CSCI 262
Data Structures

13 – The "Big 3"

CS@Mines

What are...

THE BIG 3

CS@Mines

The Big 3

Three (optional) methods for your class:

- Copy constructor: creates copies of object
  - When passing by value
  - When used in variable initializer
- Assignment operator: copies object over existing object in assignment
- Destructor: called when object goes out of scope or is deleted

C++ provides default behaviors for each of these…
(but we'll want to override the defaults!)

CS@Mines

Copy Constructor

Used to create a *new* object as a copy of another:

```
foo x;
foo y(x);       // copy of x
```

or:

```
foo y = x;      // also uses copy
                // constructor, not
                // assignment operator
```

Also called when:

- Passed by value into function
- Returned (by value) from function

CS@Mines

Assignment Operator

Used when assigning using *existing* objects:

```
foo x, y;
x = y;
```

CS@Mines

Destructor

Applied automatically when:

- Object goes out of scope
- Object is deleted

E.g.,

```
foo *p = new foo;
delete p;   // *p is deleted
```

Or

```
while (true) {
      foo f;
      break;
}
// f is now out of scope
```

CS@Mines

Default Behavior: Copy and Assignment

Simply copies instance variables…

```
class foo {
public:
      int n;
};

foo x;
x.n = 42;
foo y(x);    // y.n also now equals 42
x.n = 17;
y = x;       // y.n also now equals 17
```

This is typically the behavior we want! However…

Default Behavior with Dynamically Allocated Memory: Copy

Example:

```
class number {
public:
      number(int n) { ptr = new int(n); }
private:
      int* ptr;
};

number x(42);
number y = x;
```

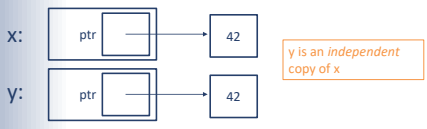
Problem: we only copied the *pointer* – x and y now "share" memory

Default Behavior with Dynamically Allocated Memory: Copy

Example, illustrated:

```
number x(42);
number y = x;
```

What we want to happen:

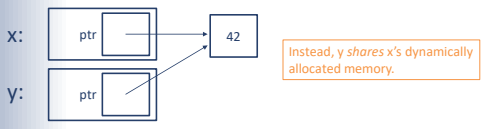


Default Behavior with Dynamically Allocated Memory: Copy

Example, illustrated:

```
number x(42);
number y = x;
```

What actually happens:



Deep Copy

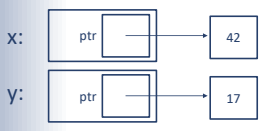
- The default behavior is called a *shallow copy*
- The behavior we want is called a *deep copy*
  - Copy memory pointed to by member pointer variables
    - Where appropriate – it isn't always correct to do so
    - May need to allocate/reallocate
  - Copy member non-pointer variables recursively

Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:

```
number x(42), y(17);
```

Initially:

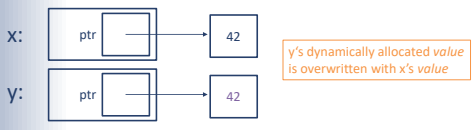


Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:

```
number x(42), y(17);
y = x;
```

What we expect:

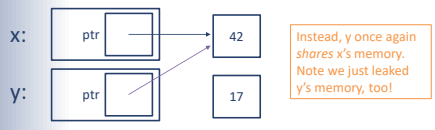


Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:

```
number x(42), y(17);
y = x;
```

What actually happens:



Default Behavior with Dynamically Allocated Memory: Destructor

The default destructor:

- Call destructors of each member variable
- Does nothing to primitive types (and pointers)

While this is generally appropriate, it will result in a memory leak for our number class.

Fixing the Defaults

We can override the defaults by defining our own copy constructor, destructor, and assignment operator:

```
class number {
public:
    number(int n) { ptr = new int(n); }
    number(const number& num);
    ~number();

    number& operator=(const number& num);
private:
    int* ptr;
};
```


Fixing the Copy Constructor

```
number::number(const number& num) {
    ptr = new int;
    *ptr = *(num.ptr);
}
```

Note: must be passed by reference! (Why?)

Step 1: *allocate* our own memory

Step 2: *copy* value (not pointer!)

Fixing the Assignment Operator

Similar to copy constructor… but different.

```
number& number::operator=(const number& num) {
    if (this == &num) return *this;  // self assignment

    *ptr = *(num.ptr);
    return *this;
}
```

Step 1: check for self-assignment
Step 2: allocate/de-allocate (if necessary)
Step 3: copy value
Step 4: return *this

Fixing the Destructor

Just need to clean up our memory…

```
number::~number() {
     delete ptr;
}
```


Adding the "Big 3" to

ARRAY LIST

Array List Class

```
class array_list {
public:
      array_list();
      int size();
      int get(int index);
      void set(int index, int val);
      …
private:
      int*  _arr;
      int   _size;
      int   _capacity;
      void  _resize();
};
```


Array List Class + Big 3

```
class array_list {
public:
      array_list();
      array_list(const array_list& lst);
      ~array_list();

      array_list& operator=(const array_list& lst);

      int size();
      int get(int index);
      void set(int index, int val);
      …
private:
      int*  _arr;
      int   _size;
      int   _capacity;
      void  _resize();
};
```


Array List: Copy Constructor

```
array_list::array_list(const array_list& src) {
      _capacity = src._capacity;     // same cap.
      _size = src._size;             // same size
      _arr = new int[_capacity];     // allocate
      for (int j = 0; j < _size; j++) {
            _arr[j] = src._arr[j];   // copy
      }
}
```


Array List: Assignment Operator

```
array_list& array_list::operator=(const array_list& src)
{
      if (this == &src) return *this;  // self-assign chk
      delete[] _arr;                   // clean up old
      _capacity = src._capacity;
      _size = src._size;
      _arr = new int[_capacity];       // allocate
      for (int j = 0; j < _size; j++) {
            _arr[j] = src._arr[j];     // copy
      }
      return *this;                    // return *this
}
```

Refactoring Opportunity

```
array_list& array_list::operator=(const array_list& src)
{
	if (this == &src) return;
	delete[] _arr;
	_capacity = src._capacity;
	_size = src._size;
	_arr = new int[_capacity];
	for (int j = 0; j < _size; j++) {
		_arr[j] = src._arr[j];
	}
	return *this;
}
```

CS@Mines

Array List Refactoring

```
void array_list::deep_copy(const array_list& src) {
	_capacity = src._capacity;
	_size = src._size;
	_arr = new int[_capacity];
	for (int j = 0; j < _size; j++) {
		_arr[j] = src._arr[j];
	}
}

array_list::array_list(const array_list& src) {
	deep_copy(src);
}

array_list& array_list::operator=(const array_list& src) {
	if (this == &src) return;
	delete[] _arr;
	deep_copy(src);
	return *this;
}
```

CS@Mines

Array List Destructor

```
array_list::~array_list() {
	delete[] _arr;
}
```

CS@Mines

Up Next

- Friday, October 26
  - Lab 9
  - APT 3 Due
  - Project 3 Assigned
- Monday, October 29
  - Templates

CS@Mines