C++

Destructors, order of construction / destruction

Destructors

- A **destructor** is a member function of a class which is called when the object is destroyed
- For a class X, the destructor is called ~X()
  - Example: X::~X()
- A destructor takes no argument
- A destructor has no return value
- There is only one destructor per class

Default destructor

- If the user does not provide a destructor, the compiler will automatically generate one: the default destructor
- The default destructor does not delete dynamically allocated variable
  - If your object uses dynamically allocated variables, you need to implement a destructor that will release such memory

Example

```
class A{
  private:
    int* array;
    int size;
  public:
    A() : size(10) {
      array = new int[size];
    }

   ~A() {
      delete[] array;
  }
};
```

Destructor call

- The destructor for an object is always automatically called
- An object can be destroyed in two ways:
  - It was allocated on the stack and goes out of scope
  - It was allocated dynamically (on the heap) and it is deleted by using the keyword **delete** (or delete[] if it is an array)

Example

```
void f()
{
 A a(10); // local var. alloc. on the stack

 // …..

} // a destroyed, ~A() called
```

```
void f()
{
 A* a = new A(10); // alloc. on heap

 // …..

 delete a; // a destroyed, ~A() called
}
```

Order of construction and destruction

- In the case where an object contains other objects (“Has - a” relationship), we want to know in which order the component objects are constructed / destructed
- In the case a function declares objects, we want to know the order in which they are destroyed

Order of construction and destruction: example

```
class A {
 public:
   A() { cout << “A()” << endl; }
   ~A() { cout << “~A()” << endl; }
};

class B {
 private:
   A a;
 public:
   B() { cout << “B()” << endl; }
   ~B() { cout << “~B()” << endl; }
};

int main() {
  cout << “beginning” << endl;
  B b;
  cout << “end” << endl;
}
```

Result:

```
Beginning
A()
B()
End
~B()
~A()
```

Order of construction / destruction

- In B(), you do not need to call the constructor for A, it is called automatically
  - B::B() {cout << “B()” << endl;}
    is equivalent to:
    B::B() : a() {cout << “B()” << endl;}
- Of course, you can always decide to control construction by calling the appropriate constructor using the initialization list syntax
- When an object is constructed, all its data members are constructed first
- In ~B() you do not need to call the destructor for A, it is called automatically
- Destruction appears in the opposite order to construction

Example with pointers

```
class A {
 public:
   A() { cout << “A()” << endl; }
   ~A() { cout << “~A()” << endl; }
};

class B {
 private:
   A* a;
 public:
   B() { a = new A(); cout << “B()” << endl; }
   ~B() { cout << “~B()” << endl; delete a; }
};

int main() {
  cout << “beginning” << endl;
  B b;
  cout << “end” << endl;
}
```

Result:

Beginning
A()
B()
End
~B()
~A()

Do not forget to **delete a**

Order of const. / dest. with multiple members

- Class's destructor will automatically invoke the destructors for member objects.
- They are destroyed in the reverse order they appear within the declaration for the class

```
class A {
 public:
  ~A();
 // …
};

class B {
 private:
   A a1;
   A a2;
   A a3;

 public:
   ~B();
 // …
};
```

```
B::~B()
{
  // compiler invoke:
  // a3.~A()
  // a2.~A()
  // a1.~A()
}
```

Order of destruction of local object

- For local objects, order of destruction is opposite to the order of construction
- For an array of objects, order of destruction is opposite to the order of construction

```
void f()
{
  A a1;
  A a2;
  // …

} // a2 destroyed
  // a1 destroyed
```

```
void f()
{
  A a[10];
  // …

} // a[9] destroyed
  // a[8] destroyed …
  // a[0] destroyed
```