CS 246 - Object Oriented Programming **Fall 2016**

Lecture 5: October 6, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

Note: *(E)* Indicates topics Brad has emphasized and should receive more attention

Example 5.1 *Deepcopy Copy Constructor*

```
1 struct Student{
2     int data
3     Node *next;
4     Node (const Node &other) : data {other, data}, next{other.next?new Node(*other.next):
          nullptr}{}
5 };
```

5.1 Copy Constructor Use Cases *(E)*

1. When an object initializes another object
2. When an object is passed by value
3. When an object is returned by value

5.2 Uniform Initialization

Example 5.2 *In C++ 03*

```
1 int x = 5;
2 string s = "Hello";
3 Student  billy (70,80,90);
4 int x(5);
5 string s ("Hello");
```

Example 5.3 *C++14 Brace Bracket Syntax*

```
1 int x{5};
2 string s{"Hello"}
3 Student billy {70, 80, 90};
```

- *Brace brackets eliminate the need to know which initialization syntax need to be used.*
- *Brace bracket syntax works in most cases, but some cases still requires initialization using = or ()*

Note:

- Avoid single argument constructors, as they will cause the compiler to undergo implicit conversion.
  - For example, you could accidentally provide an integer into a function requiring a node, which the compiler will allow
- To resolve these issues, disable the implicit conversion by making the constructor explicit.
  - Add `explicit` in front of the constructor definition *(E)*

5.3 Destructors

When a object is destroyed there is a a method called the destructor that runs.

5.3.1 Sequence of Destructor

1. Destructor runs
2. Fields destructors are invoked (if there are objects) in reverse declaration order
3. Space deallocated

5.3.2 Custom Destructor

Classes come with a destructor, but only does step 2, which is fine for most cases. Despite this, there may be cases such as, linked lists, where you may need to write your own destructor.

Example 5.4 *Standard Destructor on pointer based Linked-List*

```
1 Node *np= new Node{1, new Node{2, new Node {3, nullptr}}}
```

- *If* `np` *goes out of scope : the pointer (*`np`*) is reclaimed and the entire list is leaked*
- *If delete is called : only the second node is deleted and the remained list is still leaked.*

Example 5.5 *Custom Deep Delete Destructor for Linked Lists (E)*

```
1 struct Node{
2     ...
3     ~ Node () {delete next;}
4 }
5
6 delete np;
```

- *The ~ (Tilde) before a function name that matches the structure name is treated as the destructor.*

5.4 Copy Assignment Operator

```
Student billy {60,70,80};
Student jane = billy; // Copy Constructor
Student anne ; // Default constructor
anne = billy ; // Copy Assignment not Constructor
```

Assigning a value after a structure has been defined utilizes a copy assignment constructor.

5.4.1 Custom Assignment Operator

Once again, you may need to occasionally write your own assignment operator. A good example of when this may happen, is when you are dealing with pointers.

Example 5.6 *Custom Assignment Operator for Linked-Lists*
Incorrect Implementation :

```
struct Node {
    Node &operator=(const Node &other){
        data = other.data;
        delete next; //delete existing nodes to prevent leaks
        next = other.next ? Node {* other.next} : nullptr;
        return *this;
    }
}; //Dangerous
```

The implementation fails because self assignment `n=n` *results in original values being deleted before copy can take place, thus you may get undefined behavior.*

Somewhat Correct Implementation :

```
struct Node {
    Node &operator=(const Node &other){
        if (this == &other) return *this;
        data = other.data;
        delete next; //delete existing nodes to prevent leaks
        next = other.next ? Node {* other.next} : nullptr;
        return *this;
    }
}; //Ok
```

This implementation is incorrect because of new node. If new node fails, then the function will stop and result in a corrupted structure.

Correct Implementation :

```
struct Node {
    Node &operator=(const Node &other){
        if (this == &other) return *this; // Not needed
        Node *temp = next;
        next = other.next ? Node {* other.next} : nullptr;
        data = other.data;
        delete tmp;
        return *this;
    }
}; //Correct
```

The correct implementation makes a temporary copy, so if new fails, the resulting structure will simply be the starting structure instead of a corrupted structure. This also eliminates the need for a self assignment check, unless you which to improve efficiency in rare cases.

- *The custom operator returns a Node, so the user can cascade if he/she wishes to.*
- ***Important :*** *The Assignment operator is not a* ***Constructor****, so there is no initialization list.*

Example 5.7 *(Copy and Swap idiom) Alternative Solution to Custom Assignment Operator (E)*

```
#include <utility>

struct Node {
    ...
    void swap (Node &other){
        using std::swap;
        swap(data, other.data);
        swap(next, other.next);
    }
    Node &operator= (const Node &other){
        Node tmp = other
        swap (tmp);
        return *this;
    }
}; // Alot Cleaner
```

- *This works because a copy of old data is created and the new data is swapped in, but since temp is on the stack, temps destructor runs and destroys old data.*

5.5 Rvalues and Rvalue reference

Recall:

- An lvalue is anything with an address
- lvalue references is like a constant pointer with auto dereferencing which is always initialized to an lvalue

Consider:

```
Node n {1, new Node { 2, nullptr}};
Node m = n; // Copy Constructor
Node m2;
m2 = n; // copy assignment
```

Example 5.8 *Deepcopy from Temporary Linked List.*

```
Node plusOne (Node n){
    for (Node *p = &n; p ; p = n->next){
        ++p->data;
    }
    return n;
}

Node m3 = plusOne(n); //Copy Constructor
```

- *The Compiler creates a temporary object to hold the result of plusone.*

- *other is a reference to this temporary*
  - *Copy constructor copies data from this temporary.*
- *The temporary is just going to be discarded anyway, as soon as the statement is evaluated completed.*
- *This is wasteful copying from a temporary, why not just **steal** it?*

Rvalue, and Move Constructors to the rescue!!! : *(E)*
In order to "steal" data from a temporary object instead of simply copying the values and deleting the temporary, you need to be able to differentiate between live and temporary object. C++ 11 introduces rvalues to resolve this.

C++ rvalue reference `Node &&` is a reference to a temporary object (An Rvalue) of type node.

In order to achieve this, a secondary constructor called a **Move** Constructor;

Example 5.9 *Move Constructor for Linked Lists. (E)*

```
struct {
    ...
    Node(Node &&other) : data{other.data}, next{other.next}{
        other.next = nullptr; // Takes away other value and assigns null
    }
};

Node m3 = plusOne(n);
```

- *The other next value must be set to null to prevent the "stolen" data from being deleted once the initial statement is evaluated*
- *This is efficient because if a initialization is done on a temporary object. Assignment can be done is constant time by "stealing" data.*

Example 5.10 *Move Assignment Operator Example (E)*

```
struct Node{
    ...
    Node &operator=(Node &&other){
    //Steal Others Data and Destroy My Data
    swap(other);
    return &this;

    }
}
```

- *The current objects data will be swapped with the temporary object. The temporary object will then be destroyed once the evaluation is completed, thus deleting the old data*

Note : *(E)*

If you don't define a move constructor or assignment, the original copy versions will be used instead. If the move operations are defined, they replace all calls to the copy constructor/assignment operator when the argument is a temporary rvalue

5.6 Copy/Move Elision *(E)*

Consider:

```
1 Vec makeAvec(){
2     return {0,0}; // Invokes basic constructor
3 }
4
5 Vec v = makeAvec();
```

The above snippet does not use the Move/Copy constructor. This happens because in certain circumstances the compiler is allowed to skip calling copy/move constructors **(But does not have to)**

In the snippet above, makeAvec writes its result `{0,0}` directly into the space accepted by v in the caller, rather than copy it after

Example 5.11 *Additional Example*

```
1 void doSomething(Vec v){ // Pass By Value
2     ...
3 }
4 doSomething(makeAVec());
```

- *Result of makeAVec is written directly into the parameter and not copied*
- *This is allowed, even if dropping the constructor calls would change change the behavior of the program (i.e Printing something)*

Note :

You are not expected to know exactly when copy/move elison is allowed - just that they are possible

If you need all of the constructors to run, compile with the `-fno-elide-constructors` flag. Keep in mind, that this could considerably, slow down your program.

5.7 In Summary: Rule of 5 *(E)*

- If you need a custom version of any of the following
  - Copy Constructor
  - Copy Assignment Operator
  - Destructor
  - Move Constructor
  - Move Assignment Operator

  Then you usually, need a custom version of all 5.

Note

- Operator= is a member function not a standalone function
- When an operator is declared as a member functions, THIS plays the role of the first argument
- Always define >> and << operators as standalone functions

The following operators should be members : ***(E)***

- operator *
- operator []
- operator →
- operator ()
- operator T (Where T is a type)

5.8 Separate Compilation with Classes

Node.h :

```
1 ...
2 struct Node {
3     int data;
4     Node *next;
5     explict Node (int data, Node * next = nullptr);
6     bool barNext ();
7 };
8 ...
```

Node.cc

```
1 #include "node.h"
2
3 Node::Node(int data, Node * next) data{data}, next{next} {}
4 Node::barNext () { ...}
```

Note : `::` is called the scope scope resolution operator