# Inheritance

* Inheritance allows classes to inherit members (data & methods) from other classes – base classes.
* Inheritance lets you create class hierarchies
* Inheritance provides reusability. Use of vendor supplied *class libraries.*
* Extend & modify classes via inheritance – derive new classes from old ones.
* You extend a base class by adding data members and/or member functions in the derived class

## Syntax

*class* derived\_class : *public* base\_class

{

< class definition >

};

This declares that the class derived\_class inherits the members of the class base\_class.

Example:

class Employee {

public:

Employee();

Employee(const char \*);

char\* getName() const;

private:

char name[32];

};

class WageEmployee : public Employee {

public:

WageEmployee (const char \*);

void setWage (float);

void setHours(float);

private:

float wage;

float hours;

};

Employee is the *Base class*. WageEmployee is the *Derived class.*

The keyword *public* specifies public derivation, meaning that the

public members of the base class are public members of the derived class, and the protected members of the base class are protected members of the derived class.

The Derived class inherits the base class interface. This is the is-arelationship.

Derivation can be *private*, as in

class WageEmployee : private Employee {

…..

….

};

* With private inheritance, the public methods of the base class become private methods of the derived class.
* Those members are accessible to the derived class’s member functions, but not to anyone using the derived class.
* In other words, the derived class does not inherit the base class interface. This is the has-arelationship.
* Besides **public** & **private** inheritance, there is **protected** derivation.
* With protected inheritance, public & protected members of a base

class become protected members of the derived class.

class WageEmployee : protected Employee {

< class definition >

};

## Varieties of Inheritance

|  |  |  |  |
| --- | --- | --- | --- |
| **BASE CLASS MEMBER**  **ACCESS SPECIFIER** | **PUBLIC INHERITANCE** | **PROTECTED INHERITANCE** | **PRIVATE INHERITANCE** |
| P  U  B  L  I  C | **Public in derived class**  Can be accessed directly by member functions, friend functions and non-member functions | **Protected in derived class**  Can be accessed directly by member functions and friend functions. | **Private in derived class**  Can be accessed directly by member functions and friend functions |
| P  R  O  T  E  C  T  E  D | **Protected in derived class**  Can be accessed directly by member functions and friend functions | **Protected in derived class**  Can be accessed directly by member functions and friend functions. | **Private in derived class**  Can be accessed directly by friend functions and member functions |
| P  R  I  V  A  T  E | **Hidden in derived class**  Can be accessed by member functions and friend functions through **public** or **protected** member functions of the base class | **Hidden in derived class**  Can be accessed by member functions and friend functions through public or protected member functions of the base class. | **Hidden in derived class**  Can be accessed by member functions and friend functions through public or protected member functions of the base class |

**Redefining access**

* Selectively redefine visibility of individual methods from base classes that are derived private

class A

{

public:

int f();

int g\_;

…

private:

int p\_;

};

class B : private A

{

public:

int A::f; // make it public

protected:

int A::g; // make it protected

};

* Cannot increase the access of an inherited method in a derived class. That is cannot have:

class B : private A {

public:

int A::p\_;

};

* Cannot selectively decrease the access of base class methods in a publicly derived class. That is, cannot do

class B : public A {

private:

int A::f;

};

# Initializer Lists

* Construction of objects for derived classes involves
  + - Invoking the constructors for the base class(es) in declaration order
    - Invoking constructors for members of the class in declaration order
    - Invoking the derived constructor
* Derived class constructor may pass arguments to a base class constructor in the same list as member initializers

Example:

class Employee

{

public:

Employee ();

Employee (const char \*);

char \* getName() const;

private:

char name[32];

};

Employee::Employee()

{

name[0] = ‘\0’;

}

Employee::Employee( const char \* N

{

strcpy (N, name);

}

inline char \* Employee::getName() const

{

return name;

}

class WageEmployee : public Employee {

public:

WageEmployee (const char \*);

void setWage (float w) { wage = w; }

void setHours(float h) { hours = h; }

float computePay() const;

private:

float wage;

float hours;

};

wageEmployee::wageEmployee(const char \* N): Employee(N)

{

wage = 0.0; hours = 0.0;

}

float wageEmployee::computePay() const

{

return wage \* hours;

}

class SalesPerson : public WageEmployee {

public:

SalesPerson (const char \* N) : wageEmployee(N)

{

commission = salesMade = 0.0;

}

void setCommision (float C)

{

commission = C;

}

void setSales (float S)

{

salesMade = S;

}

float computePay () const;

private:

float commission;

float salesMade;

};

float salesPerson::computePay() const

{

return wageEmployee::computePay() +

commission \* salesMade;

}

class Manager : public Employee

{

public:

Manager(const char \* N):Employee(N), weeklySalary(0.0) {}

void setSalary (float S)

{

weeklySalary = S;

}

float computePay () const

{

return weeklySalary;

}

private:

float weeklySalary;

};

# Redefining members of the Base class

float WageEmployee::computePay () const

{

return wage \* hours;

}

// What’s wrong with this?

float SalesPerson::computePay () const

{

return hours \* wage + commission \* salesMade;

}

// Derived class’s computepay() calls base class’s

// version

float SalesPerson::computePay () const

{

return WageEmployee::computePay() +

commission \*salesMade;

}

SalesPerson SP (“Joe Doe”);

SP.setHours(40.0);

SP.setWage(6.0);

SP.setCommission(0.05);

SP.setSales(2000.0)

cout << Sales Person’s salary: “ << SP.computePay() << endl;

In this case the Sales Person’s computePay() is called. To call the

Base class’s version of computePay(), use the scope operator, as in

cout << Sales Person’s salary: “ << SP.wageEmployee::computePay() << endl;

Define a computePay() method for the Manager class.

float Manager::computePay()

{

return weeklySalary;

}

# Class hierarchy (aka inheritance hierarchy)

Employee // Base class

wageEmployee Manager // Derived classes, a base class

SalesPerson // Derived class

* An instance of a derived class contains all the members of the base class, and all those members must be initialized. So the base class’s constructor has to be called by the derived class’s constructor.
* Specify the *base initializer* with the derived class constructor

using a syntax similar to the *member initializer list.*

* The base class constructor is run first followed by the constructors for member objects and finally the derived class constructor is run.

## Conversions between base & derived classes

* Since a salesPerson *IS-A* wageEmployee, a salesPerson object can be used whenever a wageEmployee object is needed.
* To support this relationship, you convert a derived class reference or pointer to a base class reference or pointer without an explicit type cast. This is called *upcasting.*
* In the above example, you can use a pointer to an Employee object to point to a wageEmployee object or a Manager object.

Employee \*eptr;

WageEmployee we (“Bill Clinton”);

SalesPerson sp (“John Doe”);

Manager boss (“Jane Fonda”);

eptr = &we; // convert wageEmployee \* to Employee\*

eptr = &sp; // convert SalesPerson \* to Employee \*

eptr = &boss; // convert Manager \* to Employee \*

* When you refer an object thro’ a pointer, the type of the pointer determines which member functions you can call. If you refer to a derived class object with a base class pointer, you can only call base class functions.

SalesPerson sp (“John Doe”);

SalesPerson \*saleptr;

WageEmployee \*wageptr;

saleptr = &sp;

wageptr = &sp;

wageptr->setHours(40.0); // call WageEmployee::setHours

saleptr->setWage(6.0); //call WageEmployee::setWage

wageptr->setSales(100.0); //error. Why?

saleptr->setSales(1000.0); // call salesPerson::setSales

* To perform the conversion from a pointer to a base class to a pointer to a derived class, you must use an explicit cast, as in:

wageEmployee \*wageptr = &sp;

salesPerson \*saleptr;

saleptr = (salesPerson \*)wageptr; // base -> derived

This is dangerous.

* Conversion from a derived class pointer to a base class pointer is very useful.

Example:

You want to write a function that expects a pointer to an *Employee* as a parameter. You can pass a pointer to any type of employee.

class EmployeeList

{

public:

EmployeeList();

Add (Employee \*);

…

private:

…

};

// Using add() you can insert any type of employee into the   
// EmployeeList object

EmployeeList Dept;

wageEmployee \*wp;

salesPerson \*sp;

Manager \*mp;

wp = new wageEmployee (“Bill Smith”);

sp = new salesPerson (“Michael Jordan”);

mp = new Manager (“Jane Fonda”);

// Add them to the list

Dept.add(wp);

Dept.add(sp);

Dept.add(mp);

// Manipulate the list of Employees. Iterate thro all the elements

// in the EmployeeList object passed to the function. The iterator // returns a ptr to an Employee regardless of the type. Using the // pointer, the function prints the employees name.

void printNames (EmployeeList &dept)

{

int count = 0;

Employee \*person;

EmpIter Iter (dept);

while ( person = Iter.getNext() )

{

count++;

count << count << ‘ ‘ << person->getName() << endl;

}

}

* Can you compute the weekly salary of each employee using this technique?
* You could call the computePay() functions defined in the derived classes. But the function that will get called using the *Employee* pointers would be the *Employee* computePay() function. Why?
* This is not correct. We need to call each class’s version of

computePay() function while still using *Employee* pointers. You can do that by using **Virtual functions.**

* What is a **Virtual function?** A virtual function is a member

function that is expected to be redefined in derived classes. When you call a virtual function thro’ a pointer to a base class, the derived class’s version of the function is executed.

* How do you specify a virtual function? Place the keyword **virtual** before the declaration of the member function in the base class.
* The virtual keyword is optional in the declarations in the derived classes.

# Polymorphism

The ability to call member functions for an object without specifying the object’s exact type is *polymorphism.* That is, the ability to have a single statement invoke many different functions. Or, the ability to operate on & manipulate different derived objects in a uniform way is called polymorphism.

In the example, the pointer *person* can point to any type of employee, and the name *computePay* can refer to any of the salary computation functions.

Inheritance alone does not provide *polymorphic* behavior. The derived members of a class are not accessible thro’ a base class pointer. Using inheritance alone, the client (application) cannot access the derived class members using base class pointer.

C++ provides *polymorphic* behavior thro’ the use of inheritance & virtual functions. The virtual keyword tells the compiler that the member function is a special type of function and is to provide polymorphic behavior.

The virtual functions do not have a meaningful implementation at the base class level. At this point, these functions have been implemented as do-nothing functions. A better solution is to use *abstract classes*.

## Dynamic Binding

At compile time, compiler does not know what function is called by the

statement *person->computePay().* The reason is that this might refer to any

of the several different functions. The compiler evaluates the statement at

run time, when it can determine the type of object *person* points to. This is

known as **late binding** or **dynamic binding.**

Function calls in C and nonvirtual function calls in C++ are evaluated at compile time. The address of the function is fixed and known at compile time. This is known as **early binding** or **static binding.**

With dynamic binding you can make an existing module handle new types

without having to modify the source & recompile it. For ex., the function

computePayroll() can be used without modification to calculate the pay for

a new type of employee called Consultant which is derived from the

Employee class.

## Virtual Destructor

When destroying dynamically allocated objects with the **delete** operator, a problem can arise if delete is applied to a base class pointer. The compiler calls the base class destructor even if the pointer points to an instance of the derived class.

In order to call the appropriate destructor, the base class’s destructor must be declared as **virtual.** This causes the destructors of all derived classes to be virtual, even though they don’t share the same name as the base class’s destructor. If delete is applied to a base class pointer, the appropriate destructor is called regardless of the type of the object the

pointer is pointing to. In the example, *Employee* has a virtual destructor.

*If a class has virtual functions, it should have a virtual destructor even if the class doesn’t need one.*

# Multiple Inheritance

* A class is derived from more than one base class.

class SalesManager : public salesPerson, public Manager

{

…

};

* salesPerson and Manager are *direct base classes* of SalesManager.
* You cannot specify a class as a direct base class more than once. However, a class can be an *indirect base class* more than once.
* In the above example, SalesManager has Employee has an indirect base class twice – once thro salesPerson and once thro Manager. So SalesManager contains 2 copies of class Employee’s data members.
* How do you call the indirect base class’s member functions?

SalesManager Sboss;

char \* str;

str = Sboss.getName(); // But this is ambiguous.

Which copy of Employee should be used? You must specify this.

str = Sboss.Manager::getName();

// uses Manager’s copy of Employee data members

* To avoid the duplication of base class, you make the base class a *virtual base class*. For example,

class wageEmployee : public virtual Employee

{

…

};

class Manager : public virtual Employee

{

….

};

* wageEmployee and Manager need to use the **virtual** keyword.
* SalesPerson and SalesManager do not, because Employee is an indirect base class for them.

**Abstract Base Class**

* An abstract base class is an abstract concept. An Employee is an abstract concept. An Employee exists only as a wageEmployee or a Manager or a … but never just an Employee.
* This means that you cannot make an object of (or you cannot instantiate) an abstract base class.
* A wageEmployee is an example of a *concrete class.*
* *What makes a class an Abstract Base Class?*
* A class that one or more *pure virtual member functions*is an ***Abstract base class****.*

Example:

class Shape {

public:

….

virtual Shape \*draw()const = 0;//draw is a Pure Virtual function

…..

};

You implement the draw() function in a derived class.

class Circle : virtual public Shape

{

public:

….

virtual Shape \*draw() const

{

cout << drawing a circle << endl;

….

}

….

};