CSCI 262
Data Structures

17 – The “Big 3”

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
class foo {
public:
    int n;
};
foo x;
x.n = 42;
foo y(x);  // 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;
```

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 Behavior: Copy Constructor and Assignment Operator

Simply copies instance variables...

```cpp
class foo {
public:
    int n;
};
foo x;
x.n = 42;
foo y(x);  // y.n also now equals 42
x.n = 17;  // y.n also now equals 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:
```cpp
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:
- `x:`
  - `ptr` 42
- `y:`
  - `ptr` 42

But instead, y shares x's dynamically allocated int value.

Deep Copy

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

Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:
- `number x(42), y(17);`
- `y = x;`

Initially:
- `x:`
  - `ptr` 42
- `y:`
  - `ptr` 17

What we expect:
- `x:`
  - `ptr` 42
- `y:`
  - `ptr` 42

Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:
- `number x(42), y(17);` 
- `y = x;`

What we expect:
- `x:`
  - `ptr` 42
- `y:`
  - `ptr` 42
Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:
number x(42), y(17);
y = x;

What actually happens:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x:</td>
<td>ptr</td>
<td>42</td>
</tr>
<tr>
<td>y:</td>
<td>ptr</td>
<td>17</td>
</tr>
</tbody>
</table>

Note we just leaked memory, too!

Default Behavior with Dynamically Allocated Memory: Destructor

The default destructor:
– Call destructors of each member var
– 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);
}

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;
}
Array List Class

class array_list {
    public:
        array_list();
        int size();
        int get(int index);
        int 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;
    _size = src._size;
    _arr = new int[_capacity];
    for (int j = 0; j < _size; j++) {
        _arr[j] = src._arr[j];
    }
}

Refactoring Opportunity

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

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