LECTURE 8

Inheritance

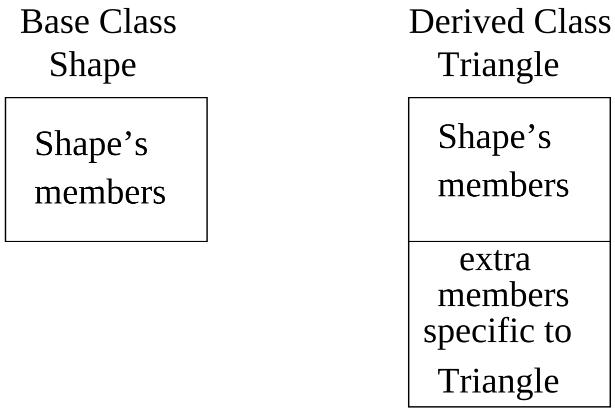
A concept does not exist in isolation. It coexists with related concepts and derives much of its power from relationships with related concepts. For example, try to explain what a car is. Soon you're talking about wheels, engines, drivers, roads, trucks, gas, etc. Since we use classes to represent concepts, we need to address how to represent relationships between classes that reflects the relationship between concepts. Classes can have features in common, and the notion of a hierarchical relationship between classes is the basis of inheritance and derived classes. For example, the concepts of a circle and a triangle are related in that they are both shapes; that is, they have the concept of a shape in common. Thus, we can explicitly define class `Circle` and class `Triangle` to have class `Shape` in common. Representing a circle and a triangle in a program without involving the notion of a shape would be to lose something essential.

When we express these kinds of relationships in terms of classes, we talk about a *base class* and *derived classes*. The `Shape` class would be the base class and the `Circle` and `Triangle` classes are derived from `Shape`. A derived class is larger than its base class in the sense that it holds more data and provides more functions. The basic syntax looks like this:

```
class Shape                      //base class definition
 {....};

class Triangle: public Shape     //Definition of derived class with
 { .                             //Shape as its base class
   .
   .
 };
```

The class `Triangle` includes `Shape`'s members plus any members that are specific to `Triangle`.



One of the benefits of using inheritance is that we can avoid replicated code. In other words we don't have to copy all the data members and member functions of `Shape` into

`Triangle` and every other special type of `Shape` that comes along, e.g., `Circle`, `Square`, etc. There are several advantages to this:

1. If you have replicated code, you could make changes made on one part of the code and forget to make the same changes elsewhere.
2. If you have replicated code, you have to check each part to make sure they're the same.
3. Some of the replicas may not be used much so bugs won't show up right away.

Another benefit of inheritance is `reuse` of code. You don't have to write programs from scratch everytime you want a program that is somewhat different; you can use the base classes and put the modifications in new derived classes.

Using a class as a base is equivalent to declaring an (unnamed) object of that base class. Consequently, a class must be defined in order to be used as a base:

```
class Shape;                      //base class declaration, no definition

class Triangle: public Shape     //Error: Shape not defined
 { . . .
 };
```

Another example is having `Employee` as the base class and `Manager` as the derived class or subclass. (Derived class and subclass are synonyms.) A manager is also an employee, but a manager has more attributes. We say that a `Manager IS_A Employee`; a `Manager` inherits the attributes of an `Employee`. Deriving `Manager` from `Employee` in this way means that `Manager` can be used wherever an `Employee` is acceptable. For example:

```
class Employee
{....};
class Manager: public Employee
{....};

void func(Employee &emp)
{ whatever func does }

void main()
{
  Manager boss;
  func(boss);
}
```

`func` expects to receive as its argument an object of class `Employee`. The call from `main()` passes it an object of class `Manager`. This is fine because a `Manager IS_A Employee`.

Functions

A derived class has all of a base class' functions. A base class function can be redefined in the derived class. Let's call this function `f1`. Then the derived class will see the derived class' version of `f1` unless you specifically call the base class' `f1` using the scope resolution operator `::`. For example,

```
class Employee
{ ....
  f1();                              //Employee's version of f1
};
class Manager: public Employee
{ ....
  f1();                              //Manager's version of f1
  f2()
  {....
      f1();                          //Manager's version of f1
      Employee::f1();                //Employee's version of f1
      Manager::f1();                 //Manager's version of f1
  }
};
```

Public, Protected, and Private Members

There are 3 types of members that a class can have. So far we have been using public and private catagories. There is also `protected`. A member of a derived class can use the public and protected members of its base class as if they were declared in the derived class itself. However, a derived class cannot use a base class' private names. A protected member is like a public member to a member of a derived class, yet it is like a private member to other functions.

```
class Employee
{
 private:
    float private_member;
 protected:
    float protected_member;
 public:
    float public_member;
 ....
};
class Manager: public Employee
{ ....
  func()
  { float x;
```

```
      x = private_member;                    //ERROR: illegal access
      x = protected_member;                  //legal access
      x = public_member;                     //legal access
    }
 };
 main()
 {
    float x;
    Manager boss;
     x = boss.private_member;                 //ERROR: illegal access
     x = boss.protected_member;               //ERROR: illegal access
     x = boss.public_member;                  //legal access
 }
```

Constructors

Some derived classes need constructors. When do they need them? If the base class has a constructor, then that constructor must be called; and if that constructor needs arguments, then such arguments must be provided. Arguments for the base class' constructor are specified in the definition of a derived class' constructor just like any class member of the derived class. For example:

```
class Employee {
   private:
    char name[40];
    short department;
   public:
    Employee(const char *pName, int dept): name(pName), department(dept)
    { ... }
};
class Manager: public Employee {
   private:
    short level;
   public:
    Manager(const char *pName, int dept, int lvl): Employee(pName, dept),
      level(lvl)
    { ...}
};
```

Constructors (and assignment operators) are not inherited. A derived class constructor can specify initializers for its own members and immediate bases only; it cannot directly initialize members of a base class. For example the following constructor for `Manager` has several mistakes:

```
Manager::Manager(const char *pName, int dept, int lvl):
```

```
    name(pName),                     //ERROR: name not declared in Manager
    department(dept),                //ERROR: department not declared in
                                     //Manager
    level(lvl)
    { ... }
```

This definition contains three errors: it fails to invoke `Employee`'s constructor, and twice it attempts to initialize members of `Employee` directly.

Class objects are constructed from the bottom up: first the base class, then the members, and then the derived class itself. They are destroyed in the opposite order: first the derived class itself, then the members, and then the base class. Members and bases are constructed in order of declaration in the class and destroyed in the reverse order. For example,

```
class Teacher;
class Student;
class Curriculum;
class Tutoree;
class Tutor: public Teacher, public Student
  {
    private:
      Curriculum curric;
      Tutoree tut;
    public:
      Tutor(TeacherArguments, StudentArguments, CurriculumArguments,
            TutoreeArguments):
            tut(TutoreeArguments)
            curric(CurriculumArguments),
            Student(StudentArguments),
            Teacher(TeacherArguments),
            {}
  };
```

When an object of `Tutor` is constructed, `Teacher` is constructed first, then `Student`, then `curric`, and finally `tut`. I am assuming that the constructor for `Teacher` requires "`TeacherArguments`", the constructor for `Student` requires "`StudentArguments`", etc.

Copying

Copying of class objects is defined by the copy constructor and assignment operators. These are not inherited from the base class. Consider:

```
class Employee {
   public:
     Employee& operator=(const Employee&);
     Employee(const Employee&);
```

```
};
void func(const Manager& boss)
{
   Employee emp = boss;       //construct emp from Employee part of boss
   emp = boss;                //assign Employee part of boss to emp
}
```

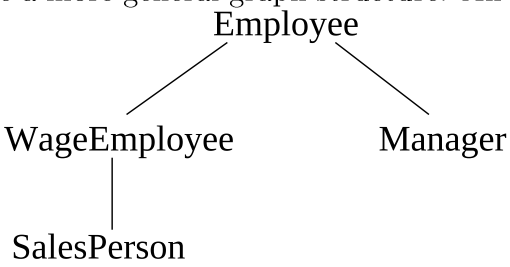
Because the `Employee` copy functions do not know anything about `Managers`, only the `Employee` part of a `Manager` is copied. This is commonly referred to as *slicing* and can be a source of surprises and errors. One reason to pass pointers and references to objects of classes is to avoid slicing. Other reasons are to preserve polymorphic behavior (which we will discuss) and to gain efficiency.

Class Hierarchies

A derived class can itself be a base class. For example:

```
class Employee
 { ...};
class Manager: public Employee
 { ...};
class Director: public Manager
 { ...};
```

Such a set of related classes is called a *class hierarchy.* Such a hierarchy is most often a tree, but it can also be a more general graph structure. An example of a tree structure is



This is the hierarchical structure that we will use in the following example. Suppose you want a database of employees in a firm, and you need to compute what to pay them. However, the employees are grouped in a variety of ways: some are paid hourly, some by commisions, etc. All have names and social security numbers, but only some have variables related to sales, etc.

The information you need is different for each type of employee. Also, there are subtypes for each type. One data structure can't hold it all, but they all have to be grouped into the same database. The solution is inheritance.

```
class Employee                       //common to all classes
    {
public:
    Employee();
```

```
    Employee(const char* nm);
    char *getName() const;
private:
    char name[30];
    };

class WageEmployee : public Employee
    {
public:
    WageEmployee(const char* nm);
    void setWage(float wg);
    void setHours(float hrs);
    float computePay() const;
private:
    float wage;
    float hours;
    };

class SalesPerson : public WageEmployee
    {
public:
    SalesPerson(const char* nm);
    void setCommission(float comm);
    void setSales(float sales);
    float computePay() const;
private:
    float commission;
    float salesMade;
    };

class Manager : public Employee
    {
public:
    Manager(const char* nm);
    void setSalary(float salary);
    float computePay() const;
private:
    float weeklySalary;
    };
```

`computePay` for `Manager` and `WageEmployee` are no problem:

```
float WageEmployee::computePay() const
```

```
    {
    return wage*hours;
    }

float Manager::computePay() const
    {
    return weeklySalary;
    }

// But computePay is a problem for SalesPerson:

float SalesPerson::computePay() const
    {
    return wage*hours + commission * salesMade;  //Error: hours and
                                                 //wages are private members
    }                                            //of WageEmployee

float SalesPerson::computePay() const
    {
    return computePay() + commission * salesMade; // Bad recursive call
    }

float SalesPerson::computePay() const
    {
    return WageEmployee::computePay() +
     commission * salesMade;  // OK
    }


//Constructors:
//-------------
WageEmployee::WageEmployee(const char* nm)
    : Employee(nm)
    {
    wage = 0;
    hours = 0;
    }

SalesPerson::SalesPerson(const char* nm)
    : WageEmployee(nm)
    {
    commission = 0;
```

```
salesMade = 0;
}
```

Conversions between Base and Derived Classes

We can always convert a derived class object to a base class object, but we can't do the reverse.

```
WageEmployee aWorker;
SalesPerson aSeller("John Smith");
aWorker = aSeller; // OK, copies the WageEmployee part of aSeller
// into aWorker
// All SalesPersons are WageEmployees

aSeller = aWorker ; // Error, can't convert
// Not all WageEmployees are SalesPersons

-------------------
Employee *empPtr;
WageEmployee aWorker("Bill Shapiro");
SalesPerson aSeller("John Smith");
Manager aBoss("Mary Brown");

empPtr = &aWorker; // OK
empPtr = &aSeller; // OK
empPtr = &aBoss; // OK, They are all employees
```

When you refer to member objects through the pointer, what you get depends on the type of the pointer.

```
SalesPerson aSeller("John Smith");
SalesPerson *salePtr;
WageEmployee *wagePtr;

salePtr = &aSeller;
wagePtr = &aSeller;

wagePtr->setHours(40.0); // call WageEmployee::setHours
salePtr->setWage(6.0); // call WageEmployee::setWage
wagePtr->setSales(1000.0); // error, no WageEmployee::setSales

salePtr->setSales(1000.0); // call SalesPerson::setSales
salePtr->setCommission(0.05); // call SalesPerson::setCommission
float base,total;
base = wagePtr->computePay();   // applies to John Smith
total = salePtr->computePay();  // applies to John Smith
```

To convert pointers in the wrong way, you must explicitly use a cast:

```
salePtr = (SalesPerson *)wagePtr;
```

This is legal, but dangerous since `*wagePtr` might not be a `SalesPerson`.

Public, Protected and Private Inheritance

There are other types of inheritance, namely protected and private. These are rarely ever used, but just for the record here is a table listing the access modes in the derived class for the different access modes in the base class. There is also an example illustrating the access to members of the base class through the derived class.

```
                    Public Access     Protected Access     Private Access
  Inheritance       Mode in Base      Mode in Base         Mode in Base
     Type              Class             Class             Class
---------------------------------------------------------------------------

   public            public            protected            private

   protected         protected         protected            private

   private           private           private              private
***************************************************************************

    class Employee
    {
     private:
        float private_member;
     protected:
        float protected_member;
     public:
        float public_member;
      ....
    };
    class Secretary: protected Employee
    { ....
      func()
      { float x;
        x = private_member;                    //ERROR: illegal access
        x = protected_member;                  //legal access
        x = public_member;                     //legal access
      }
    };
    class SuperSecretary: public Secretary
```

```
{
  func()
  { float x;
    x = private_member;                    //ERROR: illegal access
    x = protected_member;                  //legal access
    x = public_member;                     //legal access
  }
};

class SalesPerson: private Employee
{ ....
  func()
  { float x;
    x = private_member;                     //ERROR: illegal access
    x = protected_member;                   //legal access
    x = public_member;                      //legal access
  }
};
```

```
// Classes derived from SalesPerson view Employee's data members as private
// and cannot access Employee's data members.
```

```
class SuperSales: public SalesPerson
{ ....
  func()
  { float x;
    x = private_member;                     //ERROR: illegal access
    x = protected_member;                   //ERROR: illegal access
    x = public_member;                      //ERROR: illegal access
  }
};


main()
{
   float x;
   Secretary typist;
    x = typist.private_member;               //ERROR: illegal access
    x = typist.protected_member;             //ERROR: illegal access
    x = typist.public_member;                //ERROR: illegal access
   SalesPerson pushy;
    x = pushy.private_member;                //ERROR: illegal access
```

```
    x = pushy.protected_member;              //ERROR: illegal access
    x = pushy.public_member;                 //ERROR: illegal access

}
```

The default for inheritance is private. It's generally a good idea to declare the type of inheritance explicitly.

It is important to note that a privately or protectedly derived class cannot do all the things that the base class can. Therefore, a non–publically derived object cannot be used as a replacement for a base class object. This is why public inheritance is the most popular. Consider the following example:

```
class Employee
{
 public:
    void earn_pay();
};
class Manager: public Employee {};
class SalesPerson: private Employee {};

void func(Employee &emp)
{
    emp.earn_pay();
}
main()
{
   Manager boss;
   SalesPerson pushy;
   func(boss);                        //legal
   func(pushy);                       //ERROR: illegal
}
```

The function `func` expects an object of type `Employee`, but the call `func(boss)` is actually passing an object of class Manager. This is fine because `Manager IS_A Employee` and offers the same access to its members as `Employee`. However, this isn't true for the SalesPerson `pushy`. In this case the external function `func` doesn't have access to `pushy.earn_pay()`, because that `pushy.earn_pay()` is not publically accessible.