Chapter 14

Inheritance

Learning Objectives

- Inheritance Basics
  - Derived classes, with constructors
  - protected: qualifier
  - Redefining member functions
  - Non-inherited functions
- Programming with Inheritance
  - Assignment operators and copy constructors
  - Destructors in derived classes
  - Multiple inheritance

Why Inheritance?

Let’s see the following code snippets!

```
1)   class HourlyEmployee
2)   {
3)       public:
4)           HourlyEmployee( );
5)           HourlyEmployee(const string&  theName, const string&  theSsn,
                                    double theWageRate, double theHours);
6)           string getName( ) const;    void setName(const string& newName);
7)           string getSsn( ) const;           void setSsn(const string& newSsn);
8)           double getNetPay( ) const;        void setNetPay(double newNetPay);
9)           double getRate( ) const;          void setRate(double newWageRate);
10)          double getHours( ) const;         void setHours(double hoursWorked);
11)          void printCheck( ) ;
12)      private:
13)        string name;
14)        string ssn;
15)        double wageRate;
16)        double hours;
17)        double netPay;
18)  };
```

```
1)    void HourlyEmployee::printCheck( )
2)      {
3)         setNetPay(hours * wageRate);

4)         cout << "\n_____\n";
5)         cout << "Pay to the order of " << getName( ) << endl;
6)         cout << "The sum of " << getNetPay( ) << " Dollars\n";
7)         cout << "_____\n";
8)         cout << "Check Stub: NOT NEGOTIABLE\n";
9)         cout << "Employee Number: " << getSsn( ) << endl;
10)        cout << "Hourly Employee. \nHours worked: " << hours
11)             << " Rate: " << wageRate << " Pay: " << getNetPay( ) << endl;
12)        cout << "_____\n";
13)     }
```

```
1)  class SalariedEmployee
2)  {
3)      public:
4)          SalariedEmployee( );
5)          SalariedEmployee (const string&  theName,
                              const string&  theSsn, double theWeeklySalary);
6)          string getName( ) const; void setName(const string& newName);
7)          string getSsn( ) const;        void setSsn(const string& newSsn);
8)          double getNetPay( ) const;  void setNetPay(double newNetPay);
9)          double getSalary( ) const;   void setSalary(double newSalary);
10)         void printCheck( );
11)     private:
12)         double salary;//weekly
13)         string name;
14)         string ssn;
15)        double netPay;
16) };
```

```
1)    void SalariedEmployee::printCheck( )
2)      {
3)          setNetPay(salary);
4)          cout << "\n_____\n";
5)          cout << "Pay to the order of " << getName( ) << endl;
6)          cout << "The sum of " << getNetPay( ) << " Dollars\n";
7)          cout << "_____\n";
8)          cout << "Check Stub NOT NEGOTIABLE \n";
9)          cout << "Employee Number: " << getSsn( ) << endl;
10)         cout << "Salaried Employee. Regular Pay: "    << salary << endl;
11)         cout << "_____\n";
12)     }
```

What are in common between HourlyEmployee and SalariedEmployee classes?

```
1)    class Employee
2)       {
3)       public:
4)          Employee( );
5)          Employee(const string& theName, const string& theSsn);

6)          string getName( ) const;
7)          string getSsn( ) const;
8)          double getNetPay( ) const;

9)          void setName(const string& newName);
10)         void setSsn(const string& newSsn);
11)         void setNetPay(double newNetPay);

12)         void printCheck( ) const;
13)      private:
14)         string name;
15)         string ssn;
16)         double netPay;
17)      };
```

```
1)    class HourlyEmployee : public Employee
2)      {
3)      public:
4)          HourlyEmployee( );
5)          HourlyEmployee(const string&  theName, const string&  theSsn,
6)                              double theWageRate, double theHours);
7)          void setRate(double newWageRate);
8)          double getRate( ) const;
9)          void setHours(double hoursWorked);
10)         double getHours( ) const;
11)         void printCheck( ) ;
12)     private:
13)         double wageRate;
14)         double hours;
15)     };
```

Think this class same as P4: class HourlyEmployee

```
1)    class SalariedEmployee : public Employee
2)      {
3)      public:
4)          SalariedEmployee( );
5)          SalariedEmployee (const string&  theName, const string&  theSsn,
6)                                  double theWeeklySalary);
7)          double getSalary( ) const;
8)          void setSalary(double newSalary);
9)          void printCheck( );
10)     private:
11)         double salary;//weekly
12)     };
```

Think this class same as P6: class SalariedEmployee

Usage Example

```
1)  #include <iostream>
2)  #include "hourlyemployee.h"
3)  #include "salariedemployee.h"
4)  using std::cout;          using std::endl;
5)  using SavitchEmployees::HourlyEmployee;
6)  using SavitchEmployees::SalariedEmployee;

7)  int main( )
8)  {
9)      HourlyEmployee joe;
10)     joe.setName("Mighty Joe");
11)     joe.setSsn("123-45-6789");
12)     joe.setRate(20.50);
13)     joe.setHours(40);
14)     cout << "Check for " << joe.getName( )
15)          << " for " << joe.getHours( ) << " hours.\n";
16)     joe.printCheck( );
17)     cout << endl;

18)     SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
19)     cout << "Check for " << boss.getName( ) << endl;
20)     boss.printCheck( );

21)     return 0;
22) }
```

Introduction to Inheritance

- Object-oriented programming
  - Powerful programming technique
  - Provides abstraction dimension called ***inheritance***
- General form of class is defined
  - Specialized versions then inherit properties of general class
  - And add to it/modify it's functionality for it's appropriate use

Inheritance Basics

- New class inherited from another class
- **Base class**
  - "General" class from which others derive
- **Derived class**
  - New class
  - Automatically has base class's:
    - Member variables
    - Member functions
  - Can then add additional member functions and variables

Derived Classes

- Consider example: Class of "Employees"
- Composed of:
  - Salaried employees
  - Hourly employees
- Each is "subset" of employees
  - Another might be those paid fixed wage each month or week

Derived Classes

- Don’t "need" type of generic "employee"
  - Since no one’s just an "employee"
- General concept of employee helpful!
  - All have names
  - All have social security numbers
  - Associated functions for these “basics” are same among all employees
- So “general” class can contain all these "things" about employees

Employee Class

- Many members of “employee” class apply to all types of employees
  - Accessor functions
  - Mutator functions
  - Most data items:
    - SSN
    - Name
    - Pay
- We won’t have “objects” of this class, however

Employee Class

- Consider printCheck() function:
  - Will always be "redefined" in derived classes
  - So different employee types can have different checks
  - Makes no sense really for “undifferentiated” employee
  - So function printCheck() in Employee class says just that
    - Error message stating “printCheck called for undifferentiated employee!! Aborting…"

Deriving from Employee Class

- Derived classes from Employee class:
  - Automatically have all member variables
  - Automatically have all member functions
- Derived class said to “inherit” members from base class
- Can then redefine existing members and/or add new members

HourlyEmployee Class Interface

- Note definition begins same as any other
  - #ifndef structure
  - Includes required libraries
  - Also includes employee.h!
- And, the heading:
  class HourlyEmployee : public Employee
  { …
  - Specifies “publicly inherited” from Employee class

HourlyEmployee Class Additions

- Derived class interface only lists ***new*** or "to be ***redefined***" members
  - Since all others inherited are already defined
  - i.e.: "all" employees have ssn, name, etc.
- HourlyEmployee adds:
  - Constructors
  - wageRate, hours member variables
  - setRate(), getRate(), setHours(), getHours() member functions

As shown in P10

HourlyEmployee Class Redefinitions

- HourlyEmployee **redefines**:
  - printCheck() member function
  - This “overrides” the printCheck() function implementation from Employee class
- It’s definition must be in HourlyEmployee class’s implementation
  - As do other member functions declared in HourlyEmployee’s interface
    - New and "to be redefined"

As shown in P10

Inheritance Terminology

- Common to simulate family relationships
- Parent class
  - Refers to base class
- **Child** class
  - Refers to **derived** class
- Ancestor class
  - Class that's a parent of a parent …
- Descendant class
  - Opposite of ancestor

Constructors in Derived Classes

- Base class **constructors** are **NOT** inherited in derived classes!
  - But they can be invoked within derived class constructor
    - Which is all we need!
- Base class constructor must initialize all base class member variables
  - Those inherited by derived class
  - So derived class constructor simply calls it
    - "First" thing derived class constructor does

Derived Class Constructor Example

- Consider syntax for HourlyEmployee constructor:

```
HourlyEmployee::HourlyEmployee(string theName, string theNumber,
                               double theWageRate, double theHours)
        : Employee(theName, theNumber),
                wageRate(theWageRate), hours(theHours)
{
        //Deliberately empty
}
```

- Portion after : is "initialization section"
  - Includes invocation of Employee constructor

```
1)   Employee::Employee(const string& theName,
                                        const string& theNumber)
2)       : name(theName), ssn(theNumber), netPay(0)
3)     {
4)       //deliberately empty
5)     }
```

Another HourlyEmployee Constructor

- A second constructor:

```
HourlyEmployee::HourlyEmployee()
                : Employee(),    wageRate(0), hours(0)
{
    //Deliberately empty
}
```

- Default version of base class constructor is called (no arguments)
- Should always **invoke** one of the base class's constructors

Constructor: No Base Class Call

- Derived class constructor should always invoke one of the base class's constructors
- If you do not:
  - **Default** base class constructor automatically called
- **Equivalent** constructor definition:
-

```
HourlyEmployee::HourlyEmployee(): wageRate(0), hours(0)
{ }
```

```
HourlyEmployee::HourlyEmployee(): Employee(), wageRate(0),
    hours(0)
{ }
```

Pitfall: Base Class Private Data

- Derived class "inherits" private member variables
  - But still cannot directly access them
  - Not even through derived class member functions!
- Private member variables can ONLY be accessed "by name" in member functions of the class they’re defined in

Pitfall: Base Class Private Member Functions

- Same holds for base class member functions
  - Cannot be accessed outside interface and implementation of base class
  - Not even in derived class member function definitions

Pitfall: Base Class Private Member Functions Impact

- Larger impact here vs. member variables
  - Member variables can be accessed indirectly via accessor or mutator member functions
  - Member functions simply not available
- This is "reasonable"
  - Private member functions should be simply "helper" functions
  - **Should be used only in class they're defined**

The protected: Qualifier

- New classification of class members
- Allows access "by name" in derived class
  - But nowhere else
  - Still no access "by name" in other classes
- In class it’s defined → acts like private
- Considered "protected" in derived class
  - To allow future derivations
- Many feel this "violates" information hiding

Redefinition of Member Functions

- Recall interface of derived class:
  - Contains declarations for new member functions
  - Also contains declarations for inherited member functions to be changed
  - Inherited member functions NOT declared:
    - Automatically inherited unchanged
- Implementation of derived class will:
  - **Define** new member functions
  - **Redefine** inherited functions as declared

Redefining vs. Overloading

- Very **different**!
- Redefining in derived class:
  - SAME parameter list
  - Essentially "re-writes" same function
- Overloading:
  - **Different** parameter list
  - Defined "**new**" function that takes different parameters
  - Overloaded functions must have different signatures

"Redefine" Example

SAME parameter list

```
1)    void HourlyEmployee::printCheck( )
2)      {
3)        setNetPay(hours * wageRate);

4)        cout << "\n_____\n";
5)        cout << "Pay to the order of " << getName( ) << endl;
6)        cout << "The sum of " << getNetPay( ) << " Dollars\n";
7)        cout << "_____\n";
8)        cout << "Check Stub: NOT NEGOTIABLE\n";
9)        cout << "Employee Number: " << getSsn( ) << endl;
10)       cout << "Hourly Employee. \nHours worked: " << hours
11)           << " Rate: " << wageRate << " Pay: " << getNetPay( ) << endl;
12)       cout << "_____\n";
13)     }

14)   void Employee::printCheck( ) const
15)     {
16)       cout << "\nERROR: printCheck FUNCTION CALLED FOR AN \n"
17)           << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
18)           << "Check with the author of the program about this bug.\n";
19)       exit(1);
20)     }
```

A Function's Signature

- Recall definition of a "signature":
  - Function's name
  - Sequence of types in parameter list
    - Including order, number, types
- Signature does NOT include:
  - Return type
  - const keyword
  - &

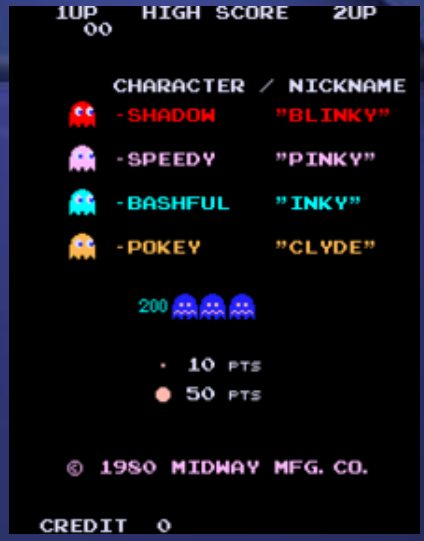
Accessing Redefined Base Function

- When redefined in derived class, base class's definition not "lost"
- Can specify it's use:
  Employee JaneE;
  HourlyEmployee SallyH;
  JaneE.printCheck(); →calls **Employee's** printCheck function
  SallyH.printCheck(); →calls HourlyEmployee printCheck function
  SallyH.Employee::printCheck(); → Calls **Employee's** printCheck function!
- Not typical here, but useful sometimes

One more Example

RPG with Pac-man characters

	Type	Strength	Hitpoints
Human	0	10	100
Blinky	1	8	85
Pinky	2	11	75
Inky	3	6	90
Clyde	4	7	80



The rules for calculating the damage are as follows:

- Every creature inflicts damage, a random number ***r***, 0 < ***r*** <= strength
- **Demons** have a 5% chance of inflicting a demonic attack, which is an additional 50 damage points. **Blinky** and **Pinky** are demons.
- **Inkys** have a 10% chance to inflict a magical attack that doubles the normal amount of damage.
- **Pinky** are very fast, so they get to attack twice.

Download & Run Creature-1

```
1)   class Creature
2)   {
3)   public:
4)        Creature(void);     // Initialize to human, 10 strength, 10 hit points
5)   //   ~Creature(void);
6)        Creature(int newType, int newStrength, int newHit);
7)        // Initialize creature to new type, strength, hit points

8)        // Also add appropriate accessor and mutator functions
9)        // for type, strength, and hit points
10)       int getDamage(); // Returns amount of damage this creature inflicts in one round of combat
11)       int getStrength();
12)       int getHitPoints();

13)  private:
14)       int type; // 0 human, 1 cyberdemon, 2 balrog, 3 elf
15)       int strength; // How much damage we can inflict
16)       int hitpoints; // How much damage we can sustain

17)       string getSpecies(); // Returns type of species

18)  };
```

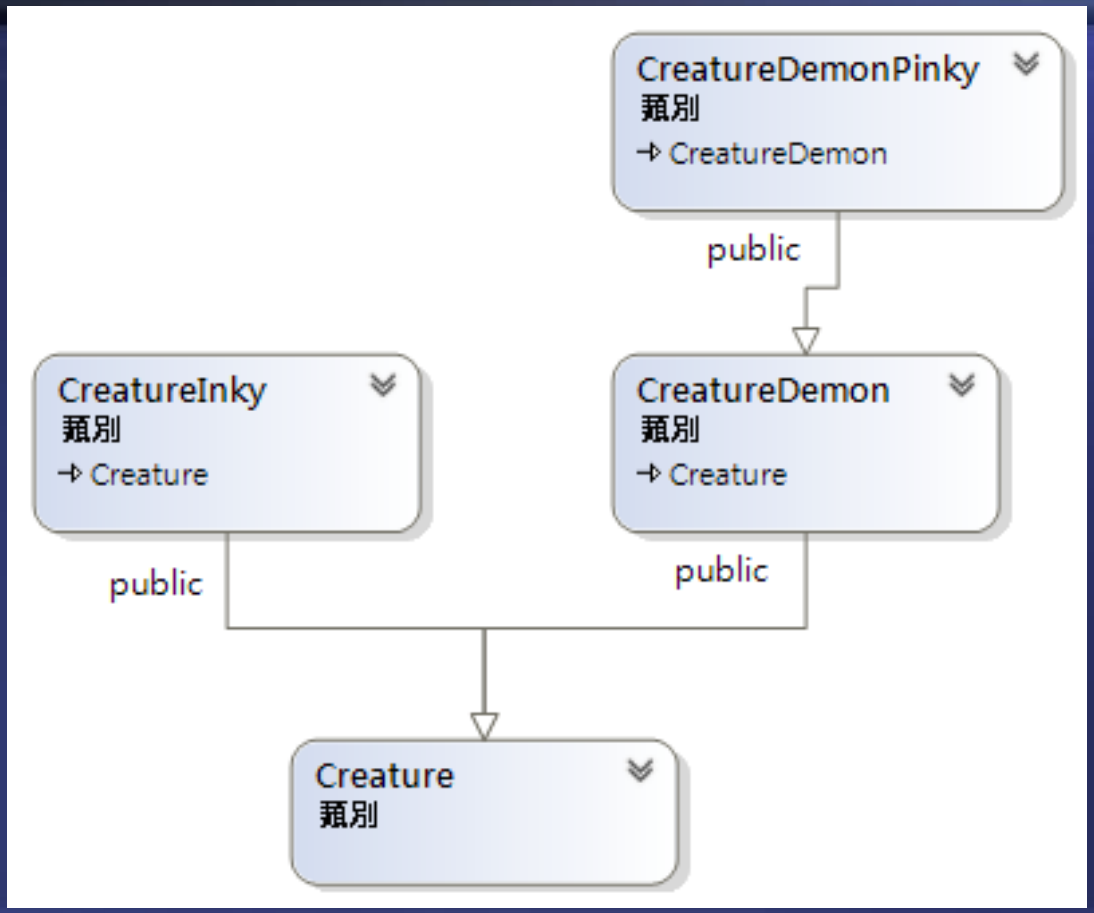
```
1)    Creature::Creature(void): type(0), strength(10), hitpoints(10) { }

2)    Creature::Creature(int newType, int newStrength, int newHit): type(newType),
      strength(newStrength), hitpoints(newHit) { }

3)    int Creature::getHitPoints(void) { return hitpoints; }

4)    int Creature::getStrength(void) { return strength; }

5)    string Creature::getSpecies()
6)    {
7)          switch (type)
8)          {
9)                case 0: return "Human";
10)               case 1: return "Blinky";
11)               case 2: return "Pinky";
12)               case 3: return "Inky";
13)               case 4: return "Clyde";
14)         }
15)         return "Unknown";
16)   }
```

```
#include <iostream>
#include "Creature.h"
using namespace std;
1)  int main(void)
2)  {
3)     Creature human(0, 10, 100);
4)      Creature Blinky(1, 8, 85);
5)      Creature Pinky(2, 11, 75);
6)      Creature Inky(3, 6, 90);
7)      Creature Clyde(4, 7, 80);

8)      cout<< "human: "<< "Strength= "<< human.getStrength()<< " Hipoints= "<< human.getHitPoints()<< endl;
9)      cout<< "damage: "<< human.getDamage()<< endl;

10)    cout<< “Blinky: ”<< “Strength= "<< Blinky.getStrength()<< " Hipoints= "<< Blinky.getHitPoints()<<  endl;
11)    cout<< "damage: "<< Blinky.getDamage()<< endl;

12)    cout<< "Pinky: "<< "Strength= "<< Pinky.getStrength()<< " Hipoints= "<< Pinky.getHitPoints()<<  endl;
13)    cout<< "damage: "<< Pinky.getDamage()<< endl;

14)    cout<< "Inky: "<< "Strength= "<< Inky.getStrength()<< " Hipoints= "<< Inky.getHitPoints()<< endl;
15)    cout<<"damage: "<< Inky.getDamage()<< endl;

16)    cout<< "Clyde: "<< "Strength= "<< Clyde.getStrength()<< " Hipoints= "<< Clyde.getHitPoints()<< endl;
17)    cout<<"damage: "<< Clyde.getDamage()<< endl;
18)    return true;

19)}
```

```
1)    int Creature::getDamage( )
2)    {
3)          int damage;
4)          // All creatures inflict damage, which is a random number up to their strength
5)          damage = (rand( ) % strength) + 1;
6)          cout<< getSpecies( ) << " attacks for " << damage << " points!" << endl;
7)          if ((type == 2) || (type == 1)) // Demons can inflict damage of 50 with a 5% chance  {
8)            if ((rand( ) % 100) < 5)            {
9)                damage = damage + 50;
10)               cout << "Demonic attack inflicts 50 "<< " additional damage points!" << endl;
11)           }
12)         }
13)         if (type == 3) // Inky inflict double magical damage with a 10% chance {
14)           if ((rand( ) % 10)==0)              {
15)               cout << "Magical attack inflicts " << damage << " additional damage points!" << endl;
16)               damage = damage * 2;
17)           }
18)         }
19)         if (type == 2) // Pinky are so fast they get to attack twice   {
20)              int damage2 = (rand() % strength) + 1;
21)             cout << "Pinky speed attack inflicts " << damage2 << " additional damage points!" << endl;
22)             damage = damage + damage2;
23)         }
24)         return damage;
25)   }
```

Download & Run Creature

Class Hierarchy



```
1)   class Creature
2)   {
3)   public:
4)         Creature(void);        // Initialize to human, 10 strength, 10 hit points
5)   //    ~Creature(void);
6)         Creature(int newStrength, int newHit);  // Initialize creature to new type, strength, hit points

7)         // Also add appropriate accessor and mutator functions
8)         // for type, strength, and hit points
9)
10)        // Returns amount of damage this creature inflicts in one round of combat
11)        int getDamage();
12)        int getStrength();
13)        int getHitPoints();

14)  private:
15)        int strength; // How much damage we can inflict
16)        int hitpoints; // How much damage we can sustain

17)  };
```

```
1)    Creature::Creature(void): strength(10), hitpoints(10) {  }

2)    Creature::Creature(int newStrength, int newHit)
                    : strength(newStrength), hitpoints(newHit) {  }

3)    int Creature::getHitPoints(void) {      return hitpoints; }

4)    int Creature::getStrength(void) {       return strength; }

5)    int Creature::getDamage( )
6)    {
7)          int damage;

8)          // All creatures inflict damage, which is a random number up to their strength
9)          damage = (rand( ) % strength) + 1;
10)         cout<< " attacks for " << damage << " points!" << endl;

11)         return damage;
12)   }
```

```
1)  #include "Creature.h"

2)  class CreatureInky: public Creature
3)  {
4)  public:
5)      CreatureInky(void);  // Initialize to human, 10 strength, 10 hit points
6)  //  ~Creature(void);
7)      CreatureInky(int newStrength, int newHit); // Initialize creature to new type, strength, hit
    points
8)      // Also add appropriate accessor and mutator functions
9)      // for type, strength, and hit points
10)     int getDamage(); // Returns amount of damage this creature inflicts in one round of combat
11)     //int getStrength();
12)     //int getHitPoints();

13) private:
14)     //int strength; // How much damage we can inflict
15)     //int hitpoints; // How much damage we can sustain

16) };
```

```
1)  CreatureInky::CreatureInky(): Creature() { }
2)  CreatureInky::CreatureInky(int newStrength, int newHit)
                    : Creature(newStrength, newHit) { }

3)  int CreatureInky::getDamage()
4)  {
5)      int damage= 0;

6)      damage=  Creature::getDamage();
7)      if ((rand( ) % 10)==0)
8)              {
9)                      cout << “Magical attack inflicts ”
                        << damage << " additional damage points!"
                        << endl;
10)                     damage = damage * 2;
11)             }
12)     return damage;
13) }
```

```
1)  class CreatureDemon: public Creature
2)  {
3)  public:
4)      CreatureDemon(void);    // Initialize to human, 10 strength, 10 hit points
5)      CreatureDemon(int newStrength, int newHit);
        // Initialize creature to new type, strength, hit points

6)      // Also add appropriate accessor and mutator functions
7)      // for type, strength, and hit points
8)      int getDamage();
        // Returns amount of damage this creature inflicts in one round of
    combat

9)  };
```

```
1)  CreatureDemon::CreatureDemon():Creature() { }
2)  CreatureDemon::CreatureDemon(int newStrength, int newHitpoints)
                    : Creature(newStrength, newHitpoints) { }
3)  int CreatureDemon::getDamage( )
4)  {
5)      int damage;
6)      damage= Creature::getDamage() ;
7)      // Demons can inflict damage of 50 with a 5% chance
8)              if ((rand( ) % 100) < 5)
9)              {
10)                     damage = damage + 50;
11)                     cout << “Demonic attack inflicts 50 “
                                << " additional damage points!" << endl;
12)             }
13)     return damage;
14) }
```

```
1)  CreatureDemonPinky::CreatureDemonPinky():CreatureDemon()
2)  { }

3)  CreatureDemonPinky::CreatureDemonPinky(int newStrength, int newHitpoints)
        : CreatureDemon(newStrength, newHitpoints)
4)  { }

5)  class CreatureDemonPinky: public CreatureDemon
6)  {
7)  public:
8)      CreatureDemonPinky(void);

9)      CreatureDemonPinky(int newStrength, int newHit);

10)     int getDamage();
        // Returns amount of damage this creature inflicts in one round of
    combat

11) };
```

```
1)  int CreatureDemonPinky::getDamage( )
2)  {
3)      int damage, damage2;


4)      damage= CreatureDemon::getDamage();
5)      damage2 = (rand() % getStrength()) + 1;
6)              cout << “Pinky speed attack inflicts ”
                        << damage2
                    << " additional damage points!" << endl;
7)      damage = damage + damage2;
8)
9)      return damage;
10)}
```

Functions Not Inherited

- All “normal” functions in base class are inherited in derived class
- **Exceptions**:
  - **Constructors** (we’ve seen)
  - **Destructors**
  - **Copy constructor**
    - But if not defined, generates "default" one
    - **Recall need to define one for pointers!**
  - Assignment operator
    - If not defined → default

Assignment Operators and Copy Constructors

- Recall: overloaded assignment operators and copy constructors
  NOT inherited
  - But can be used in derived class definitions
  - Typically MUST be used!
  - Similar to how derived class constructor invokes base class constructor

Assignment Operator Example

- Given "Derived" is derived from "Base":

```
Derived& Derived::operator =(const Derived & rightSide)
{
        Base::operator =(rightSide);
        …
}
```

- Notice code line
  - Calls assignment operator from **base class**
    - This takes care of all inherited member variables
  - Would then set new variables from derived class…

Copy Constructor Example

- Consider:
  Derived::Derived(const Derived& Object)
  : **Base**(Object), …
  {…}
- After : is invocation of base copy constructor
  - Sets inherited member variables of derived class object being created
  - Note Object is of type Derived; but it's also of type Base, so argument is valid

Destructors in Derived Classes

- If base class destructor functions correctly
  - Easy to write derived class destructor
- When derived class destructor is invoked:
  - Automatically calls base class destructor!
  - So no need for explicit call
- So derived class destructors need only be concerned with derived class variables
  - And any data they "point" to
  - Base class destructor handles inherited data automatically

Destructor Calling Order

- Consider:
  class B derives from class A
  class C derives from class B
  A ← B ← C
- When object of class C goes out of scope:
  - Class C destructor called 1st
  - Then class B destructor called
  - Finally class A destructor is called
- Opposite of how constructors are called

Examples for

Copy constructor
Assignment operator
destructor

pfarrayd.h

```
1)  class PFArrayD
2)  {
3)  public:
4)      PFArrayD( );    //Initializes with a capacity of 50.
5)      PFArrayD(int capacityValue);
6)      PFArrayD(const PFArrayD& pfaObject);

7)      void addElement(double element);
8)      bool full( ) const;    //Returns true if the array is full, false otherwise.

9)      int getCapacity( ) const;
10)     int getNumberUsed( ) const;
11)     void emptyArray( );    //Resets # to zero, effectively emptying the array.
12)     double& operator[](int index);    // access to elements

13)     PFArrayD& operator =(const PFArrayD& rightSide);

14)     ~PFArrayD( );
15) protected:
16)     double *a; //for an array of doubles.
17)     int capacity; //for the size of the array.
18)     int used; //for the number of array positions currently in use.
19) };
```

```
1)  PFArrayD::~PFArrayD( )
2)  {
3)      delete [] a;
4)  }
```

```
1)  PFArrayD::PFArrayD(const PFArrayD& pfaObject)
2)    :capacity(pfaObject.getCapacity( )),
          used(pfaObject.getNumberUsed( ))
3)  {
4)      a = new double[capacity];
5)      for (int i =0; i < used; i++)
6)          a[i] = pfaObject.a[i];
7)  }
```

```
1)  PFArrayD& PFArrayD::operator =(const PFArrayD& rightSide)
2)  {
3)      if (capacity != rightSide.capacity)
4)      {
5)          delete [] a;
6)          a = new double[rightSide.capacity];
7)      }
8)      capacity = rightSide.capacity;
9)      used = rightSide.used;
10)     for (int i = 0; i < used; i++)
11)         a[i] = rightSide.a[i];
12)     return *this;
13) }
```

```
1)  #include "pfarrayd.h"
2)  class PFArrayDBak : public PFArrayD
3)  {
4)  public:
5)      PFArrayDBak( );    //Initializes with a capacity of 50.
6)      PFArrayDBak(int capacityValue);
7)      PFArrayDBak(const PFArrayDBak& Object);

8)      void backup( );    //Makes a backup copy of the partially filled array.
9)      void restore( );   //Restores to the last saved version.

10)     PFArrayDBak& operator =(const PFArrayDBak& rightSide);

11)     ~PFArrayDBak( );
12) private:
13)     double *b; //for a backup of main array.
14)     int usedB; //backup for inherited member variable used.
15) };
```

```
PFArrayDBak::~PFArrayDBak( )
{
    delete [] b;
}
```

```
1)  PFArrayDBak::PFArrayDBak(const PFArrayDBak& oldObject)
2)              : PFArrayD(oldObject), usedB(0)
3)  {
4)      b = new double[capacity];
5)      usedB = oldObject.usedB;
6)      for (int i = 0; i < usedB; i++)
7)          b[i] = oldObject.b[i];
8)  }
```

```
class PFArrayDBak
{
    …
  private:
    double *b;
    int usedB;
};
```

```
1)  PFArrayDBak& PFArrayDBak::operator =
                                (const PFArrayDBak& rightSide)
2)  {
3)      PFArrayD::operator =(rightSide);
4)      if (capacity != rightSide.capacity)
5)      {
6)          delete [] b;
7)          b = new double[rightSide.capacity];
8)      }
9)      usedB = rightSide.usedB;
10)     for (int i = 0; i < usedB; i++)
11)         b[i] = rightSide.b[i];
12)     return *this;
13) }
```

"Is a" vs. "Has a" Relationships

- Inheritance
  - Considered an "Is a" class relationship
  - e.g., An HourlyEmployee "is a" Employee
  - A Convertible "is a" Automobile
- A class contains objects of another class as it's member data
  - Considered a "Has a" class relationship
  - e.g., One class "has a" object of another class as it's data

Protected and Private Inheritance

- New inheritance "forms"
  - Both are rarely used
- Protected inheritance:

  class SalariedEmployee : protected Employee
  {…}
  - Public members in base class become protected in derived class
- Private inheritance:

  class SalariedEmployee : private Employee
  {…}
  - All members in base class become private in derived class

Multiple Inheritance

- Derived class can have more than one base class!
  - Syntax just includes all base classes separated by commas:
    class derivedMulti : public base1, base2
    {…}
- Possibilities for ambiguity are endless!
- Dangerous undertaking!
  - Some believe should never be used
  - Certainly should only be used be experienced programmers!

Multiple Inheritance Example

```
1)  // multiple inheritance
2)  #include <iostream>
3)  using namespace std;

4)  class Polygon {
5)    protected:
6)      int width, height;
7)    public:
8)      Polygon (int a, int b) : width(a), height(b) {}
9)  };

10) class Output
11) {
12)   public:
13)     static void print (int i);
14) };

15) void Output::print (int i)
16) {
17)   cout << i << '\n';
18) }
```

```
1)   class Rectangle: public Polygon, public Output {
2)     public:
3)       Rectangle (int a, int b) : Polygon(a,b) {}
4)       int area ()
5)         { return width*height; }
6)   };

7)   class Triangle: public Polygon, public Output {
8)     public:
9)       Triangle (int a, int b) : Polygon(a,b) {}
10)      int area ()
11)        { return width*height/2; }
12)  };
13)
14)  int main () {
15)    Rectangle rect (4,5);
16)    Triangle trgl (4,5);
17)    rect.print (rect.area());
18)    Triangle::print (trgl.area());
19)    return 0;
20)  }
```

Summary of Public, Protected and Private Inheritance

Accessibility in Public Inheritance

Accessibility	private	protected	public
Accessible from own class?	yes	yes	yes
Accessible from dervied class?	no	yes	yes
Accessible outside dervied class?	no	no	yes

Accessibility in Protected Inheritance

Accessibility in Protected Inheritance

Accessibility	private	protected	public
Accessible from own class?	yes	yes	yes
Accessible from dervied class?	no	yes	yes
Accessible outside dervied class?	no	no	no

Accessibility in Private Inheritance

Accessibility	private	protected	public
Accessible from own class?	yes	yes	yes
Accessible from dervied class?	no	yes	yes
Accessible outside dervied class?	no	no	no

Summary 1

- Inheritance provides code reuse
  - Allows one class to "derive" from another, adding features
- Derived class objects inherit members of base class
  - And may add members
- Private member variables in base class cannot be accessed "by name" in derived
- Private member functions are not inherited

Summary 2

- Can redefine inherited member functions
  - To perform differently in derived class
- Protected members in base class:
  - Can be accessed "by name" in derived class member functions
- Overloaded assignment operator not inherited
  - But can be invoked from derived class
- Constructors are not inherited
  - Are invoked from derived class’s constructor

Example- Employee

```
1)  #ifndef EMPLOYEE_H
2)  #define EMPLOYEE_H

3)  #include <string>
4)  using std::string;

5)  namespace SavitchEmployees
6)  {

7)      class Employee
8)      {
9)      public:
10)         Employee( );
11)         Employee(const string& theName, const
        string& theSsn);
12)         string getName( ) const;
13)         string getSsn( ) const;
14)         double getNetPay( ) const;
15)         void setName(const string& newName);
16)         void setSsn(const string& newSsn);
17)         void setNetPay(double newNetPay);
18)         void printCheck( ) const;
19)     private:
20)         string name;
21)         string ssn;
22)         double netPay;
23)     };

24) }//SavitchEmployees

25) #endif //EMPLOYEE_H
```

```
1)  #ifndef HOURLYEMPLOYEE_H
2)  #define HOURLYEMPLOYEE_H

3)  #include <string>
4)  #include "employee.h"

5)  using std::string;

6)  namespace SavitchEmployees
7)  {

8)      class HourlyEmployee : public Employee
9)      {
10)     public:
11)         HourlyEmployee( );
12)         HourlyEmployee(const string&  theName,
        const string&  theSsn, double theWageRate,
        double theHours);
13)         void setRate(double newWageRate);
14)         double getRate( ) const;
15)         void setHours(double hoursWorked);
16)         double getHours( ) const;
17)         void printCheck( ) ;
18)     private:
19)         double wageRate;
20)         double hours;
21)     };

22) }//SavitchEmployees

23) #endif //HOURLYMPLOYEE_H
```

Employee Implementation

```
1)      Employee::Employee( ) : name("No name yet"), ssn("No number yet"), netPay(
2)        {       //deliberately empty    }

3)        Employee::Employee(const string& theName, const string& theNumber)
4)          : name(theName), ssn(theNumber), netPay(0)
5)        {       //deliberately empty    }

6)        string Employee::getName( ) const    {       return name;    }

7)        string Employee::getSsn( ) const    {       return ssn;    }

8)        double Employee::getNetPay( ) const    {       return netPay;    }

9)       void Employee::setName(const string& newName)    {       name = newName;    }

10)       void Employee::setSsn(const string& newSsn)    {       ssn = newSsn;    }

11)       void Employee::setNetPay (double newNetPay)    {       netPay = newNetPay;    }

12)       void Employee::printCheck( ) const
13)       {
14)          cout << "\nERROR: printCheck FUNCTION CALLED FOR AN \n"
15)               << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
16)               << "Check with the author of the program about this bug.\n";
17)          exit(1);
18)       }
```

```
1)   HourlyEmployee::HourlyEmployee( ) : Employee( ), wageRate(0), hours(0)   {   //deliberately empty   }

2)   HourlyEmployee::HourlyEmployee(const string&  theName, const string&  theNumber,  double theWageRate,
     double theHours)
3)        : Employee(theName, theNumber), wageRate(theWageRate), hours(theHours)
4)     {   //deliberately empty   }

5)   void HourlyEmployee::setRate(double newWageRate)   {   wageRate = newWageRate;   }

6)   double HourlyEmployee::getRate( ) const   {   return wageRate;   }

7)   void HourlyEmployee::setHours(double hoursWorked)   {   hours = hoursWorked;   }

8)   double HourlyEmployee::getHours( ) const   {   return hours;   }

9)     void HourlyEmployee::printCheck( )
10)    {
11)       setNetPay(hours * wageRate);

12)       cout << "\n_____\n";
13)       cout << "Pay to the order of " << getName( ) << endl;
14)       cout << "The sum of " << getNetPay( ) << " Dollars\n";
15)       cout << "_____\n";
16)       cout << "Check Stub: NOT NEGOTIABLE\n";
17)       cout << "Employee Number: " << getSsn( ) << endl;
18)       cout << "Hourly Employee. \nHours worked: " << hours
19)            << " Rate: " << wageRate << " Pay: " << getNetPay( ) << endl;
20)       cout << "_____\n";
21)    }
```

HourlyEmployee Implementation