Arrays

The problem of dangling pointers

- The result of pass-by-value mechanism: only pointer value of `elts` is copied, not the array `elts[]`.
  - The two objects end up sharing the same `elts[]` array!
- When `foo` finishes, `x` goes out of scope and is **destroyed**. As a result, `s.elts` **dangles**.
- When main finishes, the destructor of `s` is called. This causes double-deletion of `s.elts`.



Dynamic Arrays

The problem of dangling pointers

- It turns out that exactly the same thing happens in the following:

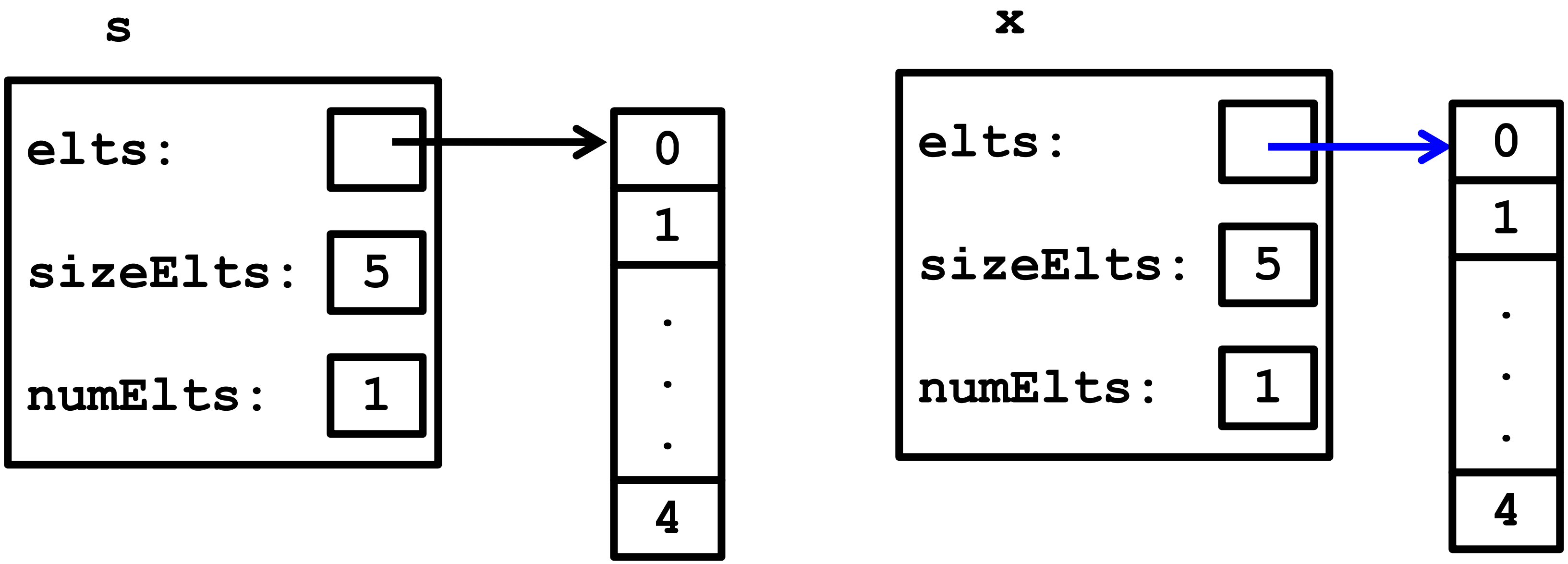
```
void foo() {
  IntSet s(5);
  s.insert(7);
  {
    IntSet x;
    x = s;
  }
  s.query(7); // Undefined!
}
```

- The assignment statement copies the elements of `s` to the elements of `x`, but they end up **sharing** the `elts` array. When `x` goes out of scope and is **destroyed**, `s.elts` **dangles**.

Dynamic Arrays

Fixing dangling pointers

- What we really want is to copy the entire **array**.



Dynamic Arrays

Fixing dangling pointers

- When a class contains pointers to **dynamic** elements, copying it is tricky.
- If we just copy the "members of the class", we get a **shallow copy**.
- Usually, we want a **full** copy of **everything**. This is called a **deep copy**.

Outline

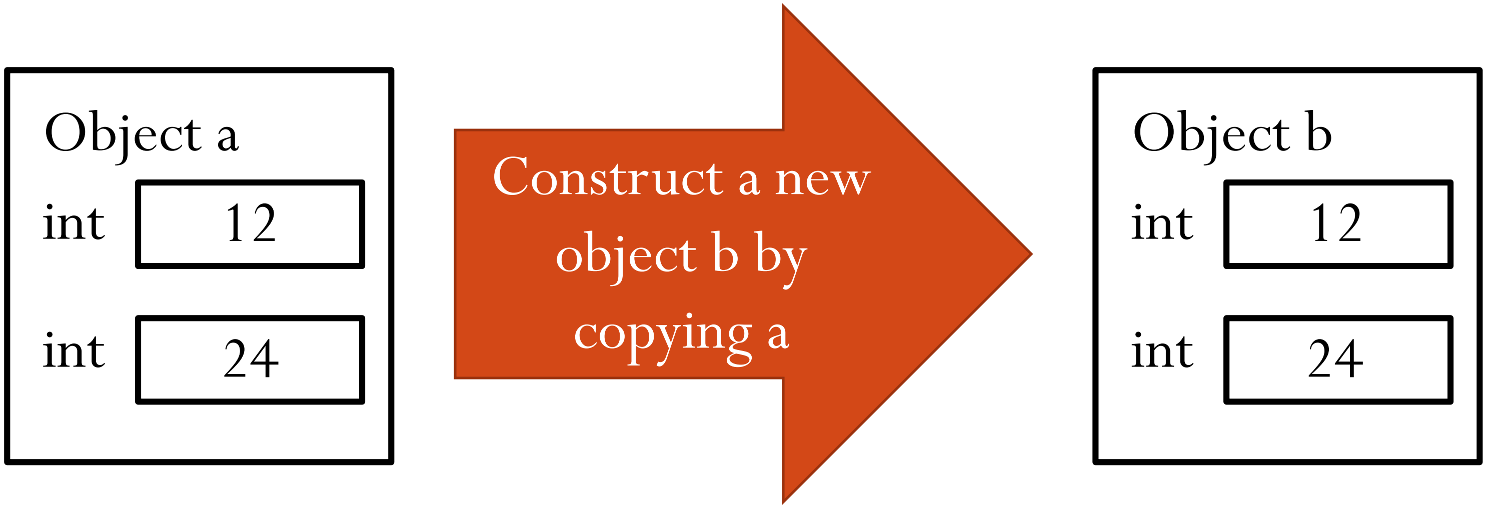
- Overloaded Constructor and Default Argument
- Destructor
- Shallow Copy versus Deep Copy
- Copy Constructor
- Assignment Operator

Fixing Dangling Pointers

- The C++ class mechanism provides two very closely related mechanisms that copy class objects:
  - **Copy constructor** and **assignment operator**.

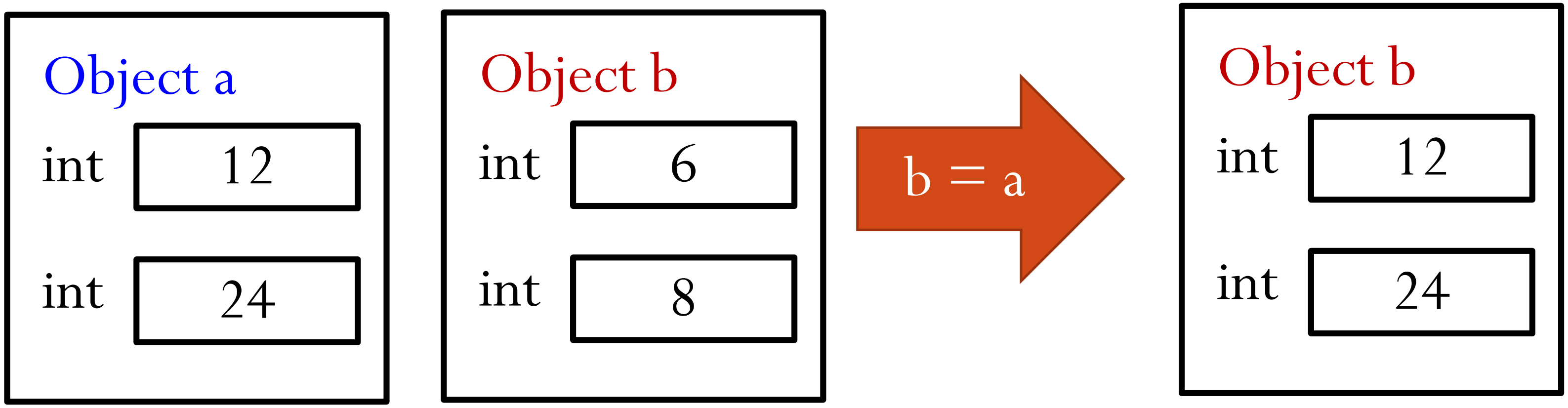
Fixing Dangling Pointers

- **Copy constructor** – it creates an object of this class by copying from another object of this class. In other words, given a "blank" block of memory, and an "example" instance, make the "blank" block a copy of the example.
  - The copy constructor plays a role identical to any other constructor.



Fixing Dangling Pointers

- **Assignment operator** – it copies the contents from one object (source) to another existing object (target).
  - Both two objects already exist.



Copy Constructors

- We could declare a copy constructor for our `IntSet` class as follows:

```
class IntSet {
  int *elts;     // array of elements
  int numElts;   // number of elements in array
  int sizeElts; // capacity of array
public:
  IntSet(int size=MAXELTS); // client optionally
                            // names size
  IntSet(const IntSet &is); // copy constructor
  ...
};
```

function overloading

Copy Constructors

```
IntSet(const IntSet &is);
```

- When passing arguments by value to a function, copy constructor is called.
  - The copy constructor is invoked on a “blank" instance of an `IntSet`, and must make the “blank" version look like an exact copy of the argument.

```
foo(s); //s is an IntSet
```

```
void foo(IntSet x) {
// copy constructor copies s to x
// do something
}
```

Copy Constructors

```
IntSet(const IntSet &is);
```

- The argument must be **passed by reference** to avoid infinite recursion.
- The argument is **const** for two reasons:
  1. Avoid accidentally changing the argument.
  2. Ensure that **any** instance (e.g., const object) of the class can serve as the **source**, not just a **mutable** one.

Copy Constructors

- The copy constructor has to accomplish the following tasks:
  1. Allocate an array of the same size as the source set's
  2. Copy each element from the source array to the new array
  3. Copy the numElts/sizeElts fields

- The copying part is going to have to happen in both the **copy constructor** and the **assignment operator**.
- So, we will abstract away the copying into a utility function.

Copy Constructors

- This adds a private method to our ADT:

```
class IntSet {
  int *elts;    // array of elements
  int numElts;  // number of elements in array
  int sizeElts; // capacity of array
  void copyFrom(const IntSet &is);
    // MODIFIES: this
    // EFFECTS:  copies is contents to this
public:
  IntSet(int size=MAXELTS); // client optionally
                            // names size
  IntSet(const IntSet &is); // copy constructor
  ...
};
```

Copy Constructors: Deep Copies

- Before implementing `copyFrom`, think about what has to happen in `copyFrom`, **in general**, not just in the context of the **copy constructor**.

- We need to figure this out because `copyFrom` will be called from the **assignment operator**.

Copy Constructors: Deep Copies

- `copyFrom` is a method and it must maintain the representational invariants. Here's what it must do:
  1. `copyFrom` has to assume that the source and destination sets might have different sizes. If so, it will have to resize the array appropriately, by **destroying** and **reallocating** it.
  2. Copy the source array to the destination array.
  3. Copy `sizeElts` and `numElts`.

Copy Constructors: Deep Copies

```
void IntSet::copyFrom(const IntSet &is) {
  if (is.sizeElts != sizeElts) { // Resize array
    delete[] elts;
    sizeElts = is.sizeElts;
    elts = new int[sizeElts];
  }
  // Copy array
  for (int i = 0; i < is.sizeElts; i++) {
    elts[i] = is.elts[i];
  }
  // Establish numElts invariant
  numElts = is.numElts;
}
```

Copy Constructors: Deep Copies

- With `copyFrom`, the copy constructor is simple.
- First, we have to establish its invariants, then call `copyFrom`.

```
IntSet::IntSet(const IntSet &is) {
  elts = NULL;
  numElts = 0;
  sizeElts = 0;
  copyFrom(is);
}
```

Copy Constructors: Deep Copies

- Contrast this copy constructor with the "default" method of copying, which does only a few things:
  - Copies the elts/numElts/sizeElts fields

- The copy constructor we've written **chases** 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 of a **shallow copy**.

Outline

- Overloaded Constructor and Default Argument
- Destructor
- Shallow Copy versus Deep Copy
- Copy Constructor
- Assignment Operator

Assignment Operators

Basics

- Assignment statement returns a value.
- The value is the **reference** to its left-hand-side object.

- Example

```
x = 4;
(y = x) += 2;
```

  - Are the above statements legal?
  - What is the value of `y`?

Assignment Operators

Basics

- Assignment statements can be “chained”. The following is legal in C++:

```
 x = y = z;
```

- This is a compound expression. Assignment operators binds **right-to-left**.
- Because "=" binds right-to-left, we first assign `z` to `y`, and this expression yields the (new) value "`y`" so that it can in turn be assigned to `x`.

Assignment Operators

On to overloading

- Now, how do we handle the following code?

```
IntSet s1(5);
IntSet s2(10);
s1 = s2; // assignment of s2 to s1
```

- By default, the compiler will use a shallow copy for the this.
- However, like a copy constructor, assignment must do a **deep copy** of the right-hand-side to the left-hand-side.
- To implement this, we **redefine** the "assignment operator" for `IntSets` by doing **operator overloading**.

Assignment Operators

Operator overloading

- Here's how we **overload** the assignment operator:

```
class IntSet {
   // data elements
   ...
   public:
   // Constructors
   ...
   IntSet &operator= (const IntSet &is);
   ...
};
```

You can overload other operators such as +, *, etc. You need to use the keyword **operator**

Assignment Operators

Operator overloading

```
IntSet &operator= (const IntSet &is);
```

- Like the copy constructor, the assignment operator takes a **reference to a const** instance to copy from.
- However, it also **returns** a **reference** to the copied-to object.

- When we call the assignment operator

```
    a = b;
```

- Essentially, we call the assignment operator of object `a`.
- `b` is the argument to the `operator=()` function.
  - Consider this as `a.operator=(b)`

Assignment Operators

Operator overloading

- The cool thing is that we have written `copyFrom` already:

```
void IntSet::copyFrom(const IntSet &is) {
  if (is.sizeElts != sizeElts) { // Resize array
    delete[] elts;
    sizeElts = is.sizeElts;
    elts = new int[sizeElts];
  }
  // Copy array
  for (int i = 0; i < is.sizeElts; i++) {
    elts[i] = is.elts[i];
  }
  // Establish numElts invariant
  numElts = is.numElts;
}
```

Assignment Operators

Operator overloading

- With `copyFrom`, the assignment operator is (almost) trivial:

```
IntSet &IntSet::operator= (const IntSet &is) {
  copyFrom(is);
  return *this;
}
```

Note: Every method has an implicit local variable "this", which is a pointer to the current instance on which that method operates.

Assignment Operators

Operator overloading

- With `copyFrom`, the assignment operator is (almost) trivial:

```
IntSet &IntSet::operator= (const IntSet &is) {
  copyFrom(is);
  return *this;
}
```

Note: This line dereferences that pointer and then returns a reference to it. We can't just return "this", because "this" is just a pointer, cannot be used as a reference.

Assignment Operators

Operator overloading

- With `copyFrom`, the assignment operator is (almost) trivial:

```
IntSet &IntSet::operator= (const IntSet &is) {
  copyFrom(is);
  return *this;
}
```

Note: We must return the reference to the **assigned-to** object, not the **assigned-from** object, i.e., we cannot `return is`.

Assignment Operators

Question

- **Question**: What happens if we do this?

```
IntSet s(50);
s = s;
```

- It is fine! Since their `sizeElts` are equal, no destroying and reallocating are needed.
- However, it is better to modify the code as follows:

```
IntSet &IntSet::operator= (const IntSet &is)
{
    if(this != &is)
        copyFrom(is);
    return *this;
}
```

The Rule of the Big Three

- What we have talked so far can be summarized with a simple rule: **the Rule of the Big Three**.

- Specifically, if you have any **dynamically allocated storage** in a class, you must provide:
  - **A destructor**
  - **A copy constructor**
  - **An assignment operator**

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

Reference

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 11.4 Classes and Dynamic Arrays