Lec_31.md

Lecture 31

Nov. 26/2020

Announcements

Quiz 6 runs on December 2nd

- Make sure to start before 11:30 pm EST (30 minutes before Midnight)
  - Classical quercus quizzes submit *immediately* at 12 Midnight
- Otherwise, same format as other quizzes!

Inheritance

Pointers to Derived and Base Objects

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

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

Problems with Polymorphism

If the data is purely static (e.g. no dynamic allocation), then function calls can be traced easily

```
Contact c;
Name n;

c.print();
n.print();
```

1. `c.print()` calls `Contact::print()`
2. `n.print()` calls `Name::print()`

But because **base** pointers can point to multiple types (derived and base), there arise certain problems.

```
Name* np;
Contact* cp;

cp = new Contact();
cp->setNameAddress(“Tarek”, “123 Main St”);
np = cp;

cp->print(); //1.
np->print(); //2.
```

Notes:

1. What gets printed in `cp->print()` ?

- Does this evoke `Contact::print()` ?

2. What gets printed in `np->print()` ?

- Does this evoke `Name::print()` ?

Turn to understanding **object binding**

Object Binding

Function calls are **binded** to specific functions

Static/Early Binding

Determine the function call (in our example, either `Contact::print()` or `Name::print()`) based on the *type* of the pointer

- This is called **static binding** or **early binding**

Except, it can be *impossible* to figure out the object that a pointer is pointing to at run-time

- Base pointer can point to base, or derived
  - If `np` points to a `Contact` object, calling `np->print()` will call `Name::print()` because of **static binding**
  - We don't want this behaviour, because `np` is a `Contact` object

Dynamic/Late Binding

Determine the function call (in our example, either `Contact::print()` or `Name::print()`) based on the *type of the object* that the pointer points to.

- This is called **dynamic binding** or **late binding**

We want to tell the compiler to generate code that inspects the *type of the objects* that a pointer points to at *run-time*.

- Want to *implement* **dynamic binding**

Virtual Functions

First, define **function signature** to be the declaration of a function and the types of its parameters:

```
void setName(constchar* newName);
```

has the **function signature**

```
void setName(constchar*);
```

basically, the **function signature** is like a *unique ID* for a function

- No two functions can have the same **function signature**

Virtual functions allow us to implement **dynamic binding**

- Declaring a function as **virtual** also makes all of the functions with the same **function signature** *also* **virtual**
  - In other words, functions in derived classes with the same **signature** are *also* **virtual**

```
class Name {
  private:
    char * theName;
```

```
  public:
    Name();
    virtual ~Name();
    ...
    virtual void print();//base class virtual print()
};
class Contact : public Name {
  private:
    char * theAddress;
  public:
    Contact();
    virtual ~Contact();
    ...
    virtual void print();//derived class virtual print()
};
```

1. `virtual ~Name();`

- This is the declaration for a **virtual** function

2. `virtual void print();`

- Notice this is called twice
  - Once in `Name`
  - Once in `Contact`
- Only the **base** class needs to have the `virtual` keyword
  - A `virtual` function definition makes all of the functions with the same **signature** also virtual.

Virtual functions are **dynamically binded**

- A virtual function that is defined multiply in derived classes will be called *based on the type of the object* that the pointer the function is called on points to

```
Name* np;
Contact* cp = new Contact();
np = cp;
np->print();
```

1. `np->print();` prints `Contact::print()`

- If `Name::print()` is defined as `virtual void Name::print`

Virtual Destructors

Destructors are not virtual by default

Why do we want to make destructors **virtual**?

```
Name* np;
Contact* cp = new Contact();
np = cp;

delete np;
```

Notes:

1. What gets deleted when `delete np;` is called?

- If the destructor is not **virtual**, this is actually *undefined behaviour*
- If the destructor is **statically binded**, calling `delete np;` actually calls `Name::~Name();`

    - Potential to leave memory leaks
        - What if there are dynamically allocated data members in `Contact` ?

OOP Language Differences

In Java, *all methods* are defined as **virtual** by default.

- Much slower, since **virtual** functions require checks (of the object type) on **dynamic binding**

In C++, the programmer must *define* functions as **virtual**.

- Different programming philosophy:

> Don't pay for something you don't use

- Must define something as **virtual**
- Pay the price only if needed