Queens College, CUNY,   Department of Computer Science
Object Oriented Programming in C++
CSCI 211 / 611
Summer 2018
Instructor: Dr. Sateesh Mane

August 7, 2018

Polymorphism: Part II

- In this lecture we continue our study of the concept of **polymorphism**.

- In this lecture we shall learn about the **virtual destructor**.

1 Polymorphism: a comment

If a class has virtual methods, the class destructor should be tagged as virtual.

2   Base class `BA`, derived class `DE`

- For later use in this lecture, we declare a base class `BA` and a derived class `DE` which inherits from `BA`.

- We print a line of output in the constructor and destructor, for educational use below.

- We do not need anything else in this lecture hence we do not write any class methods.

```
class BA {
public:
  BA()  { cout << "BA::constructor" << endl; }
  ~BA() { cout << "BA::destructor" << endl; }
};

class DE : public BA {
public:
  DE()  { cout << "DE::constructor" << endl; }
  ~DE() { cout << "DE::destructor" << endl; }
};
```

3 Dynamic memory allocation with inheritance, part 1

- Recall the order of construction and destruction of an object of a derived class.

- Construction:

    1. The compiler invokes the constructors in sequence.
    2. **The compiler works its way UP the inheritance tree, starting from the base.**
    3. The base class constructor is called first.
    4. The derived class constructor is called next.

- Destruction:

    1. The compiler invokes the destructors in sequence.
    2. **The compiler works its way DOWN the inheritance tree, starting at the top.**
    3. The derived class constructor is called first.
    4. The base class constructor is called next.

- Here is a main program to illustrate.

    1. We dynamically allocate and release an object of the derived class DE.
    2. The pointer is also to the derived class DE.
    3. Everything works correctly and there are no problems.

```
#include <iostream>
using namespace std;

// class declarations

int main()
{
  DE *pDE = new DE;          // allocate memory, invoke constructor
  cout << endl;
  delete pDE;                // release memory, invoke destructor
  return 0;
}
```

- Program output: construction and destruction is printed in the correct sequence.

```
BA::constructor          // base class constructor first
DE::constructor          // derived class constructor second

DE::destructor           // derived class destructor first
BA::destructor           // base class destructor second
```

4

4 Dynamic memory allocation with inheritance, part 2

- We run the same program but using a **pointer to the base class.**

- The program compiles and runs.

- The construction works correctly, **but the destruction is wrong.**

```
#include <iostream>
using namespace std;

// class declarations

int main()
{
  BA *pBA = new DE;                    // pointer to base class
  cout << endl;
  delete pBA;                          // *** something goes wrong ***
  return 0;
}
```

- Program output: construction is correct, **destruction is wrong.**

```
BA::constructor          // base class constructor first
DE::constructor          // derived class constructor second

BA::destructor       // ** ONLY THE BASE CLASS DESTRUCTOR IS INVOKED **
```

- **The destructor of the derived class is not invoked.**

- This causes a memory leak, because not all of the dynamically allocated memory is released.

- The source of the difficulty is the "IS-A" relationship.

  1. The call to operator `delete` invokes a function call to the destructor.
  2. The pointer "IS-A" pointer to the base class (not the derived class).
  3. **Therefore the statement "`delete pBA`" calls the base class destructor ∼BA.**
  4. This is the error.

- **The destructor must be tagged as virtual.**

5   Virtual destructor

- The operator `delete` calls the destructor as a function call.

- For polymorphic memory release to work correctly, **the destructor must be tagged as virtual.**

- As with all other virtual functions, tagging the destructor as virtual in the base class will tag all the destructors in the derived classes as virtual, throughout the inheritance tree.

  1. **We update the base class and tag its destructor as virtual.**
  2. We also edit the output string in the base class destructor for educational use.

```
class BA {
public:
  BA()  { cout << "BA::constructor" << endl; }
  virtual ~BA() { cout << "virtual BA::destructor" << endl; }   // virtual destructor
};
```

- The main program is exactly the same as in Sec. 4.

- With a virtual destructor, polymorphic dynamic memory release works correctly.

```
#include <iostream>
using namespace std;

// updated class declarations with virtual destructor

int main()
{
  BA *pBA = new DE;                   // pointer to base class
  cout << endl;
  delete pBA;                         // works correctly
  return 0;
}
```

- Program output: construction is correct, **destruction is correct.**

```
BA::constructor             // base class constructor first
DE::constructor             // derived class constructor second

DE::destructor              // derived class destructor called correctly
virtual BA::destructor      // ** VIRTUAL DESRUCTOR **
```

6 Polymorphic dynamic allocation of array

- The polymorphic code to dynamically allocate and release an array follows exactly the same paradigm for allocating and releasing arrays of objects.

- To release memory we must call operator `delete []`, with the square brackets.

- Here is the same main program as in Sec. 5, where we allocate and release an array.

- With a virtual destructor, everything works correctly.

```
#include <iostream>
using namespace std;

// updated class declarations with virtual destructor

int main()
{
  int n = 2;
  BA *array_BA = new DE[n];    // dynamically allocate array
  cout << endl;
  delete [] array_BA;          // deallocate array using operator delete []
  return 0;
}
```