Lec_32.md

Lecture 32

Dec. 01/2020

Announcements

Final Exam

- Summative Assessment
- 35% of ECE244 Mark
- December 14, 2020
    - Starting at 9:30am ET
    - 2.5 hour duration
    - Synchronous Exam
        - Everyone takes exam at same time, regardless of time zone

Final Exam Review Session

- December 13, 2020
    - 10am-12pm (ET)
- Come with questions!

Teaching Evaluation

- Please provide feedback!
- Completely anonymous

Inheritance

Pointers to Base/Derived Classes

From previously:

During run-time, a **base** pointer can point to either a **base** object or a **derived** object

Binding

Static Binding:

- **Function bindings** are determined at *compile-time*

Dynamic Binding:

- **Function bindings** are determined at *run-time*

Virtual Functions

Defining a function with the **virtual** keyword allows functions to *bind dynamically*

- 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

virtual functions address the problem of calling the right functions with the same **signature** in **derived/base** classes; however,

- What about functions that are specific to **derived** classes?
    - e.g. `setNameAddress()`
        - Not implemented in **base** `Name`

So even with **virtual** functions, there remains a challenge with **base/derived** pointers:

```
Contact *cp;
cp = new Contact();
cp->setNameAddress("Tarek",  "123 Main St");
np = cp;
np->setNameAddress("Tom", "2 Eva St");
```

Notes:

`np->setNameAddress("Tom","2 Eva St");` is a *compile-time error*

Since the type of `np` is *unknown* at compile-time, and could be **derived** but also could be **base**

- And **base** object (`Name`) does ***not*** contain a function `setNameAddress`
- Error regardless of **static/dynamic** binding

Turn to **Dynamic casting**

Dynamic Casting

We can use `dynamic_cast` to determine the type of the object a **base** pointer is pointing to

- Returns a (cast) pointer to object if `*ptr` is of type `t` , otherwise returns `nullptr`

```
dynamic_cast<t>(ptr)
```

Checking type of object with `dynamic_cast`

```
Name* np;
...
if(Contact* cp = dynamic_cast<Contact*>(np)){
  cout << "np is pointing to Contact object";
}else if(Name* np = dynamic_cast<Name*>(np)){
  cout << "np is pointing to Name object";
}
```

Can use `dynamic_cast` to then correctly call **derived** functions

Type ID

Another way to `dynamic_cast` is to use **Type ID's**

- `typeid` is *compiler specific*
    - Returns a string with the *internal compiler name* for the type of a variable
    - On ECF, these names are of the format `"Xname"` ,
        - where X is the number of characters of the name of the object,
        - followed by the name

        - e.g. `4Name`, `7Contact`, `11LongContact`

```
#include <typeinfo>

typeid(variable).name()
```

```
#include <typeinfo>

Name* np;
...
cout<< "np is pointing to a " << typeid(*np).name()<< " object" << endl;
```

Notes:

1. `typeid(*np).name()`

- Returns a string with the *internal compiler name* for variable type

In general, `dynamic_cast` is better

- Not compiler-specific?

ArrayDB Example

Want to create a database system for Skule

- Student, Staff, Prof records

```
Record*  _arrayDB[ _maxsize];
If ( _arrayDB[i]->getKey() == ....)
....
arrayDB[i]->print();
```

Notes:

1. arrayDB needs to know the *type* of `Record`
2. arrayDB needs a key to sort the `Record`s
3. arrayDB needs a print function that prints the `Record`

Base Record Class

```
class Record {
  private:
    int key;
  public:
    Record();
    virtual ~Record();
    void setKey(int k);
    int getKey();
    virtual void print();
};
```

Staff Record Class

```
class staffRecord: public Record {
  private:
    int performance[12];
    float salary;
  public:
    Record();
    virtual ~staffRecord();
    void setSalary(float k);
    ...
    virtual void print();
};
```

Notes:

1. Additional private data members (performance, salary) defined in `StaffRecord`

- **Inheritance** usages

Professor Record Class

```
class profRecord : public Record {
  private:
    ...
  public:
    profRecord();
    virtual ~profRecord();
    ...
    virtualvoid print();
};
```

Polymorphism

```
Record*  _arrayDB[ _maxsize];
```

The `Record` arrayDB can store 4 types of pointers:

1. `studentRecord*`
2. `staffRecord*`
3. `profRecord*`
4. `Record*`