

Computer Science II — CSci 1200
Lectures 23 and 24
C++ Inheritance and Polymorphism

Review from Lecture 22

- Completed our discussion of binary heaps
 - Push and pop
 - Heap as a vector
 - Make heap
 - Heap sort
- Merging heaps and leftist heaps
- The merge operations and the resulting push and pop operations.

Today's Class

- Inheritance is a relationship among classes.
- Example of inheritance: bank accounts.
- Basic mechanisms of inheritance
- Types of inheritance
- Is-A, Has-A, As-A relationships among classes.
- Example of stack inheriting from list
- Polymorphism
- A polymorphic implementation of a `Plane` class.

Motivating Example 1

- Consider different types of bank accounts
 - Savings accounts
 - Checking accounts
 - Time withdrawal accounts, which are like savings accounts, except that only the interest can be withdrawn.
- If you were designing C++ classes to represent each of these, what member functions might be repeated among the different classes? What member functions would be unique to a given class?
- To avoid repeating common member functions and member variables, we will create a **class hierarchy**, where the common member functions and variables are placed in a **base class** and specialized ones are placed in **derived classes**.

Motivating Example 2

This is based on a homework problem from a previous semester that simulated the assignment of planes to use an airport's runways.

- There are two different plane classes, one for planes that are leaving the airport (`TakeoffPlane`) and one for planes that are arriving at the airport (`LandingPlane`).
- Just as in the previous example, there is repeated functionality that might better be placed in a base class.
- Moreover, the planes are stored in two different maps of pointers, one for `TakeoffPlane` pointers and one for `LandingPlane` pointers.
 - Of course a hash table implementation of a map could be used as well.
- Ideally, instead we would have one map of `Plane` pointers. This is an example of “polymorphism”, where objects of different types are stored in one container through base-class pointers.

Accounts Hierarchy

We will use the accounts class hierarchy as a working example. The code is attached

- **Account** is the *base class* of the hierarchy.
- **SavingsAccount** is a *derived* class from **Account**.
- The member variable **balance** in base class **Account** is **protected**, which means
 - It is NOT publically accessible outside the class, but
 - It is accessible in the derived classes.
- **SavingsAccount** has inherited member functions and ordinarily-defined member functions.
- **SavingsAccount** has inherited member variables and ordinarily-defined member variables.
 - If the base class member variables were declared as private, **SavingsAccount** member functions could not access them.
- When using objects of type **SavingsAccount**, the inherited and derived member functions are treated exactly the same — they are not distinguishable. You can see this for the **SavingsAccount** class by looking at the main program.
 - In particular, note the use of **compound**, which is defined as a **SavingsAccount** member function and **get_balance**, which is defined as a **Account** member function, but used with a **SavingsAccount** object.
- **CheckingAccount** is also a derived class from base class **Account**.
- **TimeAccount** is derived from **SavingsAccount**. **SavingsAccount** is its base class and **Account** is its indirect base class.
 - We will discuss **TimeAccount** more below.

Constructors and Destructors

- Constructors of a derived class call the constructor for the base class immediately, before doing ANYTHING else.
 - This can not be prevented. You can only control the form of the constructor call.
 - See the class constructors in each of the account classes.
- The reverse is true for destructors: derived class constructors do their jobs first and then base class destructors are called, automatically.
 - This is generally only a concern for classes with dynamically allocated objects. The objects must be only deleted once, generally by the location in the class hierarchy that allocated them.

Overriding Member Functions in Derived Classes

- A derived class may redefine member functions in the base class.
- The function prototype must be identical, not even the use of `const` can be different.
 - Otherwise, both functions will be accessible, and major confusion will reign!
- Two examples are in class `TimeAccount`: `compound` and `withdraw`.
 - Notice the syntax (in the member function implementation) showing how `compound` calls the `compound` function for its base class.
- Once a function is redefined it is not possible to call the base class function directly (only indirectly, as in `compound`).

Exercise

Create a class hierarchy of 2D geometric objects, such as squares, rectangles, circles, ellipses, etc. How should this hierarchy be arranged? What member functions should be in each class? Based on your hierarchy, what member variables are available in each class?

Public, Private and Protected Inheritance: What is Accessible Where

- Notice the line

```
class Savings_Account : public Account {
```

- This specifies that the member functions and variables from `Account` do not change their *public*, *protected* or *private* status in `SavingsAccount`.
- This is called *public* inheritance.
- *protected* and *private* inheritance are also possible.
 - With protected inheritance, public members becomes protected; other members are unchanged
 - With private inheritance, all members become private. This is the default.

Stack Inheriting from List

A simple example of using private inheritance is inheriting a stack from a list.

- Here is the full declaration and definition of the stack:

```
template <class T>
class stack : private std::list<T> {
public:
    stack() {}
    stack( stack<T> const& other ) : std::list<T>( other ) {}
    ~stack() {}
    void push( T const& value ) { this->push_back( value ); }
    void pop() { this->pop_back(); }
    T const& top() const { return this->back(); }
    int size() { return std::list<T>::size(); }
    bool empty() { return std::list<T>::empty(); }
};
```

- Private inheritance hides the `std::list<T>` member functions from the outside world.
- These member functions are still available to the member functions of the `stack<T>` class.
- Finally, note that no member variables are defined — the only member variables needed are in the list class.
- When the stack member function uses the same name as the base class (list) member function, the name of the base class followed by `::` must be provided to indicate that the base class member function is to be used.
- The copy constructor just uses the copy constructor of the base class, without any special designation because the stack object is a list object as well.

Is-A, Has-A, As-A Relationships Among Classes

- When trying to determine the relationship between (hypothetical) classes C1 and C2, try to think of a logical relationship between them that can be written:
 - C1 is a C2,
 - C1 has a C2, or
 - C1 is implemented as a C2
- If writing “C1 is-a C2” is best, for example,

a savings account is an account

then C1 should be a derived class (a subclass) of C2.
- If writing “C1 has a C2” is best, for example

a cylinder has a circle as its base

then class C1 should have a member variable of type C2.
- In the case of C1 is implemented as a C2, as in the stack is implemented as a list, then C1 should be derived from C2, but with private inheritance. This is by far the least common case!

Introduction to Polymorphism

- Let us consider a small class hierarchy version of “plane” objects:
 - The base class is a `Plane`
 - The derived classes are `LandingPlane` and `TakeoffPlane`
- Here is `Plane`:

```
class Plane {
public:
    Plane( PlaneId const& id, int scheduled_minute, int num_passenger
    PlaneId const& id() const;
    PlaneStatusType status() const;
    int scheduled_minute() const;
    int runway_number() const;
    int minutes_in_queue() const;
    int num_passengers() const;
    virtual int delay() const = 0; // pure virtual, must be redefine
    virtual void increment_minute( );
    virtual void assign_to_runway( int runway_number );
    virtual TypeOfPlane type() const = 0; // pure virtual, must be r
private:
    PlaneId m_id;
    PlaneStatusType m_status;
    int m_scheduled_minute;
    int m_passengers;
    int m_in_queue_minutes;
    int m_runway;
};
```

- Functions that are common — at least have a common interface — are in `Plane`.
- Some of these functions are marked `virtual`, which means that when they are redefined by a derived class, this new definition will be used, even for pointers to base class objects.
- Some of these virtual functions, those whose declarations are followed by `= 0` are *pure virtual*, which means they must be redefined in a derived class.

- Any class that has pure virtual functions is called “abstract”.
- Objects of abstract types may not be created — only pointers to these objects may be created.
- Functions that are specific to a particular object type are declared in the derived class prototype.
- Here is the `LandingPlane` prototype:

```
class LandingPlane : public Plane {
public:
    LandingPlane( PlaneId const& id, int scheduled_minute,
                  int minutes_until_arrival, int fuel_minutes_remaining,
                  int num_passengers );
    int on_ground_minute() const;
    int minutes_until_arrival() const;
    int fuel_minutes_remaining() const;
    void land_plane( int minute );
    void crash_plane( int minute );
    virtual int delay() const;
    virtual void increment_minute( );
    virtual void assign_to_runway( int runway_number );
    virtual TypeOfPlane type() const { return LANDING_PLANE; }
private:
    int m_landed_or_crashed_minute;
    int m_minutes_until_arrival;
    int m_fuel_minutes_remaining;
};
```

A Polymorphic List of Plane Objects

- Now instead of two separate lists of plane objects, we can create one “polymorphic” list:

```
std::list< Plane* > planes
```

The pointers point to either `LandingPlane` or `TakeoffPlane` objects.

- Objects are constructed using `new`, as in

```
// ... read in information and then
Plane * p_ptr = new LandingPlane( id, scheduled, until_arrival,
                                   fuel, passengers );
planes.push_back( p_ptr );

// ... read in more information and then
p_ptr = new TakeoffPlane( id, scheduled, num_passengers );
planes.push_back( p_ptr );
```

- Notice: that the same pointer variable is used to point to objects of two different types.
- You could have also used `LandingPlane` and `TakeoffPlane` pointers and then pushed them onto the back of `planes`.

Accessing Objects Through a Polymorphic List of Pointers

- Suppose you are iterating through the list:

```
for ( std::list<Plane*>::iterator i = planes.begin(); i!=planes.e
{
    std::cout << "Delay = " << (*i)->delay() << std::endl;
}
```

Whose `delay` function is called?

- The answer depends on the type of plane object `*i` is pointing to:
 - If `*i` points to a `LandingPlane` object then the function defined in the `LandingPlane` class would be called.
 - If `*i` points to a `TakeoffPlane` object then the function defined in the `TakeoffPlane` class would be called.
 - This is an example of “run-time binding”.

Importantly, if `delay` had not been declared `virtual` then the function defined in `Plane` would be called!

- If you want to use a function in `LandingPlane`, such as `fuel_minutes_remaining`, which is not declared in `Plane` then you must “cast” the pointer:

```
for ( std::list<Plane*>::iterator i = planes.begin(); i!=planes.e
{
    std::cout << "Delay = " << (*i)->delay() << std::endl;
    LandingPlane * lp = dynamic_cast<LandingPlane*> ( *i );
    if ( lp )
        std::cout << "Fuel units remaining = "
                    << lp->fuel_minutes_remaining() << std::endl;
}
```

The pointer `lp` will be 0 if `*i` is not a `LandingPlane` object.

- In short, given a base class pointer, `p`
 - A call, `p->foo()`, when member function `foo` is declared `virtual` will be a call to the definition of `foo` provided in the *derived class* object `p` points to, but only if the derived class redefines `foo`.

- A call, `p->foo()`, when member function `foo` is **not** declared **virtual** will be a call to the definition of `foo` provided in the base class, even if the member function has overridden the definition of `foo`.

Exercise

What is the output of the following program?

```
#include <iostream>
using namespace std;

class Base {
public:
    Base() {}
    virtual void A() { cout << "Base A\n"; }
    void B() { cout << "Base B\n"; }
};

class One : public Base {
public:
    One() {}
    void A() { cout << "One A\n"; }
    void B() { cout << "One B\n"; }
};

class Two : public Base {
public:
    Two() {}
    void A() { cout << "Two A\n"; }
    void B() { cout << "Two B\n"; }
};

int main()
{
    Base* a[3];
    a[0] = new Base;
    a[1] = new One;
    a[2] = new Two;
    for ( unsigned int i=0; i<3; ++i )
    {
        a[i]->A();
        a[i]->B();
    }
}
```

```
    return 0;  
}
```

Course Summary

- Approach any problem by studying the requirements carefully, playing with hand-generated examples to understand them, and then looking for analogous problems that you already know how to solve.
- The standard library offers container classes and algorithms that simplify the programming process and raise your conceptual level of thinking in designing solutions to programming program.
 - Just think how much harder some of the homework problems would have been without generic container classes.
- When choosing between algorithms and between container classes (data structures) you should consider
 - efficiency,
 - naturalness of use, and
 - ease of programming.

All three are important

- Use classes with well-designed public member functions to encapsulate sections of code.
- Writing your own container classes and data structures usually requires building linked structures and managing memory through the big three:
 - copy constructor,
 - assignment operator, and
 - destructor.

Work hard to get these correct.

- When testing and debugging:
 - Test one function and one class at a time.
 - Figure out what your program actually does, not what you wanted it to do.
 - Always find the first mistake in the flow of your program and fix it before considering other apparent mistakes
 - Use small examples and boundary conditions when testing.

- Above all, remember the excitement and satisfaction of developing a deep, computational understanding of a problem and turning it into a program that realizes your understanding flawlessly.