Lec_33.md

Lecture 33

Dec. 02/2020

Inheritance

ArrayDB - A Polymorphic Array

We made a **polymorphic** array that stores pointers to *different* objects

Recall that the `Record` arrayDB can store 4 types of pointers:

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

But what *is* a `Record*` pointer?

- A `Record*` object is just the base class
  - A `Record*` object does not have any meaning? (is it a student, prof, or staff?)
    - We want to provide the `Record` class to just provide the skeleton/layout for other derived objects
    - Do **not** want users to be able to create `Record*` objects

So we can make `Record` **abstract**

Abstract Classes

We make a class **abstract** by making at least one of it's virtual functions **abstract**

- e.g. An **abstract** class has at least one **pure virtual** function

```
class Record{
private:
  int key;
public:
  Record();
  virtual ~Record();
  void setKey(int k);
  int getKey();
  virtual void print()=0;
};
```

Notes:

1. `virtual void print()=0;`

- This is a **pure virtual/abstract** function
- No implementation for the function
- *Cannot* make objects of `Record` class with this prototype

  - `Record` is now an **abstract** class
  - `new Record();` is a *compile-time* error

Derived classes from **abstract** base classes ***must*** provide an implementation for `print()`

- Otherwise, they become abstract as well
- Basically, you are *forced* to override the **abstract** function

Constructors *cannot* be **abstract**

Destructors can be made **abstract**

- Considered poor practice

Protected Data Members

We used `private` and `public` keywords to set access control restrictions

- External classes can't use `private` data/function members of a given class

derived classes can *see* the private data members of **base** classes, however it **cannot** access them.

- **protected** members are a way to relax this access control restriction

```
class Name {
  private:
    char * theName;
  public:
    ...
    void print();
};

class Contact : public Name{
  private:
    char * theAddress;
  public:
    ...
    void print();
};
```

Now looking in `Contact::print()`

```
void Contact::print() {
  Name::print();
  cout << theAddress << endl;
}
```

Notes:

1. `Name::print();`

- We have to call `Name::print();` since we have no way to access the private data members of `Name`
- We can use `protected` members to relax this access control restriction

```
class Name {
  protected:
    char * theName;
  public:
    ...
    void print();
```

```
};

class Contact : public Name{
  private:
    char * theAddress;
  public:
    ...
    void print();
};
```

Notes:

1. `protected:`

- Define `char* theName` as a `protected` data member instead of `private`
  - We can now use `theName` in **derived** classes

Now looking in `Contact::print()`

```
void Contact::print() {
  cout << theName << endl;
  cout << theAddress << endl;
}
```

Notes:

1. `cout << theName << endl;`

- We can now *directly* access `theName`
  - Even though `theName` is defined in the **base** class

Types of Inheritance

We can use `public`, `private`, and `protected` keywords on the *type* of inheritance as well

- Related to access control

```
class Derived: public Base{ ... };
```

Base	Derived
Public	Public
Protected	Protected
Private	Inherited, but inaccessible

Notes:

- Used in 90% of Inheritance
- Enables inheritance chains
  - Enables classes to inherit `Derived` with same properties that `Derived` inherits from `Base`

```
class Derived: private Base{ ... };
```

Base	Derived

Base	Derived
Public	Private
Protected	Private
Private	Inherited, but inaccessible

Notes:

- `private` inheritance effectively stops the inheritance chain
  - A class that inherits `Derived` will not have access to any data/function members

```
class Derived: protected Base{ ... };
```

Base	Derived
Public	Protected
Protected	Protected
Private	Inherited, but inaccessible

Notes:

- `protected` inheritance is not used often
  - Good to know about though

Complexity Analysis

We want to characterize the execution time of programs

- Want to know how long a program takes to execute
  - Given, for example, number of inputs
- Compare alternatives to solving the same problem

We want to design faster algorithms to solve problems quicker!

Complexity of Algorithms

- What affects execution time
- Big-O notation

For **complexity analysis**, we're most concerned with **execution time**

- As opposed to memory usage, or power, or other concerns

Execution Time

example program:

```
int count = 0;
for(i = 0;i < n;i++){
  if(a[i]<t) count = count+1;
}
cout << count;
```

So what does **execution time** of a program depend on?

1. Size of the input `n`
2. Input itself (data in the array w.r.t t)

- `if(a[i]<t) count = count+1;` only runs depending on values of `a[i]` and `t`

3. Hardware/Compiler optimization
4. Steps run by program (algorithm)

But since runtime is **hardware** dependent, we want a *objective/standard* way to measure runtime

- That way we can *objectively* compare program efficacy

Measuring Execution Time

Define **execution time** in terms of a "step"

Define a Step

A **step** is an instruction/operation that takes a *constant* amount of time

- **step** runtime is independent of the size of the problem (e.g. `n`)

Steps include:

`a=0`

`(c>5)`

`count = count+1`

`a[i] = k`

Not steps include:

`for(i = 0;i < n;i++)`

`if(a[i]<t) count = count+1;`

Time Complexity Scenarios

Using example:

```
int count = 0;
for(i = 0;i < n;i++){
  if(a[i]<t) count = count+1;
}
cout << count;
```

Best Case Scenario

Input data causes the algorithm to execute the least number of steps

Best Case: All elements of `a[]` are larger than `t`

- 1 step

Worst Case Scenario

Input data causes algorithm to execute the largest number of steps

Worst Case: All elements of `a[]` are larger than `t`

- 2 step

Average Case Scenario

Input data causes algorithm to execute an average number of steps

Worst Case: All elements of `a[]` are larger than `t`

- 1.5 step