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!)

Copy Constructor

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

```cpp
foo x;
foo y(x); // y is created as copy of x
```

or:

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

Also called when:
- Passed by value into function
- Returned (by value) from function

Assignment Operator

Used when assigning using existing objects:

```cpp
foo x, y;
x = y; // x already existed as an object of type foo
```

Destructor

Applied automatically when:
- Object goes out of scope
- Object is deleted

E.g.,

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

Or

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

Copy Default Behavior: with Dynamically Allocated Memory
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

Copy Default Behavior: with Dynamically Allocated Memory
Example, illustrated:
number x(42);
number y = x;

What we want to happen:

What actually happens:

Why Is This a Problem?

number x(42);
number y = x;

What if we had a setter for the value, and changed x (or y)?

Instead, y shares x’s dynamically allocated memory.

y is an independent copy of x.

y shares x’s dynamically allocated memory.
**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

**Assignment Default Behavior: with Dynamically Allocated Memory**
Example of assignment:
```
number x(42), y(17);
```
Initially:
```
x: ptr 42
y: ptr 17
```

What we expect:
```
x: ptr 42
y: ptr 42
```
y’s dynamically allocated value is overwritten with x’s value

What actually happens:
```
x: ptr 42
y: ptr 17
```
Instead, y once again shares x’s memory. Note we just leaked y’s memory, too!

**Destructor Default Behavior: with Dynamically Allocated Memory**
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.

**FIX IT!**
Fixing the Defaults

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

```cpp
class number {
public:
    number(int n) { ptr = new int(n); }
    number(const number& num); // copy constructor
    ~number(); // destructor
    number& operator=(const number& num); // assignment operator.
private:
    int* ptr;
};
```

Fixing the Copy Constructor

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

Step 1: copy non-pointer member variables
Step 2: allocate our own memory
Step 3: copy (not pointer!)

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

Fixing the Assignment Operator

Similar to copy constructor... but different.
```cpp
number& number::operator=(const number& num) {
    if (this == &num) return *this; // self assignment
    // no need to allocate/deallocate, we're the same size
    *ptr = *(num.ptr);
    return *this;
}
```

Step 1: check for self-assignment
Step 2: copy non-pointer member variables
Step 3: allocate/de-allocate (if necessary)
Step 4: copy value
Step 5: return *this

Fixing the Destructor

Just need to clean up our memory...
```cpp
number::~number() {
    delete ptr;
}
```

Array List Class

```cpp
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

```cpp
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; // copy non-ptr
    _size = src._size; // member vars
    _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;
}

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;
}

Array List Destructor

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

- Reading for Wednesday: Chapter 13.3
- Wednesday, April 3
  - Templates
  - Read Chapter 16 (16.5 is optional)
- Friday, April 5
  - Lab 11 – Queue, part 2
- Monday, April 8
  - Binary Trees
  - Lab 11 due