The Destructor and the Assignment Operator

Lecture 8
Sections 7.7, 11.6

Robb T. Koether

Hampden-Sydney College

Fri, Feb 3, 2017

1. The Destructor
   - The Automatic Destructor
   - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
   - The Automatic Assignment Operator
4. Assignment

Outline

1. The Destructor
   - The Automatic Destructor
   - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
   - The Automatic Assignment Operator
4. Assignment

The Destructor

The Destructor

```
    Type::~Type();    // Prototype;
```

- The destructor destroys an object, i.e., it deallocates the memory used by the object.

The Destructor

The Destructor

```
    Type::~Type();    // Prototype;
```

- The destructor destroys an object, i.e., it deallocates the memory used by the object.
- The destructor should never be invoked explicitly.

Purpose of the Destructor

- The destructor is used to destroy an object when it passes out of scope.
  - A global variable passes out of scope when the program terminates.
  - A variable that is local to a function passes out of scope when execution returns from the function.
  - A variable that is local to a block `{ }` passes out of scope when execution leaves that block.
  - A volatile object passes out of scope when the evaluation of the expression in which it occurs is completed.
- In general, the scope of an object is determined by where the object is created. When execution leaves that environment, the object is destroyed.

Vectr Destructor

Example (Vectr Destructor)

```
˜Vectr()
{
    delete [] m_element;
    return;
}
```

Purposes of the Default Constructor

The Destructor

```
int main()
{
    Vectr v(5, 123);
    {
        Vectr u = 5*v;
    }
    return 0;
}
```

- How many vectors are constructed by this program?
- When are they destroyed?

Purposes of the Default Constructor

The Function operator*()

```
Vectr operator*(double s, const Vectr& v)
{
return v.scalarMultiply(s);
}

Vectr Vectr::scalarMultiply(double s) const
{
    Vectr v(m_size);
    for (int i = 0; i < m_size; i++)
        v.m_element[i] = s * m_element[i];
    return v;
}
```

- How many vectors are constructed and destroyed in this example?

Outline

1. The Destructor
   - The Automatic Destructor
   - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
   - The Automatic Assignment Operator
4. Assignment

The Automatic Destructor

- The automatic destructor
  - Invokes each data member's destructor.
  - Deallocates the memory used by the data members.
- The automatic destructor does not deallocate memory that the data members point to.
- The destructor for a pointer deallocates only the pointer itself.
- In other words, if a data member is a pointer, then the automatic destructor will probably create a memory leak.

Outline

1. The Destructor
   - The Automatic Destructor
   - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
   - The Automatic Assignment Operator
4. Assignment

The makeEmpty() Function

The makeEmpty() Function

```
void makeEmpty()
{
//  Deallocate all memory allocated to the object
//  Return the object to the "empty" state or
//  the default state
}
```

- Just as we write a `makeCopy()` function to facilitate the copy constructor, we may write a `makeEmpty()` function to facilitate the destructor.

The Destructor

The Destructor

```
Type::~Type()
{
    makeEmpty();
}
```

makeEmpty()

Example (makeEmpty())

```
void makeEmpty()
{
    m_size = 0;
    delete [] m_element;
    m_element = NULL;
    return;
}

~Vectr()
{
    makeEmpty();
    return;
}
```

Outline

1. The Destructor
   - The Automatic Destructor
   - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
   - The Automatic Assignment Operator
4. Assignment

The **this** Pointer

- Every (non-static) member function has a hidden parameter named **this**.
- **this** is always the first parameter in such a function.
- **this** is a constant pointer to the object that invoked the member function.

  Type∗ **const this**

- **this** provides us with a name for the invoking object, i.e., ***this**.

The **this** Pointer

- When we write the prototype of a member function as

Apparent Prototype

```
Type::func(params);
```

the actual prototype is

Actual Prototype

```
Type::func(Type* const this, params);
```

The **this** Pointer

- Furthermore, when we create a *constant* member function

Apparent Prototype

```
Type::func(params) const;
```

the actual prototype is

Actual Prototype

```
Type::func(Type const* const this, params);
```

- In this case, **this** is a constant pointer to a constant object.

Usage of the **`this`** Pointer

- Inside a member function, we refer to a data member by its name, e.g. `m_size`.
- It is interpreted as **`this`**`->m_size`.

Usage of the **`this`** Pointer

- Inside a member function, we invoke another member function of the same class by the function's name, e.g., `scalarMultiply(5)`.
- It is interpreted as **`this`**`->scalarMultiply(5)`.

Outline

1. The Destructor
   - The Automatic Destructor
   - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
   - The Automatic Assignment Operator
4. Assignment

The Assignment Operator

The Assignment Operator Prototype

```
Type& Type::operator=(const Type&);
```

The Assignment Operator Usage

```
ObjectA = ObjectB;
```

- The assignment operator assigns to an existing object the value of another existing object of the same type.
- The assignment operator must be a member function.
- It can be invoked only through the operator =.

Form of the Function **operator=()**

The Assignment Operator

```
Type& Type::operator=(const Type& value)
{
    if (this != &value)
    {
        //  Clear out the old value
        //  Assign the new value
    }
    return *this;
}
```

Form of the Function `operator=()`

The `makeEmpty()` and `makeCopy()` Functions

- `void makeEmpty();`
- `void makeCopy(const Type& value);`

- `makeEmpty()` clears out the old value of the object.
- `makeCopy()` assigns the new value to the object.
- It is convenient write these two member functions and then use them in the copy constructor, the destructor, and the assignment operator (and the `input()` function).

The Assignment Operator

The Assignment Operator

```
Type& Type::operator=(const Type& value)
{
    if (this != &value)
    {
        makeEmpty();
        makeCopy(value);
    }
    return *this;
}
```

makeEmpty()

Example (makeEmpty())

```
Vectr& operator=(const Vectr& v)
{
    if (this != &v)
    {
        makeEmpty();
        makeCopy(v);
    }
    return *this;
}
```

The input() Function

The input() Function

```
void Type::input(istream& in)
{
    makeEmpty();    // Avoid memory leak
//  Read the object
}
```

Outline

1. The Destructor
   - The Automatic Destructor
   - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
   - The Automatic Assignment Operator
4. Assignment

The Automatic Assignment Operator

- The automatic assignment operator uses each data member's assignment operator to assign values to them from the other object.

Multiple Assignments

- The assignment operator is *right-associative*.
- The statement

  ```
  a = b = c = d;
  ```

  is equivalent to

  ```
  a = (b = (c = d));
  ```

Multiple Assignments

- What about the statements

  ```
  ((a = b) = c) = d;
  ```

  and

  ```
  (a = b) = (c = d);
  ```

- Are they legal?
- If so, what do they do?

Outline

1. The Destructor
    - The Automatic Destructor
    - The `makeEmpty()` Function
2. The `this` Pointer
3. The Assignment Operator
    - The Automatic Assignment Operator
4. Assignment

Assignment

Homework

- Read Sections 7.7, 11.6, pages 407 - 408, 704 - 710 (8th ed.).