CS 32 Week 2
Discussion 2E

Srinath

Announcements

- Homework 1 is Up!
- Due : 11:00 PM Tuesday, January 24

Outline

- Destructor
- Copy Constructor
- Assignment Operator

Destructor

Destructor

Destructor

Destructor

Why do we need a Destructor?

Destructor

Why do we need a Destructor?

- To free up resources used by our class
- Our code should not have memory leaks, dangling pointers

Destructor

Definition

```
class String {
    ....
    ...?
}
```

Destructor

Definition

```
class String {
    ....
    ~String();
}

String::~String() {
    // Destructor body
    ....
}
```

Destructor

Definition

```
class String {
    ....
    ~String();
}

String::~String() {
    // Destructor body
    ....
}
```

What if we don't define one?

Destructor

Definition

```
class String {
    ....
    ~String();
}

String::~String() {
    // Destructor body
    ....
}
```

What if we don't define one?

- A default destructor with empty body is created by the compiler

Can we define destructor with parameters?

Destructor

Definition

```
class String {
    ....
    ~String();
}

String::~String() {
    // Destructor body
    ....
}
```

What if we don't define one?

- A default destructor with empty body is created by the compiler

Can we define destructor with parameters?

- No!, we don't need them

Can we define multiple destructors?

Destructor

Definition

```
class String {
    ....
    ~String();
}

String::~String() {
    // Destructor body
    ....
}
```

What if we don't define one?

- A default destructor with empty body is created by the compiler

Can we define destructor with parameters?

- No!, we don't need them

Can we define multiple destructors?

- No! Every class/struct must have one and only one.

Destructor : Order of Destruction

??

Destructor : Order of Destruction

Is reverse order of construction

Destructor : Order of Destruction

Is reverse order of construction

1. Run body of the destructor
2. Destroy data members in reverse order (i,e call their respective destructor's)
3. —--------

Destructor : Order of Destruction

Is reverse order of construction

1. Run body of the destructor
2. Destroy data members in reverse order (i,e call their respective destructor's)
3. —--------

```
class Geometry{
   public:
      Circle circle;
      Triangle triangle;
      Rectangle rectangle;
}
```

In which order the data member's destructor are called?

Destructor : Order of Destruction

Is reverse order of construction

1. Run body of the destructor
2. Destroy data members in reverse order (i,e call their respective destructor's)
3. —--------

```
class Geometry{
   public:
      Circle circle;
      Triangle triangle;
      Rectangle rectangle;
}
```

In which order the data member's destructor are called?

- 1. rectangle, 2. triangle, 3. circle

Destructor

When is a destructor called?

Destructor

When is a destructor called?

1. When object goes out of scope

Destructor

When is a destructor called?

1. When object goes out of scope

```
1.  Void h(...){
2.      ….
3.      Rectangle rectangle(width, height);
4.      ….
5.  }
```

Destructor of rectangle is called at? (line no.)

Destructor

When is a destructor called?

1. When object goes out of scope

```
1.  Void h(...){
2.      ….
3.      Rectangle rectangle(width, height);
4.      ….
5.  }
```

Destructor of rectangle is called at? (line no.)

- 5

Destructor

When is a destructor called?

1. When object goes out of scope

```
1.  Void h(...){
2.     ….
3.     {
4.        ……//some code
5.        Rectangle rectangle(width, height);
6.        ….. //some code
7.     }
8.     ….//some other code
9.  }
```

Destructor of rectangle is called at? (line no.)

Destructor

When is a destructor called?

1. When object goes out of scope

```
1.  Void h(...){
2.     ….
3.      {
4.          ……//some code
5.          Rectangle rectangle(width, height);
6.          ….. //some code
7.      }
8.     ….//some other code
9.  }
```

Destructor of rectangle is called at? (line no.)

- 7

Destructor

When is a destructor called?

1. When object goes out of scope

```
1.  Void h(...){
2.      Rectangle rectangles[100];
3.  }
```

Destructor of rectangle is called at? (line no.)

Destructor

When is a destructor called?

1. When object goes out of scope

```
1.  Void h(...){
2.     Rectangle rectangles[100];
3.  }
```

Destructor of rectangle is called at? (line no.)

- 3

How many times the destructor is called?

Destructor

When is a destructor called?

1. When object goes out of scope

```
1.  Void h(...){
2.     Rectangle rectangles[100];
3.  }
```

Destructor of rectangle is called at? (line no.)

- 3

How many times the destructor is called?

- 100 (once for each object)

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle* c = new Circle(cx, cy, r);
3.          delete c;
4.  }
```

Destructor of circle is called at? (line no.)

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle* c = new Circle(cx, cy, r);
3.          delete c;
4.  }
```

Destructor of circle is called at? (line no.)

- 3

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle* circles = new Circle[101];
3.          delete [] circles;
4.  }
```

Destructor of circle is called at?
(line no.)

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle* circles = new Circle[101];
3.          delete [] circles;
4.  }
```

Destructor of circle is called at?
(line no.)

- 3

How many times the destructor is called?

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle* circles = new Circle[101];
3.          delete [] circles;
4.  }
```

Destructor of circle is called at?
(line no.)

- 3

How many times the destructor is called?

- 101 (once for each object)

Note : It is array of objects, not object pointers.

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          /….
3.          Rectangle* rectangle = new Rectangle(w, h);
4.          /…..
5.  }
```

Destructor of rectangle is called at?

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          /….
3.          Rectangle* rectangle = new Rectangle(w, h);
4.          /…..
5.  }
```

Destructor of rectangle is called at?

- It’s not called
- Neither the **object** is out of scope, nor explicit delete is called on the pointer.

Destructor

When is a destructor called?

delete circles;

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle * circles[200];
3.          int m = 19; // valid circles
4.          for(int i=0; i<m ; i++) {
5.                circles[i] = new Circle(i, i+1, i*i);
6.          }
7.
8.         // Code to de-allocate
9.          ……..
10.
11. }
```

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle * circles[200];
3.          int m = 19; // valid circles
4.          for(int i=0; i<m ; i++) {
5.                circles[i] = new Circle(i, i+1, i*i);
6.          }
7.
8.         // Code to de-allocate
9.          ……..
10.
11. }
```

```
delete circles;
// No, compilation error, circles is an array

delete [] circles;
```

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle * circles[200];
3.          int m = 19; // valid circles
4.          for(int i=0; i<m ; i++) {
5.               circles[i] = new Circle(i, i+1, i*i);
6.          }
7.
8.         // Code to de-allocate
9.          ……..
10.
11. }
```

```
delete circles;
// No, compilation error, circles is an array

delete [] circles;
// No, compilation error, not a dynamic allocation

for(int i=0; i<200 ;i++) {
    delete circles[i];
}
```

Destructor

When is a destructor called?

1. When object goes out of scope
2. When “delete” is called on object’s pointer.

```
1.  void h(){
2.          Circle * circles[200];
3.          int m = 19; // valid circles
4.          for(int i=0; i<m ; i++) {
5.               circles[i] = new Circle(i, i+1, i*i);
6.          }
7.
8.         // Code to de-allocate
9.          ……..
10.
11. }
```

```
delete circles;
// No, compilation error, circles is an array

delete [] circles;
// No, compilation error, not a dynamic allocation

for(int i=0; i<200 ;i++) {
    delete circles[i];
}
// No, some pointers are not valid

for(int i=0; i<m ;i++) {
    delete circles[i];
}
// Works.
```

Note : Always iterate until the valid elements. Do not assume anything about rest of elements in the array.

Copy Constructor

Copy Constructor

A special constructor to construct objects exactly same as other objects of same type.

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1); // We plan to create c2 exactly same as c1
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
};
```

Copy Constructor

Syntax : <ClassName>(const <ClassName> & other)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1); // We plan to create c2 exactly same as c1
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle(const Circle& other) {
                ....
        }
};
```

Copy Constructor

Syntax : <ClassName>(const <ClassName> & other)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1); // We plan to create c2 exactly same as c1
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle(const Circle& other) {
                ....
        }
};
```

Why are we using a **const** here?

Copy Constructor

Syntax : <ClassName>(const <ClassName> & other)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1); // We plan to create c2 exactly same as c1
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle(const Circle& other) {
                ....
        }
};
```

Why are we using a **const** here?

- So that it won't modify the provided parameter object
- Most of the time, a copy constructor passes by constant reference

Copy Constructor

Syntax : <ClassName>(**const** <ClassName> & other)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1); // We plan to create c2 exactly same as c1
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r){...}
        Circle(const Circle& other) {
                m_radius = other.m_radius;
                m_center = new Point(
                                other.m_center->m_x,
                                other.m_center->m_y
                                );
        }
};
```

Why are we using a **const** here?

- So that it won't modify the provided parameter object
- Most of the time, a copy constructor passes by constant reference

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1);
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r){...}

};
```

What if we don’t write a copy constructor?

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1);
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r){...}

        // default copy constructor written by compiler
        Circle( const Circle& other){
                m_center = other.m_center;
                m_radius = other.m_radius;
        }

};
```

What if we don't write a copy constructor?

- The compiler writes a default one for us.
- It just copies the data members one by one.

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1);
}
```

What if we don't write a copy constructor?

- The compiler writes a default one for us.
- It just copies the data members one by one.

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r){...}

        // default copy constructor written by compiler
        Circle( const Circle& other){
                m_center = other.m_center;
                m_radius = other.m_radius;
        }

};
```

Will it do the right job?

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(c1);
}
```

What if we don't write a copy constructor?

- The compiler writes a default one for us.
- It just copies the data members one by one.

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r){...}

        // default copy constructor written by compiler
        Circle( const Circle& other){
                m_center = other.m_center;
                m_radius = other.m_radius;
        }

};
```

Will it do the right job?

- No, both pointers in c1, c2 pointing to same **Point** object. If one modifies it's center, the other is modified too.

Copy Constructor

```
int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        double p1_x;
        double p1_y;
        double p2_x;
        double p2_y;
        double p3_x;
        double p3_y;
};
```

Will the default copy constructor do the right job here?

Copy Constructor

```
int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        double p1_x;
        double p1_y;
        double p2_x;
        double p2_y;
        double p3_x;
        double p3_y;
};
```

Will the default copy constructor do the right job here?

- Yes, it copies the members one by one.

Copy Constructor

```
int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        double p_x[3];
        double p_y[3];
};
```

Will the default copy constructor do the right job here?

Copy Constructor

```
int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        double p_x[3];
        double p_y[3];
};
```

Will the default copy constructor do the right job here?

- Yes, it copies the members one by one, even arrays.

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        Point * points[3];
};
```

Will the default copy constructor do the right job here?

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        Point * points[3];
};
```

Will the default copy constructor do the right job here?

- No, pointers are copied but objects are shared !!

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        Point * points[3];
        Triangle( const Triangle & other ){
                … ??
        }
};
```

Will the default copy constructor do the right job here?

- No, pointers are copied but objects are shared !!

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        Point * points[3];
        Triangle( const Triangle & other ){
                for(int i=0; i<3; i++) {
                        points[i] = new Point(
                                        other.points[i]->m_x,
                                        other.points[i]->m_y
                                            )
                }
        }
};
```

Will the default copy constructor do the right job here?

- No, pointers are copied but objects are shared !!

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        Point * points[3];
        Triangle( const Triangle & other ){
                for(int i=0; i<3; i++) {
                        points[i] = new Point(
                                        other.points[i]->m_x,
                                        other.points[i]->m_y
                                            )
                }
        }
};
```

Will the default copy constructor do the right job here?

- No, pointers are copied but objects are shared !!

If we write a copy constructor, does the compiler still write default constructor for us?

Copy Constructor

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Triangle t1;
        Triangle t2(t1);
}
```

```
class Triangle {
        Point * points[3];
        Triangle( const Triangle & other ){
                for(int i=0; i<3; i++) {
                        points[i] = new Point(
                                        other.points[i]->m_x,
                                        other.points[i]->m_y
                                            )
                }
        }
};
```

Will the default copy constructor do the right job here?

- No, pointers are copied but objects are shared !!

If we write a copy constructor, does the compiler still write default constructor for us?

– No

Copy Constructor

When is a copy constructor invoked?

1. `Triangle a(b)` // explicit call

Copy Constructor

When is a copy constructor invoked?

1. `Triangle a(b)` // explicit call
2. `Triangle a = b` // declaration with initialization
3. `void area(Triangle triangle)` // pass by value
4. `return triangle` // return by value

Assignment Operator

Assignment Operator

A special function to handle assignments of class/struct objects.

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
};
```

Assignment Operator

Syntax : <ClassName>**& operator=**(const <ClassName> & rhs)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle & operator=(const Circle & rhs) {
                …. ??
        }
};
```

Assignment Operator

Syntax : <ClassName>**& operator=**(const <ClassName> & rhs)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle & operator=(const Circle & rhs) {
                m_radius = rhs.m_radius;
                delete m_center;
                m_center = new Point(
                                rhs.m_center->m_x,
                                rhs.m_center->m_y
                                    );
                 return *this;
        }
};
```

Assignment Operator

Syntax : <ClassName>**& operator=**(const <ClassName> & rhs)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle & operator=(const Circle & rhs) {
                m_radius = rhs.m_radius;
                delete m_center;
                m_center = new Point(
                                rhs.m_center->m_x,
                                rhs.m_center->m_y
                                    );
                return *this;
        }
};
```

Can something go wrong with our code?

Assignment Operator

Syntax : <ClassName>**& operator=**(const <ClassName> & rhs)

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle & operator=(const Circle & rhs) {
                m_radius = rhs.m_radius;
                delete m_center;
                m_center = new Point(
                                rhs.m_center->m_x,
                                rhs.m_center->m_y
                                    );
                return *this;
        }
};
```

Can something go wrong with our code?

- **Aliasing**: two different variables have the same reference (e.g. u = u).
- **Always be cautious of aliasing!**

Assignment Operator

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle & operator=(const Circle & rhs) {
            if( this != &rhs) {
                m_radius = rhs.m_radius;
                delete m_center;
                m_center = new Point(
                                rhs.m_center->m_x,
                                rhs.m_center->m_y
                                    );
            }
            return *this;
        }
};
```

Assignment Operator

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle & operator=(const Circle & rhs) {
            if( this != &rhs) {
                m_radius = rhs.m_radius;
                delete m_center;
                m_center = new Point(
                                rhs.m_center->m_x,
                                rhs.m_center->m_y
                                    );
            }
            return *this;
        }
};
```

This is the traditional way, not widely used. Why?

Assignment Operator

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        Circle & operator=(const Circle & rhs) {
            if( this != &rhs) {
                m_radius = rhs.m_radius;
                delete m_center;
                m_center = new Point(
                                rhs.m_center->m_x,
                                rhs.m_center->m_y
                                    );
            }
            return *this;
        }
};
```

This is the traditional way, not widely used. Why?

- What if allocation with ‘new’ fails?, you already deleted m_center :(

Assignment Operator

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

The modern way :)

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}
        void swap(Circle & other) {
                std::swap(m_radius, other.m_radius);
                std::swap(m_center, other.m_center);
        }

        Circle & operator=(const Circle & rhs) {
             if( this != &rhs) {
                Circle temp(rhs); // copy
                swap(temp);
             } // temp's destructor will be called here.
        }
};
```

Assignment Operator

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}

};
```

What if you don’t define one?

Assignment Operator

```
struct Point {
        double m_x;
        double m_y;
        Point(int x, int y) {...}
};

int main() {
        Circle c1(10, 11, 100);
        Circle c2(5, 6, 50);
        ….
        c1 = c2 // We plan to make c1, similar to c2.
}
```

```
class Circle {
        Point* m_center;
        double m_radius;
        Circle(int c_x, int c_y, int r) {...}

        // default assignment operator
        Circle & operator=(const Circle & rhs) {
            if( this != &rhs) {
                m_center = rhs.m_center;
                m_radius = rhs.m_radius;
            }
            return *this;
        }


};
```

What if you don’t define one?

- Default assignment operator will be written for you
- It just assigns data members one by one.

Summary

- Destructors
  - Order of destruction
  - Invoked when object is out of scope or delete is called on object pointers
  - Destructors with array of objects, array of object pointers, ..
- Copy constructor
  - Definition, default copy constructor
  - How to properly implement and When is it invoked
- Assignment operator
  - Definition, default assignment operator
  - Traditional way, Modern way and When is it invoked

Summary

- Destructors
  - Order of destruction
  - Invoked when object is out of scope or delete is called on object pointers
  - Destructors with array of objects, array of object pointers, ..
- Copy constructor
  - Definition, default copy constructor
  - How to properly implement and When is it invoked
- Assignment operator
  - Definition, default assignment operator
  - Traditional way, Modern way and When is it invoked

Questions??