CS32 Discussion
Week 4

Muhao Chen
muhaochen@ucla.edu
http://yellowstone.cs.ucla.edu/~muhao/
Outline

• Stack & Queue
• Inheritance
• Polymorphism
Stack and Queue
class Stack
{
    public:
    bool push(const ItemType& item); // true if successful
    ItemType pop(); // pop
    bool empty() const; // true if empty
    int count() const; // number of items

    private:
    // Some data structure that keeps the items.
};
Applications of Stack

• Stack memory: that’s how function-call works.
• Compiling mathematical expressions: infix expression, matching brackets
• Depth-first-search
Implementation of Stacks

• Container: *linked list, (or dynamic array)*.
• If linked list:
  • Push: Insert node before head.
  • Pop: remove head.
  • Top: read head.
• Count() \ size(): online maintain (with a member variable).
Queue (FIFO)

class Queue
{
    public:
        bool enqueue(const ItemType& item); // push
        ItemType dequeue(); // pop
        bool empty() const; // true if empty
        int count() const; // number of items
    private:
        // some data structure that keeps the items
};

Queue Interface
Applications of Queues

• Windowed data streams.
• Process scheduling (Round Robin)
• Breadth-first-search
Implementation of Queues

• Container: *linked list* with a *tail* pointer.
• Enqueue: Insert node after tail.
• Dequeue: remove head.
• Front(), back(): read head / read tail
• Count() \ size(): online maintain (with a member variable).
class Deque
{
public:
bool push_front(const ItemType& item);
bool push_back(const ItemType& item);
bool pop_front(const ItemType& item);
bool pop_back(const ItemType& item);
bool empty() const; // true if empty
int count() const; // number of items
private:
!! // Some data structure that keeps the items.
};
Implementation

• Standard library uses dynamic array as its container
  • We can also use doubly linked list.
• STL also uses deque as the container of stack and queue class under some configurations.
Question

• How to implement a queue with two stacks?
class Queue<E>
{
    public:
    void enqueue(E item) {
        inbox.push(item);
    }

    E dequeue() {
        if (!outbox.isEmpty()) {
            while (!inbox.isEmpty()) {
                outbox.push(inbox.top());
                inbox.pop();
            }
        }
        E rst = outbox.top()
        outbox.pop();
        return rst;
    }

    private:
    Stack<E> inbox, outbox;
};
Stack, Queue, and Deque in C++ STL

• http://www.cplusplus.com/reference/stack/stack/
• http://www.cplusplus.com/reference/queue/queue/
• http://www.cplusplus.com/reference/deque/deque/
Inheritance
Inheritance

• The process of deriving a new class using another class as a base.

• Our example:

  Dog
  Features of a dog

  Cat
  Features of a cat

• But there might be some common features in the two…
Inheritance

- The process of deriving a new class using another class as a base.
- Our example:
Deriving a class from another

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

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

• Dog inherits Animal.
Deriving a class from another

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

```java
Animal a1;
a1.speak();
a1.setName("abc");
```
Deriving a class from another

• What’s inherited:
  – all member functions except the overloaded assignment operator (operator=), constructors, and the destructor
  – all member variables

• However, the derived class cannot access the private members of the base class directly (e.g. Dog cannot access m_age).

• class D : public B
  – a D object is a kind of B
  – a D is a B (a Dog is an Animal)
Construction

- So, a Dog is an Animal. What happens when we construct a Dog?
- 1. The base part of the class (Animal) is constructed.
Construction

• So, a Dog is an Animal. What happens when we construct a Dog?
• 2. The member variables of Dog are constructed.
Construction

• So, a Dog is an Animal. What happens when we construct a Dog?
• 3. The body of Dog’s constructor is executed.
Construction

• Suppose I want to overload Dog’s constructor to create:

   Dog(string initName, int initAge);

• How would I go about implementing it?
Construction

• Suppose I want to overload Dog’s constructor to create:

```cpp
Dog::Dog(string initName, int initAge)
: m_age(initAge), m_name(initName)
{}  
```

Incorrect
Construction

- Suppose I want to overload Dog's constructor to create:

```cpp
Dog::Dog(string initName, int initAge)
: Animal(initAge), m_name(initName)
{}
```

```cpp
class Animal
{
    public:
        Animal(int initAge);
    ...
};
```
Destruction

- Just reverse the order of construction.
- 1. The body of destructor is executed.
- 2. The member variables are removed.
- 3. The base part of the class is destructed.
Overriding member functions

- Assume `speak()` is implemented as follows.

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

- Dog inherits this function.
- But we want our Dog to really say something when ordered to speak!
Overriding member functions

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;
}

Animal a1;
a1.speak();

• Output
...

Dog d1;
d1.speak();

• Output
Woof!
Overriding member functions

```cpp
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;
}

• Why do we call this **overriding**, not **overloading**?
Overriding member functions

```cpp
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;
}
```

• Why do we call this **overriding**, not **overloading**?

• Overload – same function name, but different return type and/or different set of arguments

• Override – same function name, same return type, same everything, except defined “again” in the derived class.
Overriding member functions

• Can I still call the base class’s `speak()` on a Dog object?
• Yes, just do:

```cpp
Dog d1;
d1.Animal::speak();
```
Polymorphism
Virtual functions: Motivation

• Back to this diagram:

  ![Diagram showing Animal, Dog, and Cat with relationships](image)

  - Animal
    - Implement common features of all animals here
  - Dog
    - Implement features specific to dogs here
  - Cat
    - Implement features specific to cats here

• Suppose we have `speak()` overridden in Cat, where it goes “Meow!”;
Virtual functions: Motivation

• C++ allows a pointer to the base class to point to a derived class.

• What do you think pAni->speak(); will do? should do?

```cpp
Animal *pAni;
int x;
cin >> x;

switch (x)
{
    case 1:
        pAni = new Dog;
        break;
    case 2:
        pAni = new Cat;
        break;
    default:
        pAni = new Animal;
        break;
}
```
Virtual functions: Motivation

• What it will do:
  “…” no matter what x is.
• What it should do:
  “Woof!” if x == 1,
  “Meow!” if x == 2,
  “…” otherwise
• We want the overridden function to be called!

```cpp
Animal *pAni;
int x;
cin >> x;

switch (x)
{
    case 1:
        pAni = new Dog;
        break;
    case 2:
        pAni = new Cat;
        break;
    default:
        pAni = new Animal;
        break;
}
```
Virtual functions

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

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

- `pAni->speak();`

Will use the appropriate version of `speak()` according to the class of the pointee.
Polymorphism

• Late binding / dynamic binding
  – The appropriate version is selected during runtime!

• Polymorphism
  – pAni can take multiple forms.

```cpp
Animal *pAni;
int x;
cin >> x;

switch (x)
{
    case 1:
        pAni = new Dog;
        break;
    case 2:
        pAni = new Cat;
        break;
    default:
        pAni = new Animal;
        break;
}
```
Polymorphism: a realistic example

- **Entity**
  - Functions: damage(), heal(), move()
  - Variables: energy, position

- **NonEnemyNPC**
  - Functions: speak()
  - Variables: stuffToSay

- **Player**
  - Functions: shoot()
  - Variables: equipments, items, exp

- **Enemy**
  - Functions: action()
  - Variables: itemsToDrop

- **Zombie**
  - Functions: action()

- **Robot**
  - Functions: action()

Diagram:

- Player
- Zombie
- Robot

List of Enemy Pointers:

- Player
- Zombie
- Robot

List of Entity Pointers:

- Player
- Zombie
- Robot
Virtual functions

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

base class
```

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

derived class
```

- Wait, what's that `virtual` doing before the destructor of Animal?
Animal speaks?

• speak is a common feature among all (or many) animals.

• But it really means something only if we know what this animal is.

• Option 1:
  – Get rid of speak() function in Animal, and implement it in all the derived classes.
    • Then we can’t do pAni->speak()...

• Option 2:
  – Make it a **pure virtual function.**
Pure virtual functions

- You declare it in the base class, but don’t define it, and add “= 0” in the declaration.
- It is a *dummy* function.
- The derived class **must** implement all the pure virtual functions of its base class.

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

- If a class has at least one pure virtual function, it is called an \textit{abstract base class}.

\begin{verbatim}
Animal a1;           // won't compile
Animal *pAni = new Animal;  // won't compile
Animal *pAni = new Dog;    // still works
\end{verbatim}

- Animal is like a “common” interface without complete implementation. Or, one can think of it as a “framework.”
Bugs in your software are actually special features :)

```plaintext
if ($thirsty==TRUE)
{
}
else
{
}
```