CS 32: Discussion 1D

TA: Shichang Zhang
LA: Stephanie Doan, Rish Jain

Announcements

- Homework 3 due 11 pm Tuesday (May 4th)

Overview

- Inheritance and polymorphism
- Recursion

Inheritance & Polymorphism

- **Inheritance**
  - Motivation & Definition: Deriving a class from another
  - Reuse, extension, specification (override)
  - Construction & Destruction
  - Override a member function

- **Polymorphism**
  - Virtual functions

Inheritance: motivation

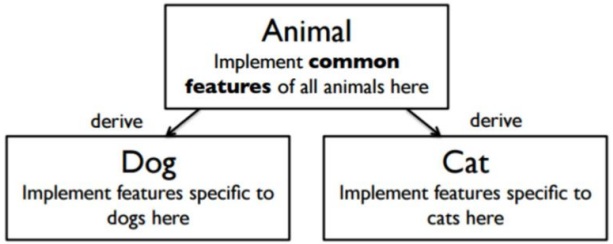
- The process of deriving a new class using another class as base.
- Difference of *"is a"(class hierarchy)* and *"has a"(has member/properties)*

```
class Person {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
private:
  string m_sName;
  int m_nAge;
};
```

```
class Student {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
  int getStudentID();
  void setStudentID();
  float getGPA();
private:
  string m_sName;
  int m_nAge;
  int m_nStudentID;
  float m_GPA;
};
```

```
class Professor {
public:
  string getName(void);
  void setName(string & n);
  int getAge(void);
  void setAge(int age);
  int getProfID();
  void setProfID();
  bool getIsTenured();
private:
  string m_sName;
  int m_nAge;
  int m_nStudentID;
  bool isTenured;
};
```

Inheritance: reuse and extension

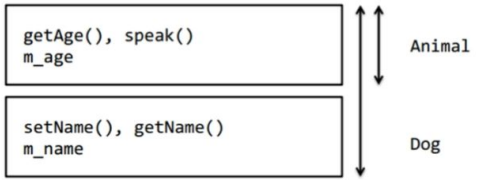



```
class Animal
{
  public:
    Animal();
    ~Animal();
    int getAge() const;
    void speak() const;
  private:
    int m_age;
};
```

base class

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
  private:
    string m_name;
};
```

derived class




```
Dog d1;
d1.setName("puppy");
d1.getAge();
d1.speak();
```

```
Animal a1;
a1.speak();
a1.setName("abc");
```

Inheritance: reuse and extension

- Reuse
  - Every public method in the base class is automatically reused/exposed in the derived class (just as if it were defined).
  - Only public members in the base class are exposed/reused in the derived class(es)! Private members in the base class are hidden from the derived class(es)!
- Extension
  - All public extensions may be used normally by the rest of your program.
  - Extended methods or data are unknown to your base class.
- What about overriding a member function from base classes?

Inheritance: specialization/overriding

- Overriding: same function name, return type and parameter list, defined again in derived classes and different from the base class.
- Different from overloading (same function name, different return type and/or different set of arguments)
- You can still call the member function of base classes, but it seems very rare.

```
Dog d1;
d1.Animal::speak();
```

- Consider how to apply `virtual` keyword in overriding member functions

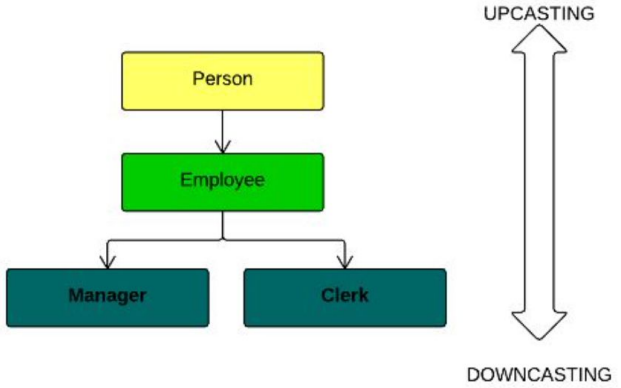
```
void Animal::speak() const
{
  cout << "..." << endl;
}
```

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
    void speak() const;
  private:
    string m_name;
};

void Dog::speak() const
{
  cout << "Woof!" << endl;
}
```

Inheritance: automatic conversion

- Upcasting: A derived class pointer (or reference) to be treated as base class pointer
- Downcasting: Converting base class pointer (or reference) to derived class pointer.



Inheritance: construcion

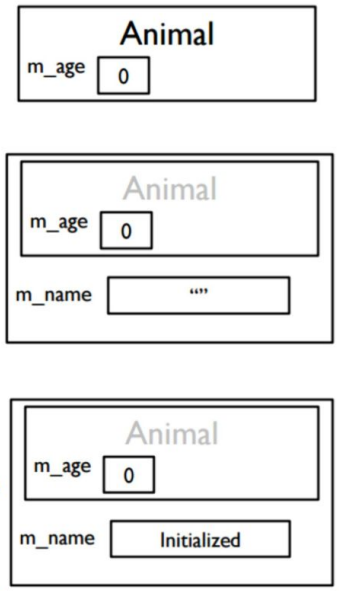
- How to construct a Dog, which is a derived class from Animal?
- Steps:
  - The base part of the class (Animal) is constructed.
  - The member variables of Dog are constructed.
  - The body of constructor (Dog) is executed.

```
class Animal
{
  public:
    Animal();
    ~Animal();
    int getAge() const;
    void speak() const;
  private:
    int m_age;
};
```

base class

```
class Dog : public Animal
{
  public:
    Dog();
    ~Dog();
    string getName() const;
    void setName(string name);
  private:
    string m_name;
};
```

derived class



Inheritance: order of construction and destruction

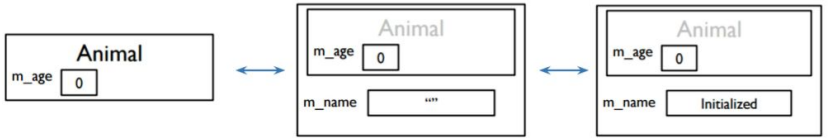
The order of destruction of a derived class: **Just reverse the order of construction.**

Order of construction:

1. Construct the base part, consulting the member initialization list (If not mentioned there, use base class's default constructor)
2. Construct the data members, consulting the member initialization list.(If not mentioned there, use member's default constructor if it's of a class type, else leave uninitialized.)
3. Execute the body of the constructor.

Order of destruction:

1. Execute the body of the destructor.
2. Destroy the data members (doing nothing for members of builtin types).
3. Destroy the base part.



Polymorphism: motivation and definition

- Polymorphism is how you make Inheritance truly useful.
- Think about example of dogs and animals. Once I define a function that accepts **a** (reference or pointer to **a**) `Animal`, not only can I pass `Animal` variables to that class, But I can also pass any variable that was derived from a `Animal`(such as `Dogs`)!

Polymorphism: virtual functions

```
class Shape {
  public:
    virtual double getArea()
    { return (0); }
    ...
  private:
    ...
};
```

```
class Square: public Shape {
  public:
    Square(int side){ m_side=side; }
    virtual double getArea()
    { return (m_side*m_side); }
    ...
  private:
    int m_side;
};
```

```
class Circle: public Shape {
  public:
    Circle(int rad){ m_rad=rad; }
    virtual double getArea()
    { return (3.14*m_rad*m_rad);}
    ...
  private:
    int m_rad;
};
```

```
void PrintPrice(Shape &x)
{
  cout << "Cost is: $";
  cout << x.getArea()*3.25;
}
int main() {
  Square s(5);
  Circle c(10);
  PrintPrice(s);
  PrintPrice(c);
}
```

When you use the `virtual` keyword, C++ figures out what class is being referenced and calls the right function.

Polymorphism works with pointers too.

I will not forget to add `virtual` in front of my destructors when I use inheritance/polymorphism. → *What is the problem if not?*

Polymorphism: pure virtual functions

- Sometimes we have no idea what to implement in base functions. For example, without knowing what the animal is, it is difficult to implement the `speak()` function.
- Solution: Pure virtual functions
- Note:
  - Declare pure virtual functions in the base class. (=0!)
  - Considered as dummy function.
  - The derived class **MUST** implement all the pure virtual functions of its base class.
- If a class has at least one pure virtual function, it is called *abstract base class*.

```
class Shape {
  public:
    virtual double getArea()
    { return (0); }
    ...
  private:
    ...
};
```

Never actually used!

```
class Animal
{
  public:
    Animal();
    virtual ~Animal();
    int getAge() const;
    virtual void speak() const = 0;
  private:
    int m_age;
};
```

Recursion

- Function-writing technique where the functions refers to itself.
- Let's talk about the factorial example again!
  - Similar to mathematical induction → Prove `k=1` is valid and prove `k=n` is valid when `k=n-1` is valid.
  - Base cases are important and need to be carefully considered.

```
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

```
int factorial(int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n - 1);
}
```

Without explicit loops!

Recursion

- Step 1: Find the base case(s).
  - What are the trivial cases? Eg. empty string, empty array, single-item subarray.
  - When should the recursion stop?
- Step 2: Decompose the problem.
  - Take tail recursion as example.

➔ Take the first (or last) of the ***n*** items of information
➔ Make a recursive call to the rest of **(*n*-1)** items. The recursive call will give you the correct results.
➔ Given this result and the information you have on the first (or last item) conclude about current ***n*** items.

- Step 3: Just solve it! *(Well, easier said than done~)*

Break: 5 mins

Worksheet

Codeshare

Room 1

Room 2

Room 3

Room 4

Worksheet Solution