Lec_30.md

Lecture 30

Nov. 25/2020

Inheritance

Implications of Derived and Base Objects

A **derived** object is *also* a **base** object

- The **derived** object has all the function/data members (public and private) of **base**
  - Can contract/shrink to form a **base** object

On the other hand, A **base** object cannot grow into a **derived** object

Inheritance Demo Notes

Polymorphism

A property of **object-oriented programming**

- Objects can behave as different objects (based on the **inheritance chain**)

Source Files

1. Constructor prototype in the `.h` file *do not* need initializer lists

- Write initializer list in function implementation in `.cpp` file

2. Source files needed to implement inheritance

- Need to provide `.h` and `.o` files
  - Do *not* need to provide `.cpp`

3. Need to provide *all* `.h` and `.o` files for *all* inherited classes in the **inheritance chain**

- Need to provide `Name.o`, `Name.h`, `Contact.o`, `Contact.h` when implementing `LongContact` which inherits `Contact`
  - And `Contact` inherits `Name`

Object Constructor

1. Constructor prototype in the `.h` file *do not* need initializer lists

- Write initializer list in **function implementation** in `.cpp` file

2. Object creation is bottom up for inherited classes.

- Use initializer lists to call base object constructors

3. Using initializer lists calls constructors on the *same* object

- The base and derived constructors are called on *same* object

    - Same location in memory

4. Forgetting the initializer list in the derived class

- Calls the **default** constructor

Object Destruction

1. Object destruction is top down for inherited classes.

- Do not need to delete base objects in derived object
    - C++ handles this object deletion for you
    - Write destructor *only* for **derived** class

Pointers to Derived and Base Objects

Pointer to objects of type **base** are *also* pointers to objects of types **derived**

```
Name* name_ptr;
Contact* contact_ptr = new Contact();
name_ptr = contact_ptr;
```

Reasoning: The derived object has **everything** the base object has

Pointer to objects of type **derived** ***can not*** point to objects of type **base**

```
Contact* contact_ptr;
Name* name_ptr = new Name();
contact_ptr = name_ptr;
```

Notes:

1. Problem: `contact_ptr` may be used to access **derived** members (`Contact`), which *potentially not exist* in **base** members (`Name`)