Overview

15.1 Inheritance Basics
15.2 Inheritance Details
15.3 Polymorphism
15.1 Inheritance Basics
Inheritance Basics

Inheritance is the process by which a new class, called a **derived class**, is created from another class, called the **base class**

- A derived class automatically has all the member variables and functions of the base class

- A derived class can have additional member variables and/or member functions

- The derived class is a **child of the base or parent class**
Employee Classes

- To design a record-keeping program with records for salaried and hourly employees...
  - Salaried and hourly employees belong to a class of people who share the property "employee"
  - A subset of employees are those with a fixed wage
  - Another subset of employees earn hourly wages

- All employees have a name and SSN
  - Functions to manipulate name and SSN are the same for hourly and salaried employees
A Base Class

- We will define a class called Employee for all employees

- The Employee class will be used to define classes for hourly and salaried employees
Interface for the Base Class Employee

// This is the header file employee.h.
// This is the interface for the class Employee.
// This is primarily intended to be used as a base class to derive
// classes for different kinds of employees.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

namespace employeessavitch
{

    class Employee
    {
    public:
        Employee();
        Employee(string the_name, string the_ssn);
        string get_name() const;
        string get_ssn() const;
        double get_net_pay() const;
        void set_name(string new_name);
        void set_ssn(string new_ssn);
        void set_net_pay(double new_net_pay);
        void print_check() const;
    
    private:
        string name;
        string ssn;
        double net_pay;
    
    };

}//employeessavitch

#endif //EMPLOYEE_H
// This is the file: employee.cpp.
// This is the implementation for the class Employee.
// The interface for the class Employee is in the header file employee.h.
#include <string>
#include <cstdlib>
#include <iostream>
#include "employee.h"
using namespace std;

namespace employeessavitch
{
    Employee::Employee( ) : name("No name yet"), ssn("No number yet"), net_pay(0)
    {
        // deliberately empty
    }

    Employee::Employee(string the_name, string the_number)
        : name(the_name), ssn(the_number), net_pay(0)
    {
        // deliberately empty
    }

    string Employee::get_name( ) const
    {
        return name;
    }

    string Employee::get_ssn( ) const
    {
        return ssn;
    }
}
Implementation for the Base Class Employee (part 2 of 2)

```cpp
double Employee::get_net_pay() const
{
    return net_pay;
}

void Employee::set_name(string new_name)
{
    name = new_name;
}

void Employee::set_ssn(string new_ssn)
{
    ssn = new_ssn;
}

void Employee::set_net_pay(double new_net_pay)
{
    net_pay = new_net_pay;
}

void Employee::print_check() const
{
    cout << "\nERROR: print_check FUNCTION CALLED FOR AN \n" << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n" << "Check with the author of the program about this bug.\n";
    exit(1);
}

} // employeeessa_vitch
```
Function print_check

Function print_check will have different definitions to print different checks for each type of employee

- An Employee object lacks sufficient information to print a check

- Each derived class will have sufficient information to print a check
Class HourlyEmployee

- HourlyEmployee is derived from Class Employee
  - HourlyEmployee inherits all member functions and member variables of Employee
  - The class definition begins

```cpp
class HourlyEmployee : public Employee

:public Employee shows that HourlyEmployee is derived from class Employee
```

- HourlyEmployee declares additional member variables wage_rate and hours
interface for the derived class HourlyEmployee

// This is the header file hourlyemployee.h.
// This is the interface for the class HourlyEmployee.
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H

#include <string>
#include "employee.h"

using namespace std;

namespace employeessavitch
{
    class HourlyEmployee : public Employee
    {
    public:
        HourlyEmployee();
        HourlyEmployee(string the_name, string the_ssn,
                        double the_wage_rate, double the_hours);
        void set_rate(double new_wage_rate);
        double get_rate() const;
        void set_hours(double hours_worked);
        double get_hours() const;
        void print_check();
    private:
        double wage_rate;
        double hours;
    }
}

}//employeessavitch

#endif //HOURLYEMPLOYEE_H
Inherited Members

A derived class inherits all the members of the parent class

- The derived class does not re-declare or re-define members inherited from the parent, except...
- The derived class re-declares and re-defines member functions of the parent class that will have a different definition in the derived class
- The derived class can add member variables and functions
Implementing a Derived Class

- Any member functions added in the derived class are defined in the implementation file for the derived class
  
  - Definitions are not given for inherited functions that are not to be changed
The class SalariedEmployee is also derived from Employee

- Function print_check is redefined to have a meaning specific to salaried employees

- SalariedEmployee adds a member variable salary
Interface for the Derived Class SalariedEmployee

//This is the header file salariedemployee.h.
//This is the interface for the class SalariedEmployee.
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <string>
#include "employee.h"

using namespace std;

namespace employeessavitch
{

class SalariedEmployee : public Employee
{
public:
    SalariedEmployee();
    SalariedEmployee (string the_name, string the_ssn,
                      double the_weekly_salary);

    double get_salary( ) const;
    void set_salary(double new_salary);
    void print_check( );

private:
    double salary; //weekly
};
}

#endif //SALARIEDEMPLOYEE_H
Implementation for the Derived Class SalariedEmployee (part 1 of 2)

// This is the file salariedemployee.cpp.
// This is the implementation for the class SalariedEmployee.
// The interface for the class SalariedEmployee is in
// the header file salariedemployee.h.
#include <iostream>
#include <string>
#include "salariedemployee.h"
using namespace std;

namespace employeessavitch
{
    SalariedEmployee::SalariedEmployee( ) : Employee( ), salary(0)
    {
        // deliberately empty
    }

    SalariedEmployee::SalariedEmployee(string the_name, string the_number,
        double the_weekly_salary)
        : Employee(the_name, the_number), salary(the_weekly_salary)
    {
        // deliberately empty
    }

    double SalariedEmployee::get_salary( ) const
    {
        return salary;
    }

    void SalariedEmployee::set_salary(double new_salary)
    {
        salary = new_salary;
    }
}
Implementation for the Derived Class SalariedEmployee (part 2 of 2)

```cpp
void SalariedEmployee::print_check()
{
    set_net_pay(salary);
    cout << 
    cout << "Pay to the order of " << get_name() << endl;
    cout << "The sum of " << get_net_pay() << " Dollars\n";
    cout << 
    cout << "Check Stub NOT NEGOTIABLE \n";
    cout << "Employee Number: " << get_ssn() << endl;
    cout << "Salaried Employee. Regular Pay: "
    << salary << endl;
    cout << 
}
```
Parent and Child Classes

- Recall that a child class automatically has all the members of the parent class.
- The parent class is an ancestor of the child class.
- The child class is a descendent of the parent class.
- The parent class (Employee) contains all the code common to the child classes.
  - You do not have to re-write the code for each child.
Derived Class Types

An hourly employee is an employee
- In C++, an object of type HourlyEmployee can be used where an object of type Employee can be used
- **An object of a class type can be used wherever any of its ancestors can be used**
- An ancestor cannot be used wherever one of its descendents can be used
Derived Class Constructors

- A base class constructor is not inherited in a derived class
  - The base class constructor can be invoked by the constructor of the derived class
  - The constructor of a derived class begins by invoking the constructor of the base class in the initialization section:
    ```
    HourlyEmployee::HourlyEmployee : Employee( ),
                                 wage_rate( 0),
                                 hours( )
    { //no code needed }
    ```

Any Employee constructor could be invoked
Default Initialization

- If a derived class constructor does not invoke a base class constructor explicity, the base class default constructor will be used.

- If class B is derived from class A and class C is derived from class B
  - When a object of class C is created
    - The base class A's constructor is the first invoked
    - Class B's constructor is invoked next
    - C's constructor completes execution
Private is Private

A member variable (or function) that is private in the parent class is not accessible to the child class.

The parent class member functions must be used to access the private members of the parent.

This code would be illegal:
```cpp
void HourlyEmployee::print_check()
{
    net_pay = hours * wage_range;
}
```

- net_pay is a private member of Employee!
The protected Qualifier

Protected members of a class appear to be private outside the class, but are accessible by derived classes

- If member variables name, net_pay, and ssn are listed as protected (not private) in the Employee class, this code, illegal on the previous slide, becomes legal:

```cpp
HourlyEmployee::print_check()
{
    net_pay = hours * wage_range;
}```
Programming Style

- Using protected members of a class is a convenience to facilitate writing the code of derived classes.

  **Protected members are not necessary**
  - Derived classes can use the public methods of their ancestor classes to access private members

- Many programming authorities consider it bad style to use protected member variables
Redefinition of Member Functions

- When defining a derived class, only list the inherited functions that you wish to change for the derived class.
  - The function is declared in the class definition.
  - HourlyEmployee and SalariedEmployee each have their own definitions of print_check.
Using Derived Classes (part 1 of 2)

```cpp
#include <iostream>
#include "hourlyemployee.h"
#include "salariedemployee.h"
using std::cout;
using std::endl;
using namespace employeessavitch;

int main( )
{
    HourlyEmployee joe;
    joe.set_name("Mighty Joe");
    joe.set_ssn("123-45-6789");
    joe.set_rate(20.50);
    joe.set_hours(40);
    cout << "Check for " << joe.get_name( )
         << " for " << joe.get_hours( ) << " hours.\n"
         << "Employee Check: \n"
         << joe.print_check( );
    cout << endl;

    SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
    cout << "Check for " << boss.get_name( ) << endl;
    boss.print_check( );

    return 0;
}
```

*The functions set_name, set_ssn, set_rate, set_hours, and get_name are inherited unchanged from the class Employee.*

*The function print_check is redefined.*

*The function get_hours was added to the derived class HourlyEmployee.*
Sample Dialogue

Check for Mighty Joe for 40 hours.

_____________________________________________________
Pay to the order of Mighty Joe
The sum of 820 Dollars

_____________________________________________________
Check Stub: NOT NEGOTIABLE
Employee Number: 123-45-6789
Hourly Employee.
Hours worked: 40 Rate: 20.5 Pay: 820

_____________________________________________________

Check for Mr. Big Shot

_____________________________________________________
Pay to the order of Mr. Big Shot
The sum of 10500.5 Dollars

_____________________________________________________
Check Stub NOT NEGOTIABLE
Employee Number: 987-65-4321
Salaried Employee. Regular Pay: 10500.5
Redefining or Overloading

- A function redefined in a derived class has the same number and type of parameters
  - The derived class has only one function with the same name as the base class

- An overloaded function has a different number and/or type of parameters than the base class
  - The derived class has two functions with the same name as the base class
    - One is defined in the base class, one in the derived class
Function Signatures

- A function signature is the function's name with the sequence of types in the parameter list, not including any const or ' &'.

  - An overloaded function has multiple signatures.

- Some compilers allow overloading based on including const or not including const.
Access to a Redefined Base Function

When a base class function is redefined in a derived class, the base class function can still be used

- To specify that you want to use the base class version of the redefined function:

```cpp
HourlyEmployee sally_h;
sally_h.Employee::print_check();
```
15.2 Inheritance Details
Inheritance Details

Some special functions are, for all practical purposes, not inherited by a derived class.

- Some of the special functions that are not effectively inherited by a derived class include:
  - Destructors
  - Copy constructors
  - The assignment operator
Copy Constructors and Derived Classes

If a copy constructor is not defined in a derived class, C++ will generate a default copy constructor

- This copy constructor copies only the contents of member variables and will not work with pointers and dynamic variables
- The base class copy constructor will not be used
If a base class has a defined assignment operator = and the derived class does not:

- C++ will use a default operator that will have nothing to do with the base class assignment operator
Destructors and Derived Classes

- A destructor is **not inherited by a derived class**

- The derived class **should define its own destructor**
The Assignment Operator

In implementing an overloaded assignment operator in a derived class:

- It is normal to use the assignment operator from the base class in the definition of the derived class's assignment operator.
- Recall that the assignment operator is written as a member function of a class.
The Operator = Implementation

This code segment shows how to begin the implementation of the = operator for a derived class:

```cpp
Derived& Derived::operator= (const Derived& rhs) 
{
  Base::operator=(rhs)
  // This line handles the assignment of the inherited member variables by calling the base class assignment operator
  // The remaining code would assign the member variables introduced in the derived class
```
The Copy Constructor

Implementation of the derived class copy constructor is much like that of the assignment operator:

```cpp
Derived::Derived(const Derived& object) : Base(object), <other initializing>
{
...
}
```

- Invoking the base class copy constructor sets up the inherited member variables
- Since object is of type Derived it is also of type Base
Destructors in Derived Classes

- If the base class has a working destructor, defining the destructor for the defined class is relatively easy
  - When the destructor for a derived class is called, the destructor for the base class is automatically called
  - The derived class destructor need only use delete on dynamic variables added in the derived class, and data they may point to
Destruction Sequence

- If class B is derived from class A and class C is derived from class B…
  - When the destructor of an object of class C goes out of scope
    - The destructor of class C is called
    - Then the destructor of class B
    - Then the destructor of class A
  - Notice that destructors are called in the reverse order of constructor calls
15.3 Polymorphism
Polymorphism

- Polymorphism refers to the ability to associate multiple meanings with one function name using a mechanism called **late binding**

- Polymorphism is a key component of the philosophy of object oriented programming
Imagine a graphics program with several types of figures

- Each figure may be an object of a different class, such as a circle, oval, rectangle, etc.
- Each is a descendant of a class Figure
- Each has a function `draw()` implemented with code specific to each shape
- Class Figure has functions common to all figures
A Problem

- Suppose that class Figure has a function center
  - Function center moves a figure to the center of the screen by erasing the figure and redrawing it in the center of the screen
  - Function center is inherited by each of the derived classes
    - Function center uses each derived object's draw function to draw the figure
    - The Figure class does not know about its derived classes, so it cannot know how to draw each figure
Virtual Functions

Because the Figure class includes a method to draw figures, but the Figure class cannot know how to draw the figures, **virtual functions** are used.

Making a function virtual tells the compiler that you don't know how the function is implemented and to wait until the function is used in a program, then get the implementation from the object.

- This is called late binding.
Virtual Functions in C++

As another example, let's design a record-keeping program for an auto parts store.

- We want a versatile program, but we do not know all the possible types of sales we might have to account for.
  - Later we may add mail-order and discount sales.
  - Functions to compute bills will have to be added later when we know what type of sales to add.
  - To accommodate the future possibilities, we will make the bill function a virtual function.
The Sale Class

- All sales will be derived from the base class Sale
- The bill function of the Sale class is virtual
- The member function savings and operator < each use bill
- The Sale class interface and implementation are shown in
//This is the header file sale.h.
//This is the interface for the class Sale.
//Sale is a class for simple sales.
#ifndef SALE_H
#define SALE_H

#include <iostream>
using namespace std;

namespace salesavitch {

    class Sale {
    public:
        Sale();
        Sale(double the_price);
        virtual double bill() const;
        double savings(const Sale& other) const;
        //Returns the savings if you buy other instead of the calling object.
    protected:
        double price;
    };

    bool operator < (const Sale& first, const Sale& second);
    //Compares two sales to see which is larger.

} // salesavitch

#endif // SALE_H
// This is the implementation file: sale.cpp
// This is the implementation for the class Sale.
// The interface for the class Sale is in
// the header file sale.h.
#include "sale.h"

namespace salesavitch
{

    Sale::Sale() : price(0)
    {}

    Sale::Sale(double the_price) : price(the_price)
    {}

    double Sale::bill() const
    {
        return price;
    }

    double Sale::savings(const Sale& other) const
    {
        return (bill() - other.bill());
    }

    bool operator < (const Sale& first, const Sale& second)
    {
        return (first.bill() < second.bill());
    }
}

}//salesavitch
Virtual Function bill

Because function bill is virtual in class Sale, function savings and operator <, defined only in the base class, can in turn use a version of bill found in a derived class.

- When a DiscountSale object calls its savings function, defined only in the base class, function savings calls function bill.
- Because bill is a virtual function in class Sale, C++ uses the version of bill defined in the object that called savings.
The Derived Class DiscountSale

```cpp
// This is the interface for the class DiscountSale.
#ifndef DISCOUNTSALE_H
#define DISCOUNTSALE_H
#include "sale.h"

namespace salesavitch
{
    class DiscountSale : public Sale
    {
    public:
        DiscountSale();
        DiscountSale(double the_price, double the_discount);
        // Discount is expressed as a percent of the price.
        virtual double bill() const;

    protected:
        double discount;
    }
};
} // salesavitch
#endif // DISCOUNTSALE_H
```

// This is the implementation for the class DiscountSale.
#include "discountsale.h"

namespace salesavitch
{
    DiscountSale::DiscountSale() : Sale(), discount(0) {}

    DiscountSale::DiscountSale(double the_price, double the_discount) : Sale(the_price), discount(the_discount)
    {}

    double DiscountSale::bill() const
    {
        double fraction = discount/100;
        return (1 - fraction)*price;
    }
};
```
Use of a Virtual Function

//Demonstrates the performance of the virtual function bill.
#include <iostream>
#include "sale.h" // Not really needed, but safe due to ifndef.
#include "discountsale.h"
using namespace std;
using namespace salesavitch;

int main()
{
    Sale simple(10.00); // One item at $10.00.
    DiscountSale discount(11.00, 10); // One item at $11.00 with a 10% discount.

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    if (discount < simple)
    {
        cout << "Discounted item is cheaper.\n"
        cout << "Savings is $" << simple.savings(discount) << endl;
    }
    else
    {
        cout << "Discounted item is not cheaper.\n";

        return 0;
    }

Sample Dialogue

Discounted item is cheaper.
Savings is $0.10
DiscountSale::bill

- Class DiscountSale has its own version of virtual function bill
  - Even though class Sale is already compiled, Sale::savings( ) and Sale::operator< can still use function bill from the DiscountSale class
  - The keyword virtual tells C++ to wait until bill is used in a program to get the implementation of bill from the calling object
Virtual Details

To define a function differently in a derived class and to make it virtual

- Add keyword virtual to the function declaration in the base class
- virtual is not needed for the function declaration in the derived class, but is often included
- virtual is not added to the function definition
- Virtual functions require considerable overhead so excessive use reduces program efficiency
Overriding

- Virtual functions whose definitions are changed in a derived class are said to be overridden

- Non-virtual functions whose definitions are changed in a derived class are redefined
Type Checking

- C++ carefully checks for type mismatches in the use of values and variables.
- This is referred to as **strong type checking**.
  - Generally the type of a value assigned to a variable must match the type of the variable.
  - Recall that some automatic type casting occurs.
- **Strong type checking interferes with the concepts of inheritance.**
Consider

```cpp
class Pet {
    public:
        virtual void print();
        string name;
}
```

and

```cpp
class Dog : public Pet {
    public:
        virtual void print();
        string breed;
}
```
A Sliced Dog is a Pet

C++ allows the following assignments:

```cpp
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

However, `vpet` will lose the breed member of `vdog` since an object of class `Pet` has no breed member.

- This code would be illegal: `cout << vpet.breed;`

This is the slicing problem.
The Slicing Problem

It is legal to assign a derived class object into a base class variable

- This slices off data in the derived class that is not also part of the base class
- Member functions and member variables are lost
Extended Type Compatibility

- It is possible in C++ to avoid the slicing problem
  
  - Using pointers to dynamic variables we can assign objects of a derived class to variables of a base class without losing members of the derived class object.
Dynamic Variables and Derived Classes

Example:

```cpp
define Pet *ppet;
define Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;
void Dog::print( )
{
    cout << "name: "
    << name << endl;
    cout << "breed: "
    << breed << endl;
}
ppet->print( );  is legal and produces:  name: Tiny
    breed: Great Dane
//Program to illustrate use of a virtual function
//to defeat the slicing problem.

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

class Pet
{
 public:
   virtual void print();
   string name;
};

class Dog : public Pet
{
 public:
   virtual void print(); //keyword virtual not needed, but put
                       //here for clarity. (It is also good style!)
   string breed;
};

int main()
{
  Dog vdog;
  Pet vpet;

  vdog.name = "Tiny";
  vdog.breed = "Great Dane";
  vpet = vdog;

  //vpet.breed; is illegal since class Pet has no member named breed

  Dog *pdog;
  pdog = new Dog;
More Inheritance with Virtual Functions (part 2 of 2)

```cpp
pdog->name = "Tiny";
pdog->breed = "Great Dane";

Pet *ppet;
ppet = pdog;
ppet->print(); // These two print the same output:
pdog->print(); // name: Tiny breed: Great Dane

// The following, which accesses member variables directly
// rather than via virtual functions, would produce an error:
// cout << "name: " << ppet->name << " breed: "
//     << ppet->breed << endl;
// generates an error message: 'class Pet' has no member
// named 'breed'.
// See Pitfall section "Not Using Virtual Member Functions"
// for more discussion on this.

return 0;
}

void Dog::print()
{
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}

void Pet::print()
{
    cout << "name: " << endl; // Note no breed mentioned
}
```

Sample Dialogue

- name: Tiny
- breed: Great Dane
- name: Tiny
- breed: Great Dane
Use Virtual Functions

- The previous example:
  ```cpp
ppet->print();
```
  worked because print was declared as a virtual function

- This code would still produce an error:
  ```cpp
cout << "name: " << ppet->name
   << "breed: " << ppet->breed;
```
Why?

- 
  - ppet->breed is still illegal because ppet is a pointer to a Pet object that has no breed member
  
- Function print( ) was declared virtual by class Pet
  
  - When the computer sees ppet->print( ), it checks the virtual table for classes Pet and Dog and finds that ppet points to an object of type Dog
    
    - Because ppet points to a Dog object, code for Dog::print( ) is used
To help make sense of object oriented programming with dynamic variables, remember these rules

- If the domain type of the pointer p_ancestor is a base class for the domain type of pointer p_descendant, the following assignment of pointers is allowed:
  
  ```
  p_ancestor = p_descendant;
  ```

  and no data members will be lost

- Although all the fields of the p_descendant are there, virtual functions are required to access them
Virtual Compilation

When using virtual functions, you will have to define each virtual function before compiling

- Declaration is no longer sufficient
- Even if you do not call the virtual function you may see error message:
  "undefined reference to Class_Name virtual table"
Virtual Destructors

- Destructors should be made virtual
  - Consider `Base *pBase = new Derived;
    ...
    delete pBase;
  - If the destructor in Base is virtual, the destructor for Derived is invoked as pBase points to a Derived object, returning Derived members to the freestore
  - The Derived destructor in turn calls the Base destructor
Non-Virtual Destructors

- If the Base destructor is not virtual, only the Base destructor is invoked

- This leaves Derived members, not part of Base, in memory