CSCI 262
Data Structures

12 – 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:
- 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

Assignment Operator
Used when assigning using existing objects:
- foo x, y;
- x = y;

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
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.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:

x:
  ptr [value: 42]
y:
  ptr [value: 42]

y is an independent copy of x

What actually happens:

x:
  ptr [value: 42]
y:
  ptr [value: 42]

Instead, 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

---

Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:

number x(42), y(17);

Initially:

x:
  ptr [value: 42]
y:
  ptr [value: 17]
Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:

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

What we expect:

- `x`: value 42
- `y`: value 42

Default Behavior with Dynamically Allocated Memory: Assignment

Example of assignment:

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

What actually happens:

- `x`: value 42
- `y`: value 17

Instead, `y` once again shares `x`’s memory. Note we just leaked `y`’s memory, too!

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:

```cpp
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

```cpp
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.

```cpp
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...

\begin{verbatim}
number::~number() {
    delete ptr;
}
\end{verbatim}

Array List Class

\begin{verbatim}
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();
};
\end{verbatim}

Array List Class + Big 3

\begin{verbatim}
class array_list {
    public:
        array_list();
        array_list(const array_list& lst); // same cap.
        ~array_list();
        array_list& operator=(const array_list& lst); // same size
        int size();
        int get(int index);
        void set(int index, int val);
    
    private:
        int* _arr;
        int _size;
        int _capacity;
        void _resize();
};
\end{verbatim}

Array List: Copy Constructor

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

Array List: Assignment Operator

\begin{verbatim}
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++) { // copy
        _arr[j] = src._arr[j];
    }
    return *this; // return *this
}
\end{verbatim}
**Refactoring Opportunity**

```cpp
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**

```cpp
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];
    }
}
```

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

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

**Array List Destructor**

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

**Up Next**

- **Wednesday, March 7**
  - Templates
  - Read chapter 13
- **Friday, March 9**
  - Lab 8 – Operator Overloading
  - APT 4 due
  - Project 4 assigned