Inheritance

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

Inheritance

- Inheritance is an “is-a” relationship
- For instance,“every employee is a person”
- Inheritance lets us create new classes from existing classes
- New classes are called the derived classes
- Existing classes are called the base classes
- Derived classes inherit the properties of the base classes

Inheritance (continued)

- Single inheritance: derived class has a single base class
- Multiple inheritance: derived class has more than one base class
- Can be viewed as a tree (hierarchy) where a base class is shown with its derived classes
- Public inheritance: all public members of base class are inherited as public members by derived class

- Inheritance can be viewed as a tree-like, or hierarchical, structure wherein a base class is shown with its derived classes.

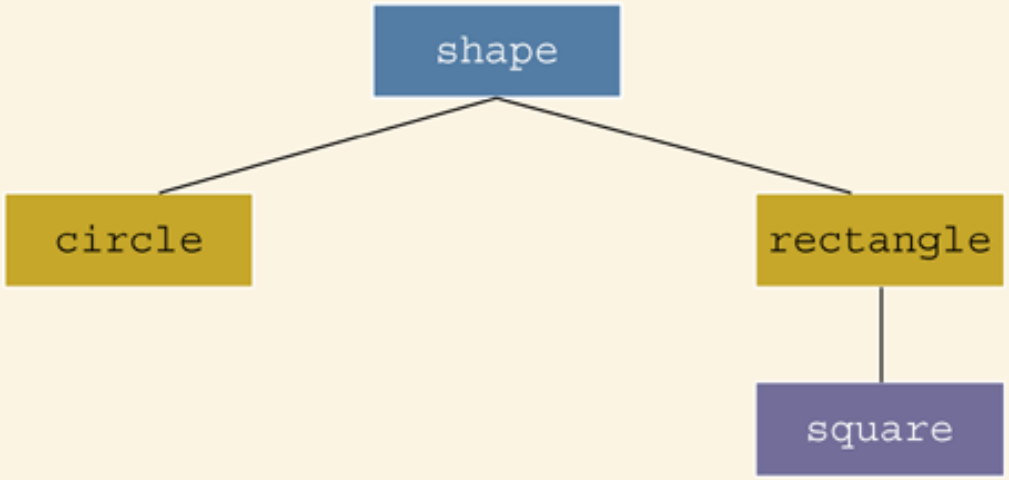

FIGURE 12-1 Inheritance hierarchy

The general syntax of a derived class is:

```
class className: memberAccessSpecifier baseClassName
{
    member list
};
```

where `memberAccessSpecifier` is `public`, `protected`, or `private`. When no `memberAccessSpecifier` is specified, it is assumed to be a `private` inheritance.

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 may not have "objects" of this class, however

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

Derived Classes

- The private members of a base class are private to the base class; hence, the members of the derived class cannot directly access them. In other words, when you write the definitions of the member functions of the derived class, you cannot directly access the private members of the base class.

- The public members of a base class can be inherited either as public members or as private members by the derived class. That is, the public members of the base class can become either public or private members of the derived class.

- The derived class can include additional members--data and/or functions.

Deriving from 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…"

Interface for the Derived Class HourlyEmployee

```
11      class HourlyEmployee : public Employee
12      {
13      public:
14          HourlyEmployee( );
15          HourlyEmployee(string theName, string theSsn,
16                                  double theWageRate, double theHours);
17          void setRate(double newWageRate);
18          double getRate( ) const;
19          void setHours(double hoursWorked);
20          double getHours( ) const;
21          void printCheck( );
22      private:
23          double wageRate;
24          double hours;
25      };

26  }//SavitchEmployees

27  #endif //HOURLYEMPLOYEE_H
```

You only list the declaration of an inherited member function if you want to change the definition of the function.

HourlyEmployee Class Interface

- Note 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

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"

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

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)
  { }
```

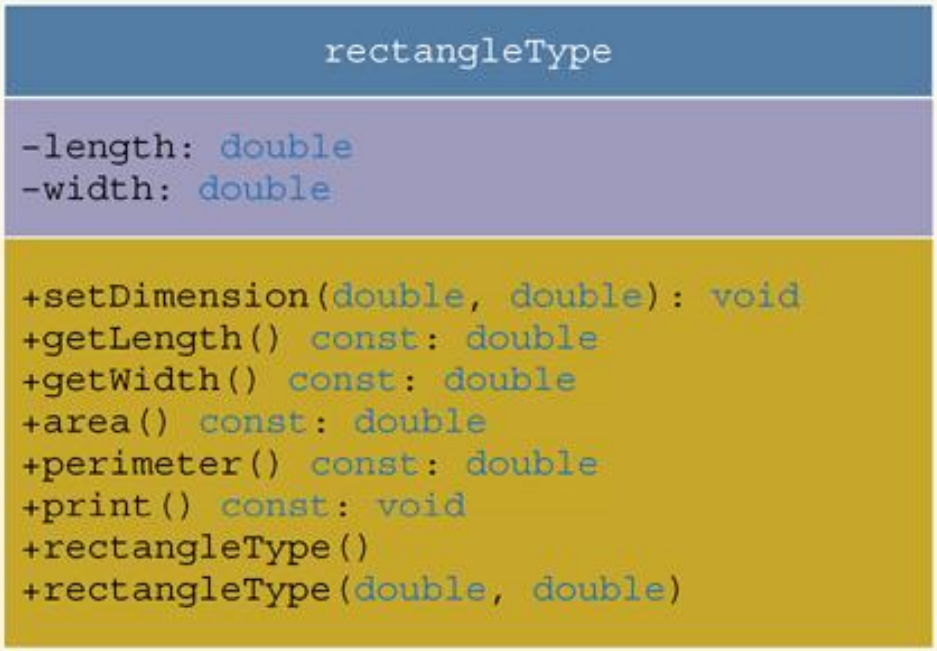

FIGURE 12-2 UML class diagram of the `class rectangleType`

```
class rectangleType
{
public:
    void setDimension(double l, double w);
      //Function to set the length and width of the rectangle.
      //Postcondition: length = l; width = w;

    double getLength() const;
      //Function to return the length of the rectangle.
      //Postcondition: The value of length is returned.

    double getWidth() const;
      //Function to return the width of the rectangle.
      //Postcondition: The value of width is returned.

    double area() const;
      //Function to return the area of the rectangle.
      //Postcondition: The area of the rectangle is
      //               calculated and returned.

    double perimeter() const;
      //Function to return the perimeter of the rectangle.
      //Postcondition: The perimeter of the rectangle is
      //               calculated and returned.
```

```
    void print() const;
      //Function to output the length and width of
      //the rectangle.

    rectangleType();
      //Default constructor
      //Postcondition: length = 0; width = 0;

    rectangleType(double l, double w);
      //Constructor with parameters
      //Postcondition: length = l; width = w;

private:
    double length;
    double width;
};
```

```
void rectangleType::setDimension(double l, double w)
{
    if (l >= 0)
        length = l;
    else
        length = 0;

    if (w >= 0)
        width = w;
    else
        width = 0;
}

double rectangleType::getLength() const
{
    return length;
}

double rectangleType::getWidth()const
{
    return width;
}

double rectangleType::area() const
{
    return length * width;
}
```

```
double rectangleType::perimeter() const
{
     return 2 * (length + width);
}

void rectangleType::print() const
{
    cout << "Length = "  << length
         << "; Width = " << width;
}

rectangleType::rectangleType(double l, double w)
{
    setDimension(l, w);
}

rectangleType::rectangleType()
{
    length = 0;
    width = 0;
}
```

```
class boxType: public rectangleType
{
public:
    void setDimension(double l, double w, double h);
      //Function to set the length, width, and height
      //of the box.
      //Postcondition: length = l; width = w; height = h;

    double getHeight() const;
      //Function to return the height of the box.
      //Postcondition: The value of height is returned.

    double area() const;
      //Function to return the surface area of the box.
      //Postcondition: The surface area of the box is
      //                calculated and returned.

    double volume() const;
      //Function to return the volume of the box.
      //Postcondition: The volume of the box is
      //               calculated and returned.

    void print() const;
      //Function to output the length and width of a rectangle.

    boxType();
      //Default constructor
      //Postcondition: length = 0; width = 0; height = 0;
```

```
    boxType(double l, double w, double h);
      //Constructor with parameters
      //Postcondition: length = l; width = w; height = h;

private:
    double height;
};
```

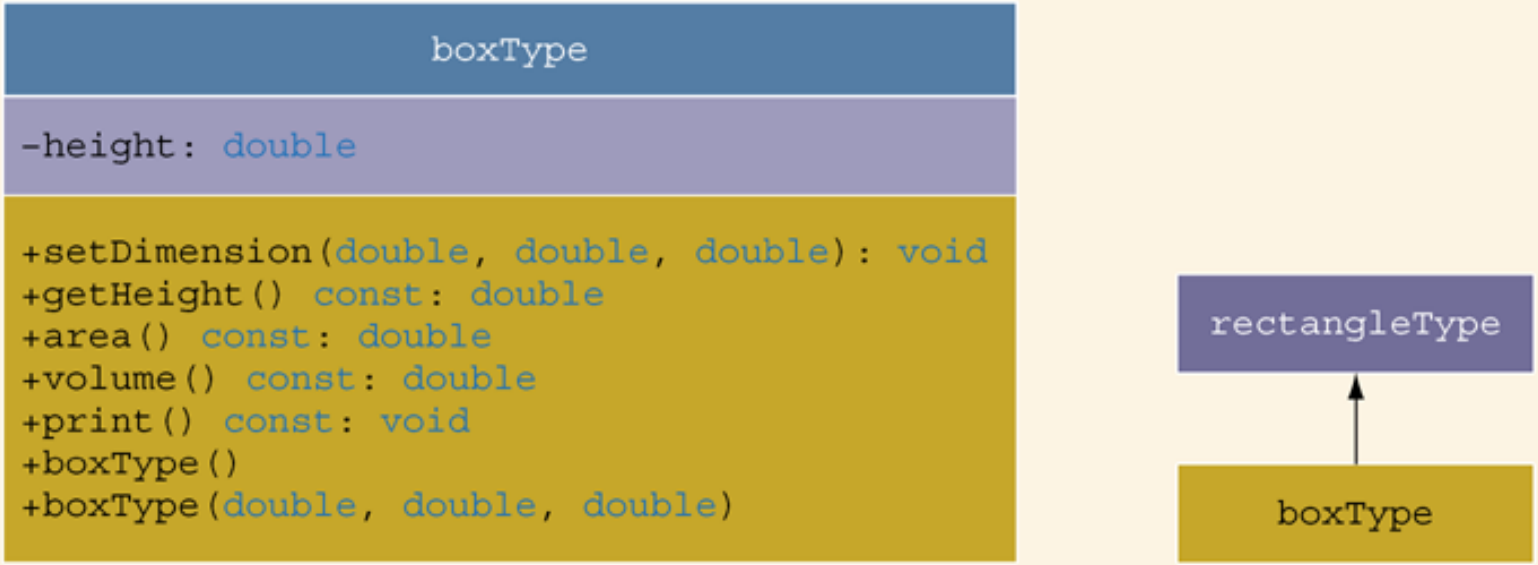

FIGURE 12-3 UML class diagram of the `class` `boxType` and the inheritance hierarchy

```
rectangleType myRectangle(5.0, 3.0);   //Line 1
boxType myBox(6.0, 5.0, 4.0);          //Line 2
```

FIGURE 12-4 Objects `myRectangle` and `myBox`

```
myRectangle.print();      //Line 3
cout << endl;             //Line 4
myBox.print();            //Line 5
cout << endl;             //Line 6
```

Output:

```
Length = 5.0; Width = 3.0
Length = 6.0; Width = 5.0; Height = 4.0
```

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 private 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
  - Private 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

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

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

"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., Personal_Info has date of birth (which is an object of DateType

Composition

- In composition, one or more member(s) of a class are objects of another class type
- Composition is a “has-a” relation
- Arguments to the constructor of a member-object are specified in the heading part of the definition of the constructor

Composition (example)

- Consider the `class personType` that stores a person's first name, last name and a personal ID (e.g., a Social Security number)

- We define another `class`, `dateType`, to store a person's date of birth.

- Then we construct the `class personalInfo` from the `class`es `personType` and `dateType`.

```
class dateType
{
public:
    void setDate(int month, int day, int year);
      //Function to set the date.
      //The member variables dMonth, dDay, and dYear are set
      //according to the parameters.
      //Postcondition: dMonth = month; dDay = day;
      //                dYear = year

    int getDay() const;
      //Function to return the day.
      //Postcondition: The value of dDay is returned.

    int getMonth() const;
      //Function to return the month.
      //Postcondition: The value of dMonth is returned.

    int getYear() const;
      //Function to return the year.
      //Postcondition: The value of dYear is returned.

    void printDate() const;
      //Function to output the date in the form mm-dd-yyyy.
```

```
    dateType(int month = 1, int day = 1, int year = 1900);
      //Constructor to set the date
      //The member variables dMonth, dDay, and dYear are set
      //according to the parameters.
      //Postcondition: dMonth = month; dDay = day; dYear = year;
      //               If no values are specified, the default
      //               values are used to initialize the member
      //               variables.

private:
    int dMonth; //variable to store the month
    int dDay;   //variable to store the day
    int dYear;  //variable to store the year
};
```

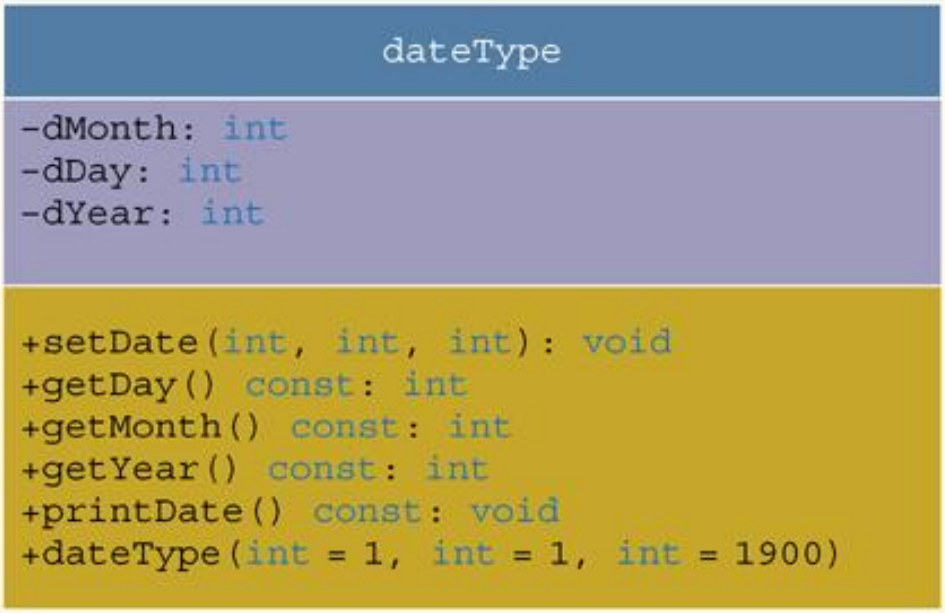

FIGURE 12-7 UML class diagram of the `class dateType`

```
void dateType::setDate(int month, int day, int year)
{
    dMonth = month;
    dDay = day;
    dYear = year;
}

int dateType::getDay() const
{
    return dDay;
}

int dateType::getMonth() const
{
    return dMonth;
}

int dateType::getYear() const
{
    return dYear;
}
```

```
void dateType::printDate() const
{
    cout << dMonth << "-" << dDay << "-" << dYear;
}

    //Constructor with parameters
dateType::dateType(int month, int day, int year)
{
    dMonth = month;
    dDay = day;
    dYear = year;
}
```

```
class personalInfo
{
public:
    void setpersonalInfo(string first, string last, int month,
                         int day, int year, int ID);
      //Function to set the personal information.
      //The member variables are set according to the
      //parameters.
      //Postcondition: firstName = first; lastName = last;
      //               dMonth = month; dDay = day;
      //               dYear = year; personID = ID;

    void printpersonalInfo () const;
      //Function to print the personal information.

    personalInfo(string first = "", string last = "",
                 int month = 1, int day = 1, int year = 1900,
                 int ID = 0);
      //Constructor
      //The member variables are set according to the
      //parameters.
      //Postcondition: firstName = first; lastName = last;
      //               dMonth = month; dDay = day;
      //               dYear = year; personID = ID;
      //               If no values are specified, the default
      //               values are used to initialize the member
      //               variables.
```

```
private:
    personType name;
    dateType bDay;
    int personID;
};
```

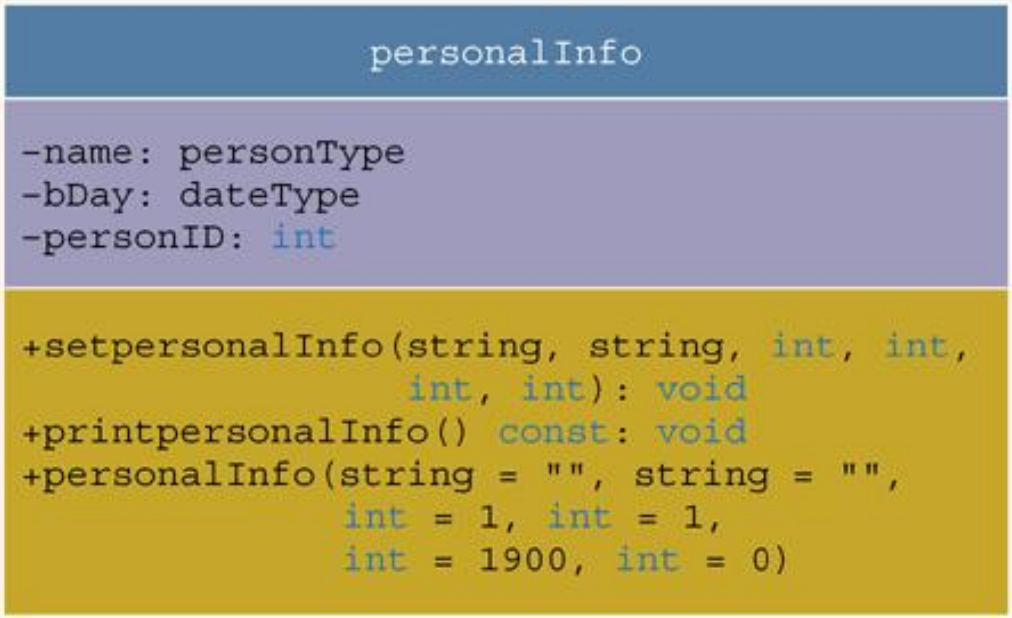

FIGURE 12-8 UML class diagram of the `class personalInfo`

```
void personalInfo::setpersonalInfo(string first, string last,
                                   int month, int day, int year, int ID)
{
    name.setName(first,last);
    bDay.setDate(month,day,year);
    personID = ID;
}

void personalInfo::printpersonalInfo() const
{
    name.print();
    cout << "'s date of birth is ";
    bDay.printDate();
    cout << endl;
    cout << "and personal ID is " << personID;
}

personalInfo::personalInfo(string first, string last, int month,
                           int day, int year, int ID)
        : name(first, last), bDay(month, day, year)
{
    personID = ID;

}
```

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!

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