

# CSCI-1200 Data Structures — Spring 2014

## Lecture 21 — C++ Inheritance and Polymorphism

### Review from Lecture 20

- Operators as non-member functions, as member functions, and as friend functions.
- STL Queues and Stacks
- What's a Priority Queue?
- A Priority Queue as a Heap, `percolate_up` and `percolate_down`
- A Heap as a Vector, Building a Heap, Heap Sort
- Merging heaps are the motivation for *leftist heaps*

### Today's Class

- Inheritance is a relationship among classes. Examples: bank accounts, polygons, stack & list
- Basic mechanisms of inheritance
- Types of inheritance
- Is-A, Has-A, As-A relationships among classes.
- Polymorphism

### 21.1 Motivating Example: Bank Accounts

- Consider different types of bank accounts:
  - Savings accounts
  - Checking accounts
  - Time withdrawal accounts (like savings accounts, except that only the interest can be withdrawn)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 members are placed in a **base class** and specialized members are placed in **derived classes**.

### 21.2 Accounts Hierarchy

- `Account` is the *base class* of the hierarchy.
- `SavingsAccount` is a *derived* class from `Account`. `SavingsAccount` has inherited member variables & functions and ordinarily-defined member variables & functions.
- The member variable `balance` in base class `Account` is **protected**, which means:
  - `balance` is NOT publicly accessible outside the class, but it is accessible in the derived classes.
  - if `balance` was declared as `private`, then `SavingsAccount` member functions could not access it.
- When using objects of type `SavingsAccount`, the inherited and derived members are treated exactly the same and are not distinguishable.
- `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.

## 21.3 Exercise: Draw the Accounts Class Hierarchy

```
#include <iostream>
// Note we've inlined all the functions (even though some are > 1 line of code)

// -----
class Account {
public:
    Account(double bal = 0.0) : balance(bal) {}
    void deposit(double amt) { balance += amt; }
    double get_balance() const { return balance; }

protected:
    double balance;    // account balance
};

// -----
class SavingsAccount : public Account {
public:
    SavingsAccount(double bal = 0.0, double pct = 5.0)
        : Account(bal), rate(pct/100.0) {}

    double compound() { // computes and deposits interest
        double interest = balance * rate;
        balance += interest;
        return interest;
    }
    double withdraw(double amt) { // if overdraft ==> return 0, else return amount
        if (amt > balance) {
            return 0.0;
        } else {
            balance -= amt;
            return amt;
        }
    }

protected:
    double rate;    // periodic interest rate
};

// -----
class CheckingAccount : public Account {
public:
    CheckingAccount(double bal = 0.0, double lim = 500.0, double chg = 0.5) : Account(bal) {
        limit = lim;
        charge = chg;
    }

    double cash_check(double amt) {
        assert (amt > 0);
        if (balance < limit && (amt + charge <= balance)) {
            balance -= amt + charge;
            return amt + charge;
        } else if (balance >= limit && amt <= balance) {
            balance -= amt;
            return amt;
        } else {
            return 0.0;
        }
    }

protected:
    double limit;    // lower limit for free checking
    double charge;   // per check charge
};
```

```
// -----
class TimeAccount : public SavingsAccount {
public:
    TimeAccount(double bal = 0.0, double pct = 5.0) : SavingsAccount(bal, pct) {
        funds_avail = 0.0;
    }

    // redefines 2 member functions from SavingsAccount
    double compound() {
        double interest = SavingsAccount::compound();
        funds_avail += interest;
        return interest;
    }
    double withdraw(double amt) {
        if (amt <= funds_avail) {
            funds_avail -= amt;
            balance -= amt;
            return amt;
        } else {
            return 0.0;
        }
    }
    double get_avail() const { return funds_avail; };

protected:
    double funds_avail; // amount available for withdrawal
};
```

## 21.4 Constructors and Destructors

- Constructors of a derived class *call the base class constructor* immediately, before doing ANYTHING else. The only thing you can control is which constructor is called and what the arguments will be. Thus when a `TimeAccount` is created 3 constructors are called: the `Account` constructor, then the `SavingsAccount` constructor, and then finally the `TimeAccount` constructor.
- The reverse is true for destructors: derived class constructors do their jobs first and then base class destructors are called at the, automatically. *Note: destructors for classes which have derived classes must be marked virtual for this chain of calls to happen.*

## 21.5 Overriding Member Functions in Derived Classes

- A derived class can 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).
- For example, see `TimeAccount::compound` and `TimeAccount::withdraw`.
- Once a function is redefined it is not possible to call the base class function, unless it is explicitly called as in `SavingsAccount::compound`.

## 21.6 Public, Private and Protected Inheritance

- 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 other options:
  - With protected inheritance, public members becomes protected and other members are unchanged
  - With private inheritance, all members become private.

## 21.7 Stack Inheriting from List

- For another example of inheritance, let's re-implement the `stack` class as a derived class of `std::list`:

```
template <class T>
class stack : private std::list<T> {
public:
    stack() {}
    stack(stack<T> const& other) : std::list<T>(other) {}
    virtual ~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. However, these member functions are still available to the member functions of the `stack<T>` class.
- Note: 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.

## 21.8 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”, for example: “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!

## 21.9 Exercise: 2D Geometric Primitives

Create a class hierarchy of geometric objects, such as: triangle, isosceles triangle, right triangle, quadrilateral, square, rhombus, kite, trapezoid, circle, ellipse, etc. How should this hierarchy be arranged? What member variables and member functions should be in each class?

## 21.10 Note: Multiple Inheritance

- When sketching a class hierarchy for geometric objects, you may have wanted to specify relationships that were more complex... in particular some objects may wish to inherit from *more than one base class*.
- This is called *multiple inheritance* and can make many implementation details significantly more hairy. Different programming languages offer different variations of multiple inheritance.

## 21.11 Introduction to Polymorphism

- Let's consider a small class hierarchy version of polygonal objects:

```
class Polygon {
public:
    Polygon() {}
    virtual ~Polygon() {}
    int NumVerts() { return verts.size(); }
    virtual double Area() = 0;
    virtual bool IsSquare() { return false; }
protected:
    vector<Point> verts;
};

class Triangle : public Polygon {
public:
    Triangle(Point pts[3]) {
        for (int i = 0; i < 3; i++) verts.push_back(pts[i]); }
    double Area();
};

class Quadrilateral : public Polygon {
public:
    Quadrilateral(Point pts[4]) {
        for (int i = 0; i < 4; i++) verts.push_back(pts[i]); }
    double Area();
    double LongerDiagonal();
    bool IsSquare() { return (SidesEqual() && AnglesEqual()); }
private:
    bool SidesEqual();
    bool AnglesEqual();
};
```

- Functions that are common, at least have a common interface, are in `Polygon`.
- 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.

## 21.12 A Polymorphic List of Polygon Objects

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

```
std::list<Polygon*> polygons;
```

- Objects are constructed using `new` and inserted into the list:

```
Polygon *p_ptr = new Triangle( .... );
polygons.push_back(p_ptr);
p_ptr = new Quadrilateral( ... );
polygons.push_back(p_ptr);
Triangle *t_ptr = new Triangle( .... );
polygons.push_back(t_ptr);
```

Note: We've used the same pointer variable (`p_ptr`) to point to objects of two different types.

## 21.13 Accessing Objects Through a Polymorphic List of Pointers

- Let's sum the areas of all the polygons:

```
double area = 0;
for (std::list<Polygon*>::iterator i = polygons.begin(); i!=polygons.end(); ++i)
    area += (*i)->Area();
```

Which `Area` function is called? If `*i` points to a `Triangle` object then the function defined in the `Triangle` class would be called. If `*i` points to a `Quadrilateral` object then `Quadrilateral::Area` will be called.

- Here's code to count the number of squares in the list:

```
int count = 0;
for (std::list<Polygon*>::iterator i = polygons.begin(); i!=polygons.end(); ++i)
    count += (*i)->IsSquare();
```

If `Polygon::IsSquare` had not been declared `virtual` then the function defined in `Polygon` would always be called! In general, given a pointer to type `T` we start at `T` and look “up” the hierarchy for the closest function definition (this can be done at compile time). If that function has been declared `virtual`, we will start this search instead at the actual type of the object (this requires additional work at runtime) in case it has been redefined in a derived class of type `T`.

- To use a function in `Quadrilateral` that is not declared in `Polygon`, you must “cast” the pointer. The pointer `*q` will be `NULL` if `*i` is not a `Quadrilateral` object.

```
for (std::list<Polygon*>::iterator i = polygons.begin(); i!=polygons.end(); ++i) {
    Quadrilateral *q = dynamic_cast<Quadrilateral*> (*i);
    if (q) std::cout << "diagonal: " << q->LongerDiagonal() << std::endl;
}
```

## 21.14 Exercise

What is the output of the following program?

```
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;
}
```