COEN 79
OBJECT-ORIENTED PROGRAMMING AND ADVANCED DATA STRUCTURES
DEPARTMENT OF COMPUTER ENGINEERING, SANTA CLARA UNIVERSITY
BEHNAM DEZFOULI, PHD

Derived Classes and Inheritance

Slide Set: 11

Learning Objectives

- Recognize situations in which inheritance can simplify the implementation of a group of related classes
- Implement derived classes
- Recognize situations in which creating an abstract base class will allow the later implementation of many derived classes that share underlying functions

Derived Classes

Derived Classes

- Object-oriented languages provide support that allows programmers to easily create new classes that acquire some or many of their properties from an existing class
- The original class is called the **base class** and the new, slightly different class is the **derived class**

Derived Classes

- Assume you write a clock class to keep track of a time value such as 9:48 P.M
- Now suppose you're writing a program with various kinds of clocks: 12-hour clocks, 24-hour clocks, alarm clocks, cuckoo clocks, and so on
  - For example, a `cuckoo_clock` might have an extra function, `is_cuckooing`, that returns true if its cuckoo bird is currently making noise
- One possible solution uses no new ideas: Modify the original clock definition by adding an extra member function
- **Even though all of these have similar or identical constructors and member functions, we'll still end up repeating the member function implementations for each different kind of clock**

Derived Classes

- Derived classes use a concept called **inheritance**
- Once we have a class, we can then declare new classes that **contain all of the members of the original class— plus any extras that you want to throw in**
- This new class is called a **derived class** of the original class, and The original class is called the **base class**
- The members that the derived class receives from its base class are called **inherited members**

How to Declare a Derived Class

- In the definition of the derived class, the name of the derived class is followed by a **single colon**, the keyword ***public***, and then the **name of the base class**

```
class cuckoo_clock : public clock
{ ...
```

- **All of the public members of an ordinary clock are immediately available as public members of a `cuckoo_clock`**

How to Declare a Derived Class

A Derived Class Definition

```
class cuckoo_clock : public clock
{
  public:
    bool is_cuckooing( ) const;
};
```

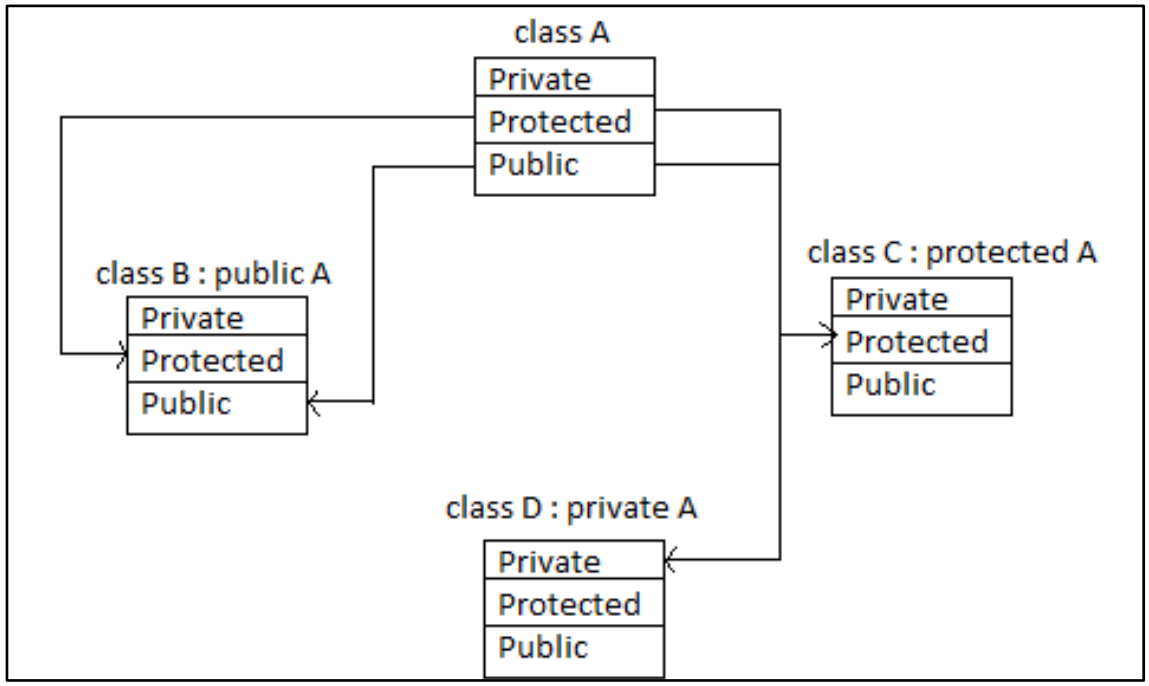
A Member Function Implementation

```
bool cuckoo_clock::is_cuckooing( ) const
{
    return (get_minute( ) == 0);
}
```

- Once the `cuckoo_clock` definition is available, a program may declare `cuckoo_clock` objects using all the public clock member functions and also using the new `is_cuckooing` function

How to Declare a Derived Class

- We can use keyword ***private*** instead of ***public***, resulting in a **private base class**
- With a private base class, all of the public members of an ordinary clock are immediately **available as private members** of a `cuckoo_clock`



Derived Classes

Example

```
class clock {
    public:
        //...
        // CONSTANT FUNCTIONS
        int get_hour( ) const;
        int get_minute( ) const;
        bool is_morning( ) const;

    private:
        int my_hour;
        int my_minute;
        int my_morning;
    };
```

```
class clock24 : private clock
{
    public:
        int get_hour( ) const;
};
```

```
int main()
{
    clock24 myClock1;
    cout<<myClock1.get_hour()<<endl;
    cout<<myClock1.get_minute()<<endl;
}
```

- Works fine!

Does not work!
`get_minute` is a private member of `clock24`

Derived Classes

Example

```
class clock {
    public:
        //...
        // CONSTANT FUNCTIONS
        int get_hour( ) const;
        int get_minute( ) const;
        bool is_morning( ) const;

    private:
        int my_hour;
        int my_minute;
        int my_morning;
    };
```

```
class clock24 : protected clock
{
    public:
        int get_hour( ) const;
};
```

```
int main()
{
    clock24 myClock1;
    cout<<myClock1.get_hour()<<endl;
    cout<<myClock1.get_minute()<<endl;
}
```

- Works fine!

Does not work!
`get_minute` is a protected member of `clock24`

The Automatic Constructor of a Derived Class

- A derived class may declare its own constructors, or it may use the automatic default constructor and the automatic copy constructor that are provided for every C++ class

The Automatic Constructor of a Derived Class

The automatic default constructor

- If we don't declare any constructors for a derived class, then C++ will automatically provide a default constructor
- This default constructor will carry out two steps:
  - **Activate the default constructor for the base class** (to initialize any member variables that the base class uses)
  - **Activate default constructors for any new member variables that the derived class has**, but the base class does not have

The Automatic Constructor of a Derived Class

❑Example

```
cuckoo_clock noisy;
```

- The `cuckoo_clock`'s automatic default constructor will activate the ordinary `clock` default constructor for `noisy`, initializing any private member variables of the ordinary `clock`
- Remember, these private member variables are present in any cuckoo clock, and therefore they must be initialized even though there is no way to directly access these member variables

Derived Classes and Inheritance

The Automatic Constructor of a Derived Class

```
class clock {
public:
  // CONSTRUCTOR
  clock( ) {
  my_hour=12;
  my_minute=0;
  my_morning=true;
  std::cout<<"clock
     constructor!"<<std::endl; };

    // ...

   private:
        intmy_hour;
        intmy_minute;
        intmy_morning;
};
```

```
class clock_design {
public:
    clock_design(){
    color="brown";
    weight=12;
    std::cout<<"clock_design
        constructor!"<<std::endl; }
    private:
        std::string color;
        int weight;
};
```

```
class clock24 : public clock
{
public:
    clock24() {
    std::cout<<"clock24
     constructor!"<< std::endl; };
     // ...
    private:
        clock_design design_spec;
    };
```

Output when we create a clock24 object:

```
clock constructor!
clock_design constructor!
clock24 constructor!
```

The Automatic Constructor of a Derived Class

The automatic copy constructor

- If a derived class does not define a copy constructor of its own, then C++ will automatically provide a copy constructor
- This copy constructor is similar to the automatic default constructor in that it carries out two steps:
  - **Activate the copy constructor for the base class (to copy any member variables that the base class uses)**
  - **Activate copy constructors for any new member variables that the derived class has** but the base class does not have
- The copy constructors that are activated in Steps 1 and 2 may themselves be automatic copy constructors, or they may be specially written to accomplish correct copying of dynamic data structures

The Automatic Assignment Operator of a Derived Class

The automatic assignment operator

- If a derived class does not define its own assignment operator, then C++ will automatically provide an assignment operator
- This automatic assignment operator is similar to the automatic constructors, carrying out two steps:
  - Activate the assignment operator for the base class (to copy any member variables that the base class uses)
  - Activate the assignment operator for any new member variables that the derived class has but the base class does not have
- The assignment operators that are activated in Steps 1 and 2 may themselves be automatic assignment operators, or they may be specially written to accomplish correct copying of dynamic data structures

The Automatic Assignment Operator of a Derived Class

- Example

```
clock ordinary;
cuckoo_clock fancy;
fancy.advance(60);
ordinary = fancy;
```

- The assignment `ordinary = fancy` is **permitted** because a cuckoo clock (such as `fancy`) can be used at any point where an ordinary `clock` is expected
- An assignment in the other direction, `fancy = ordinary`, is **forbidden** because **an object of the base class (the ordinary `clock`) cannot be used as if it were an object of the derived class** (the cuckoo clock)

The Automatic Assignment Operator of a Derived Class

Allowed:

- When a base class is public, an object of a derived class may be used as if it were an object of the base class.

Forbidden:

- But an object of the base class cannot usually be used as if it were an object of the derived class

The Automatic Destructor of a Derived Class

- If a class does not have a declared destructor, then C++ provides an automatic destructor that carries out two steps:
  - **The destructors are called for any member variables that the derived class has**, but the base class does not have
  - **The destructor is called for the base class**

- Notice that an automatic destructor works differently than an automatic constructor: **An automatic destructor first activates the destructors for member variables and then activates the destructor for the base class**
- But an automatic constructor first activates the constructor for the base class, and then activates the constructors for member variables

Overriding Inherited Member Functions

- A derived class must sometimes perform some actions differently from the way the base class does

❑Example

- The original clock provides the current hour via `get_hour`, using a 12-hour clock
- Suppose we want to implement a derived class that provides its hour on a 24-hour basis, ranging from 0 to 23

- The new clock can be defined as a derived class called `clock24`
- The `clock24` class inherits everything from the ordinary clock, but it provides a new `get_hour` member function
- **This is called overriding an inherited member function**

Derived Classes

Overriding Inherited Member Functions

```
class clock24 : public clock
{
  public:
    int get_hour( ) const; // Overridden from the clock class
};
```

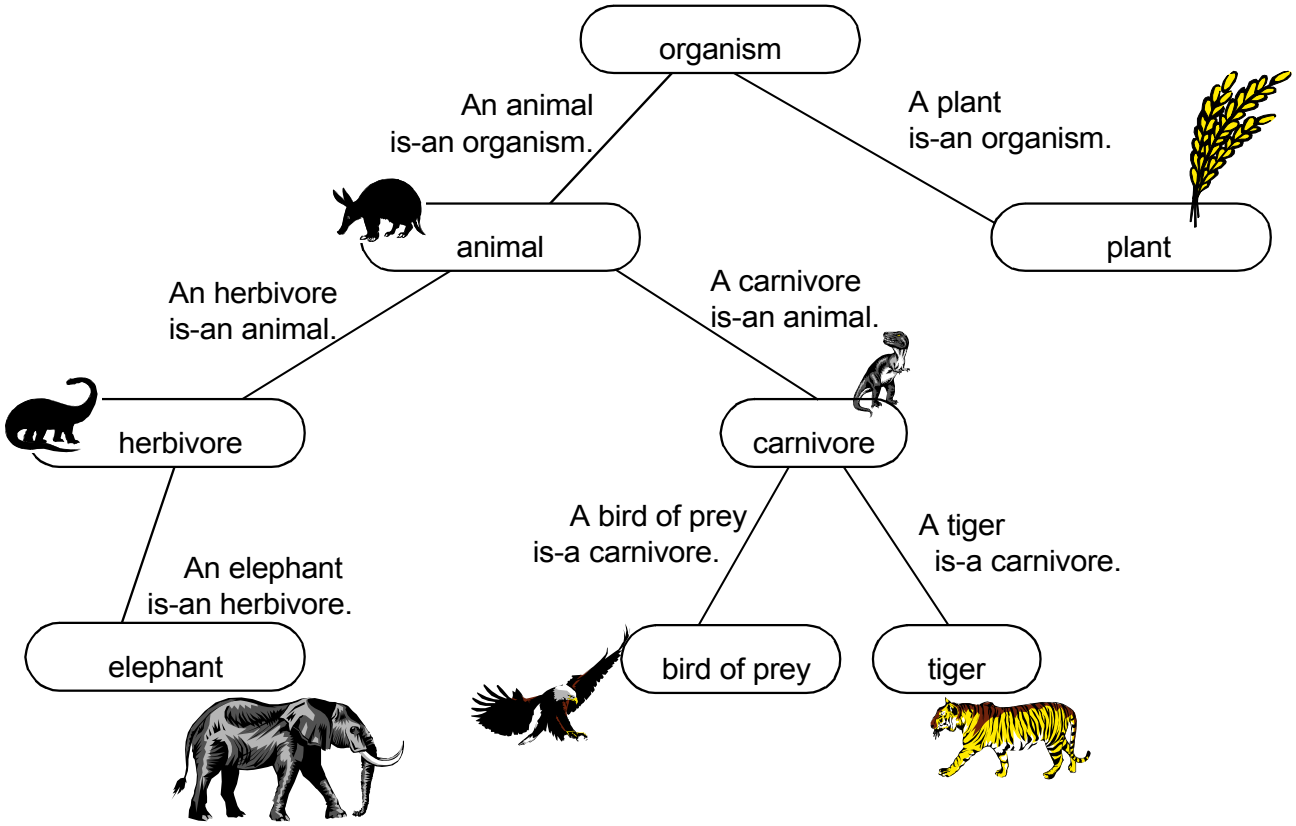
```
int clock24::get_hour( ) const {

    int ordinary_hour;
    ordinary_hour = clock::get_hour( );

    if (is_morning( ))
    {
        if (ordinary_hour == 12) return 0;
    else
        return ordinary_hour;
}
else …
```

When we write `clock::get_hour()`, we are activating the original clock's `get-hour` function

Object Hierarchy Tree

Object Hierarchy Tree

- "**A is-a B**" means that each A object is a particular kind of B object
- The relationships are drawn in a tree called an **object hierarchy tree**
- In this tree, each base class is placed as the parent of its derived classes



Object Hierarchy Tree

The animal Class: A Derived Class with New Private Member Variables

- The animal class will have two new private member variables, which are not part of the organism class, as shown in this partial definition

```
class animal : public organism
{
public:
    || We discuss the animal's public members in a moment.
private:
    double need_each_week;
    double eaten_this_week;
};
```

- When a derived class has some new member variables, **it will usually need a new constructor** to initialize those member variables

Object Hierarchy Tree

How to Provide a New Constructor for a Derived Class

```
class animal : public organism
{
public:
     animal(double init_size = 1, double init_rate = 0, double init_need = 0);
```

- The work of the animal's constructor is easy enough to describe: The first two arguments must somehow initialize the size and growth rate of the `animal`; the last argument initializes `need_each_week`; the value of `eaten_this_week` is initialized to zero

- **But how do we manage to use `init_size` and `init_rate` to initialize the size and growth rate of the animal?**
- Most likely the size and growth rate are stored as private member variables of the `organism` class, but the animal has no direct access to the `organism`'s private member variables

How to Provide a New Constructor for a Derived Class

- Solution: **member initialization list**
- This list is an extra line in the constructor of a derived class
- The purpose is to provide initialization, including a call to the constructor of the base class

```
animal::animal
(double init_size, double init_rate, double init_need)
: organism(init_size, init_rate), // Activate base class constructor
  need_each_week(init_need), // Initial value for need_each_week
  eaten_this_week(0)         // Initial value for eaten_this_week is 0
{
    // Because of the initialization list, this constructor has no work.
}
```

How to Provide a New Constructor for a Derived Class

- The member initialization list appears in the implementation of a constructor, after the closing parenthesis of the parameter list
- The list begins with a colon, followed by a list of items separated by commas
- For a derived class, **the list can contain the name of the base class, followed by an argument list (in parentheses) for the base class constructor**
- The list can also contain **any member variable** followed by its initial value in parentheses
- When the derived class constructor is called, the member variables are initialized with the specified values and the constructor for the base class will be activated before anything else is done
- If a base class constructor is not activated in the member initialization list, then the default constructor for the base class will automatically be activated before any of the rest of the derived class constructor is executed

Object Hierarchy Tree

How to Provide a New Constructor for a Derived Class

- Now assume that we want to implement a class `herbivore`, which includes a function `nibble`

```
class herbivore : public animal
{
public:
    // CONSTRUCTOR
    herbivore(double init_size = 1, double init_rate = 0, double init_need = 0);
    // MODIFICATION MEMBER FUNCTIONS
    void nibble(plant& meal);
};
```

```
herbivore::herbivore(double init_size, double init_rate, double init_need)
: animal(init_size, init_rate, init_need)
{
    // No work is done here, except calling the animal constructor.
}
```

Virtual Member Functions

Virtual Member Functions

- **Virtual member functions** are a new kind of member function that allows certain aspects of activating a function to be delayed until a program is actually running

- To make the explanation of virtual functions concrete, we'll present an example of a class called **game**, which will make it easier for us to write programs that play various two-player games such as chess, checkers, or Othello

Virtual Member Functions

```
class game
{
   public:
   // ENUM TYPE
   enum who { HUMAN, NEUTRAL, COMPUTER }; // Possible game outcomes

   // CONSTRUCTOR and DESTRUCTOR
    game( ) { move_number = 0; }
    virtual ~game( ) { }

    // PUBLIC MEMBER FUNCTIONS
    // The play function should not be overridden. It plays one
    // game, with the human player moving first and the computer
    // second.
    // The computer uses an alpha-beta look ahead algorithm to
    // select its moves. The return value is the winner of the game
    // (or NEUTRAL for a tie).
        who play( );
```

Virtual Member Functions

The programmer of a derived class may override these to obtain different behavior

```
protected:

//*****************************************************************
        // OPTIONAL VIRTUAL FUNCTIONS (overriding these is optional)
//*****************************************************************

    virtual void display_message(const std::string& message) const;

    virtual std::string get_user_move( ) const;

    virtual who last_mover( ) const
            { return (move_number % 2 == 1 ? HUMAN : COMPUTER); }

    virtual int moves_completed( ) const { return move_number; }

    virtual who next_mover( ) const
            { return (move_number % 2 == 0 ? HUMAN : COMPUTER); }

    virtual who opposite(who player) const
            { return (player == HUMAN) ? COMPUTER : HUMAN; }

    virtual who winning( ) const;
```

Virtual Member Functions

- The game class requires that two other virtual functions must be overridden by any derived class
- In each case, **the game class has done a little bit of the work** needed for these functions, **but most of the work must be done by the derived class**
- When these two methods are overridden, the new method in the derived class must activate the original version of the method (otherwise the little bit of work that the game class does will be omitted)
- For these two functions, there is no way that the game class can actually require the derived class to provide the overriding versions, except by making this request in the game class documentation

```
**************************************************************
    // VIRTUAL FUNCTIONS THAT MUST BE OVERRIDDEND:
    // The overriding function should call the original when it
    // finishes.
**************************************************************

        // Have the next player make a specified move:
        virtual void make_move(const std::string& move)
            { ++move_number; }

        // Restart the game from the beginning:
        virtual void restart( ) { move_number = 0; }
```

Virtual Member Functions

```
//*******************************************************************
    // PURE VIRTUAL FUNCTIONS
//*******************************************************************
    // (these must be provided for each derived class)
    // Return a pointer to a copy of myself:
    virtual game* clone( ) const = 0;
    // Compute all the moves that the next player can make:
    virtual void compute_moves(std::queue<std::string>& moves)
        const = 0;

    // ...


    private:
        // MEMBER VARIABLES
        int move_number;   // Number of moves made so far
...
```

- **A pure virtual function is indicated by "= 0" before the end of the semicolon in the prototype**
- The class does not provide any implementation of a pure virtual function

Virtual Member Functions

Protected Members

```
protected:
    // OPTIONAL VIRTUAL FUNCTIONS (overriding these is optional)
    virtual void display_message
        (const std::string& message) const;
```

- The keyword ***protected*** indicates that the items that follow will be a new kind of member, somewhat between *public* and *private*
- **A protected member can be used (and overridden) by a derived class—but apart from within a derived class, any protected member is private; It cannot be used elsewhere**

Virtual Member Functions

- The first protected member function in the game class is `display_message`

```
game::who game::play( )
{
    display_message("Welcome!");
    ...
```

- When we play the game, the welcome message is printed at the start
- But, suppose that we write a game that needs to display messages by some other method
- Our derived class will inherit all the game class members, but it can override any method that it doesn't like

Virtual Member Functions

- Perhaps we want our Connect Four game to print "***" before each displayed message (just to make the messages more exciting)
- We could override the `display_method` function so that `connect4::display_method` first prints "***" and then prints the message

- **With this approach, when the `play` method begins, it will activate `display_message("Welcome!")`, but what will this print?**
  - If it uses `game::display_message`, then the word "Welcome!" appears by itself
  - If it uses `connect4::display_message`, then "***Welcome!"

Virtual Member Functions

When One Member Function Activates Another

- **Normally, when a member function f() activates another member function g(), the version of g() that is actually activated comes from the same class as the function f()**
- This behavior occurs even if the activated function g is later overridden, and f is activated by an object of the derived class

- In this example, this means that `game::play` normally would use `game::display_message`, even if a derived object activates the play method
- Solution: use "**virtual**"

Activating a Virtual Member Function

- Whenever a **virtual** member function is activated, **the data type of the activating object is examined when the program is running**, and the correct version of the function is used

- For a virtual member function, the choice of which version of the function to use is never made until the program is running
- At run time**, the program examines the data type of the object** that activated the method, and thereby uses the correct version of the function

Pure Virtual Functions and Abstract Classes

- A pure virtual function is indicated by “= 0” before the end of the semicolon in the prototype
- The class does not provide any implementation of a pure virtual function
- Because there is no implementation, any class with a pure virtual function is called an **abstract class** and **no instances of an abstract class may appear in a program**
- Abstract classes are used as base classes, and it is up to the derived class to provide an implementation for each pure virtual function

Virtual Member Functions

Example

```
class person
{
public:
    person (std::string name = "empty", int age =0)
    { this->name = name; this->age = age; };

    void print_name() const {
        std::cout << "Name is: "<< get_name() << std::endl; };

protected:
    virtual std::string get_name () const { return name; };

private:
    std::string name;
    int age;
};
```

Virtual Member Functions

Example

```
class professor : public person
{
public:
   professor(std::string name ="empty", int age =0,
     double salary =0, bool nice =true) :
       person(name, age), salary(salary),nice(nice)
   { }

   // Overriding the get_name function
   // Now this is a public function of this class
   std::string get_name () const{
       return(person::get_name() +", he is nice!"); };


private:
   double salary;
   bool nice;
};
```

Virtual Member Functions

Example

```
int main(int argc, const char* argv[]) {

    professor myProf("Ben",36);

    myProf.print_name();

    return0;
}
```

If we use

```
virtual std::string get_name () const { return name; };
```

Then the output is:

Name is: Ben, he is nice!

If we use

```
std::string get_name () const { return name; };
```

Then the output is:

Name is: Ben

Summary

Summary

- Object-oriented programming supports reusable components by permitting new derived classes to be declared, which automatically inherit all members of an existing base class
- All members of a base class are inherited by the derived class, but only the non-private members of the base class can be accessed by the programmer who implements the derived class
- The connection between a derived class and its base class can often be characterized by the “is-a” relationship
- An abstract base class (such as the game class) can provide a common framework that is needed by many derived classes
- An abstract base class has one or more pure virtual functions, which are functions that must be overridden before the class can be used