# CS 246 Fall 2016 - Tutorial 5

October 14, 2016

## 1 Summary

- Rvalues and Lvalues
- Constructors
- Rule of Five

## 2 Tip of the Week: .vimrc

The /.vimrc file contains a list of commands which will run each time vim is opened. It can be useful to place commands in the .vimrc file to set user preferences.

Potential useful commands:

- set number: turns on line numbers
- set expandtab: pressing tab is replaced with spaces
- set tabwidth=n: displays and types tabs with width n
- set shiftwidth=n: set width of auto-indent tools
- set smartindent: indents based on the type of the file

You can also set the colour scheme for vi using your .vimrc file.

## 3 Rvalues and Lvalues

- An lvalue is any object which has an address. They get their name because an lvalue is a value which can occur on the left side of an expression.
- An lvalue reference: &
- An rvalue anything which is not an lvalue. They get their name because an rvalue can only occur on the right side of an expression.
- An rvalue reference: &&

## 4 Constructors

- When working with C, when you wanted to program a class, you would typically write a separate function to allocate memory for the object and initialize the fields to be logical default values.
- In C++, we will instead write constructors. A constructor is a special method which allocates the memory for a class and (potentially) initializes the fields of the object.

  Example:

```
struct Vec{
  int x;
  int y;
  Vec(int x, int y);
};
```

A constructor will always be defined as `ClassName( parameters );`

- Note that we can overload the constructor. In our example above it is we could also add in `Vec(int x)` if we desire. We can also give the parameters default values.
- A constructor with no parameters is the default constructor for the class. This is the constructor which is called when we have `Vec v;`
- Notice that constructors do not have a return type.
- If we do not write a constructor, the compiler produces a default constructor and C-style initialization list. The default constructor calls the default constructor fields which are classes and leaves primitive types uninitialized. We lose both these constructor if we define a constructor.

# 5 Deep Copy

- How would we write the constructors in the following class?

```
struct Node{
  int value;
  Node* next = nullptr;
  Node* prev = nullptr;

  Node(int value);          // create a list with one node

Node(int begin, int end); // create a list with all values from begin
// to end in order

void add(int insert);     // add insert as the last node of the list
};


Node(int value): value{value}{}

Node(int begin, int end): value{begin}{
int dir = 1;
if ( begin > end ){
dir *= -1;
}
while ( begin != end ){
begin += dir;
add(begin);
}
}

void Node::add(int insert){
  if ( next ){
    next->add(insert)
  } else {
    next = new Node{insert};
    next->prev = this;
  }
}
```

The way the function is currently implemented, we will get a memory leak. Why?

## 5.1 Destructor

- The destructor is a function which is called when a object is destroyed. A destructor is provided for us by the compiler. This destructor will delete all fields of the object. However, it does not call delete on pointers which are fields.
- This means that in our example above, the data which next points at will not be freed when the Node is destroyed.
- We can write our own destructor for a class. For the Node class:

```
~Node(){
  delete next;
}
```

- Why are we not deleting prev here?
- Why don't we set next to nullptr?
- Note: similar to constructors, destructors do not have a return type.

## 5.2 Copy Constructor

- What happens when we run the following code?

```
Node* n1 = new Node{7};
n1->add(3);
n1->add(5);
n1->add(10);
Node n2 = *n1;

delete n1;
cout << n2;
```

- When running this code, we see that b2 now has garbage values in some of it's nodes. Why?
- The pointers are now dangling pointers because the pointers were deleted when we deleted n1.
- By default, the copy constructor copies all fields. This means that when pointers are copied, the address stored in the first pointer is copied to the new pointer and the objects share memory.
- We do not want these objects to be sharing memory. This means that we should implement our own copy constructor.

```
Node( const Node& n ): value{n.value}, next{ n.next ? new Node{*(n.next)}}{
  if ( next )
    next->prev = this;
}
```

- The copy constructor is called in three situations: when initializing an object from another object, when passing an object by value, and when returning by value.

## 5.3 Copy Assignment Operator

- What happens when we run the following code?

```
Node* n1 = new Node{7};
n1->add(3);
n1->add(5);
n1->add(10);
Node* n2 = new Node{8};
n2->add(12);
*n2 = *n1;

delete n1;
cout << n2;
```

- Similar to the copy constructor, we get garbage values printed. This is for a similar reason. The compiler gives a default version of the assignment operator but this method also does not perform a deep copy.

```
Node& operator=( const Node& n ){
  if ( this != &n ){
    Node copy = n;

    std::swap(next, copy.next);

    value = copy.value;
    next->prev = this;
  }
  return *this;
}
```

- The way this assignment operator is written it referred to the copy-and-swap idiom. We are first creating a local copy of the node we are copying. This calls the copy constructor. We then swap our next pointer with the next pointer of the Node copy. This means that when copy goes out of scope, our old data will be deleted by the destructor.

## 5.4 Move Copy Constructor

- Suppose we have the following function:

```
Node add(Node n, int inc){
  for ( Node* m = &n; m != nullptr; m = m->next) {
    m->value += inc;
  }
  return n;
}
```

- When we run this function, the copy contructor will be run to make a copy of the node which we pass as a parameter. It then returns a temporay object.
- Now, if we have something like `Node n2 = plus(n1, 2)`, what happens? The copy constructor will run to create n2 from the object returned from plus. However, this is a temporary object (rvalue) which will be destroyed as soon as we are done copying it.
- Idea: we should steal the data which is about to be thrown away instead of creating a copy of the data. How? Since we know we have an rvalue, we should write a copy constructor which takes an rvalue.

```
Node( Node&& n ): value{n.value}, next{n.next} {
  if ( next ){
```

```
    next->prev = this;
  }
  n.next = nullptr;
}
```

- Important note: we must set all pointers which will be deleted by the destructor to be `nullptr` or the destructor will delete the data we stole when the temporary object goes out of scope.

### 5.5 Move Assignment Operator

- Similar to the move copy constructor, we may want to have the following:

```
Node n1{3,7};
Node n2{1,9};

n2 = plus{n1, 2};
```

- We want to make an assignment operator which take rvalues as well.

```
Node operator=( Node&& n ) {
  value = n.value;
  swap(n.next, next);
  next->prev = this;
}
```

## 6 Rule of Five

If you have to write one of the copy constructor, move copy constructor, copy assignment operator, move assignment operator or destructor, you most likely have to write all five.

Why? You typically have to write each if you are dealing with non-contiguous memory.