



# Operator Overloading

---

## Exercise from last lesson

Implement a "Complex" class, which represents complex numbers. The data members are the real and imaginary part of the number.

There must be a constructor composed by 2 double.

Overload the operators `+`, `-`, `*`, `=`, `==`, `!=`, `<<`, `>>`

N. B .: the functions that overload the operators `<<` and `>>` must be declares as "friend" function

# friend Functions and friend Classes

- A **friend** function of one class is defined outside class's scope but has the right to access non-public members (private and protected)
- Friend functions are important for the performance of the code
- To declaring a **friend** function of one class, the keyword **friend** must precede function prototype
- To declare class **ClassTwo** as **friends** of class **ClassOne**, in the declaration one has to write:

```
friend class ClassTwo;
```

In the definition of **ClassOne**

- Properties of friendship:
- Friendship granted, not taken : In order to be that **class B** is friend of **Class A**, **class A** must explicitly declare that **class B** is friend
- Friendship is Not symmetric: if **class B** friend of **class A**, **class A** not necessarily friend of **class B**
- Friendship is Not transitive: if **class A** friend of **class B**, **class B** is friend of **class C**, **class A** not necessarily friend of **class C**

# friend Functions and friend Classes

```
1 // Fig. 7.11: fig07_11.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // Count class definition
9 class Count {
10     friend void setX( Count &, int ); // friend declaration
11
12 public:
13
14     // constructor
15     Count()
16         : x( 0 ) // initialize x to 0
17     {
18         // empty body
19
20     } // end Count constructor
21
```

Precede function prototype with keyword **friend**.

Friendfunction.cpp

# friend Functions and friend Classes

```
22     // output x
23     void print() const
24     {
25         cout << x << endl;
26
27     } // end function print
28
29 private:
30     int x; // data member
31
32 }; // end class Count
33
34 // function setX can modify private data of Count
35 // because setX is declared as a friend of Count
36 void setX( Count &c, int val )
37 {
38     c.x = val; // legal: setX is a friend of Count
39
40 } // end function setX
41
```

Pass **Count** object since C-style, standalone function

Since **setX** friend of **Count**, can access and modify **private** data member **x**.

# friend Functions and friend Classes

```
42  int main()
43  {
44      Count counter;          // create Count object
45
46      cout << "counter.x after instantiation: ";
47      counter.print();
48
49      setX( counter, 8 );    // set x with a friend
50
51      cout << "counter.x after call to setX friend function: ";
52      counter.print();
53
54      return 0;
55
56  } // end main
```

Use **friend** function to access and modify **private** data member **x**.

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

# friend Functions and friend Classes

```
1 // Fig. 7.12: fig07_12.cpp
2 // Non-friend/non-member functions cannot access
3 // private data of a class.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // Count class definition
10 // (note that there is no friendship declaration)
11 class Count {
12
13 public:
14
15     // constructor
16     Count()
17         : x( 0 ) // initialize x to 0
18     {
19         // empty body
20
21     } // end Count constructor
22
```

# friend Functions and friend Classes

```
23     // output x
24     void print() const
25     {
26         cout << x << endl;
27
28     } // end function print
29
30 private:
31     int x; // data member
32
33 }; // end class Count
34
35 // function tries to modify private data of Count,
36 // but cannot because function is
37 void cannotSetX( Count &c, int val
38 {
39     c.x = val; // ERROR: cannot access private member in Count
40
41 } // end function cannotSetX
42
43 int main()
44 {
45     Count counter; // create Count object
46
47     cannotSetX( counter, 3 ); // cannotSetX is not a friend
48
49     return 0;
50
51 } // end main
```

Attempting to modify **private** data member from non-**friend** function results in error.



# friend Functions and friend Classes

```
23     // output x
24     void print() const
25     {
26         cout << x << endl;
27
28     } // end function print
29
30 private:
31     int x; // data member
32
33 }; // end class Count
34
35 // function tries to modify private data of Count,
36 // but cannot because function is not a friend of Count
37 void cannotSetX( Count &c, int val )
38 {
39     c.x = val; // ERROR: cannot access private member in Count
40
41 } // end function cannotSetX
42
43
```

```
Notfriendfunctions.cpp: In function 'void cannotSetX(Count&, int)':
Notfriendfunctions.cpp:39:5: error: 'int Count::x' is private within this context
    c.x = val; // ERROR: cannot access private member in Count
    ^
Notfriendfunctions.cpp:31:7: note: declared private here
    int x; // data member
```

Attempting to modify **private** data member from non-**friend** function results in error.

# Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
  - Member functions
    - Use **this** keyword to implicitly get argument
    - Gets left operand for binary operators (like **+**)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **friend** to access **private** or **protected** data
  - Called when
    - Left operand of binary operator of same class
    - Single operand of unitary operator of same class

# Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
  - Member functions
    - Use **this** keyword to implicitly get argument
    - Gets left operand for binary operators (like **+**)
    - Leftmost object must be of same class as operator
  - Non member functions:
    - If the leftmost object have be of a different type wrt to the left type, the function **must** be defined a d non-member function
    - Can have object of different class than operator
    - Need parameters for both operands
    - Must be a **friend** to access **private** or **protected** data
  - Called when
    - Left operand of binary operator of same class
    - Single operand of unitary operator of same class

# Operator Functions As Class Members Vs. As Friend Functions

- Overloaded `<<` operator
  - Left operand of type **ostream** &
    - Such as **cout** object in **cout << classObject** → it has to be defined as a non-member function
- Similarly, overloaded `>>` needs **istream** &
  - **cin >> classObject**
- Thus, both must be non-member functions
- To access to private data member, the `>>` and `<<` operator overloading have to be declared as **friend** function

# Solution - 1

```
// Definizione Classe Complessi

#ifndef COMPLESSI_H
#define COMPLESSI_H
#include <iostream>

using std::ostream;
using std::istream;

class Complessi {
    friend ostream &operator<<( ostream &, const Complessi & );
    friend istream &operator>>( istream &, Complessi & );

public:
    Complessi( double = 0.0, double = 0.0 );    // constructor

    Complessi operator+( const Complessi& ) const; // addition
    Complessi operator-( const Complessi& ) const; // subtraction
    Complessi operator*( const Complessi& ) const; // multiplication
    Complessi& operator=( const Complessi& );    // assignment
    bool operator==( const Complessi& ) const;
    bool operator!=( const Complessi& ) const;

private:
    double reale;    // parte reale
    double immaginaria; // parte immaginaria
}; // end classe Complessi

#endif // COMPLESSI_H
```

Overloaded operators

# Solution - 2

```
#include "Complessi.h"
#include <iostream>

using std::ostream;
using std::istream;

// Constructor
Complessi::Complessi( double r, double i )
{
    reale = r;
    immaginaria = i;
} // end constructor
```

# Solution- 3

```
// Overloaded addition operator
Complessi Complessi::operator+( const Complessi &operand2 ) const
{
    Complessi sum;

    sum.reale = reale + operand2.reale;
    sum.immaginaria = immaginaria + operand2.immaginaria;
    return sum;

} // end function operator+
```

```
// Overloaded subtraction operator
Complessi Complessi::operator-( const Complessi &operand2 ) const
{
    Complessi diff;

    diff.reale = reale - operand2.reale;
    diff.immaginaria = immaginaria - operand2.immaginaria;
    return diff;

} // end function operator-
```

```
// Overloaded multiplication operator
Complessi Complessi::operator*( const Complessi &operand2 ) const
{
    Complessi times;

    times.reale = reale * operand2.reale + immaginaria *
                operand2.immaginaria;
    times.immaginaria = reale * operand2.immaginaria +
                immaginaria * operand2.reale;

    return times;

} // end function operator*
```

# Solution- 4

```
// Overloaded = operator
Complessi& Complessi::operator=( const Complessi &right )
{
    reale = right.reale;
    immaginaria = right.immaginaria;
    return *this;    // enables concatenation
} // end function operator=

// Overloaded == operator
bool Complessi::operator==( const Complessi &right ) const
{
    return right.reale == reale && right.immaginaria ==
           immaginaria ? true : false;
} // end function operator==

// Overloaded != operator
bool Complessi::operator!=( const Complessi &right ) const
{
    return !( *this == right );
} // end function operator!=
```



# Solution- 5

```
// Overloaded << operator
ostream& operator<<( ostream &output, const Complessi &complessi )
{
    output << complessi.reale << " + " << complessi.immaginaria << 'i';
    return output;
} // end function operator<<

// Overloaded >> operator
istream& operator>>( istream &input, Complessi &complessi )
{
    input >> complessi.reale;
    input.ignore( 3 );          // skip spaces and +
    input >> complessi.immaginaria;
    input.ignore( 2 );

    return input;
} // end function operator>>
```



# Composition and Inheritance

---

# Objectives

- Often the usage of existing classes is used to define new classes.
- This can be done using two methods:
  - Composition
  - Inheritance

# Composition

- Composition (also called *containment* or *aggregation*) of classes refers to the use of one or more classes within the definition of another class.
- When a data member of the new class is an object of another class, we say that the new class is a **composite** of the other objects.

# Composition: Objects as Members of Classes

```
1 // Fig. 7.6: date1.h
2 // Date class definition.
3 // Member functions defined in date1.cpp
4 #ifndef DATE1_H
5 #define DATE1_H
6
7 class Date {
8
9 public:
10     Date( int = 1, int = 1, int = 1900 ); // default constructor
11     void print() const; // print date in month/day/year format
12     ~Date(); // provided to confirm destruction order
13
14 private:
15     int month; // 1-12 (January-December)
16     int day; // 1-31 based on month
17     int year; // any year
18
19     // utility function to test proper day for month and year
20     int checkDay( int ) const;
21
22 }; // end class Date
23
24 #endif
```

Note no constructor with parameter of type **Date**.  
Recall compiler provides default copy constructor.

date1.h

# Composition: Objects as Members of Classes

```
1 // Fig. 7.7: date1.cpp
2 // Member-function definitions for class Date.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // include Date class definition from date1.h
9 #include "date1.h"
10
11 // constructor confirms proper value for month; calls
12 // utility function checkDay to confirm proper value for day
13 Date::Date( int mn, int dy, int yr )
14 {
15     if ( mn > 0 && mn <= 12 ) // validate the month
16         month = mn;
17
18     else { // invalid month set to 1
19         month = 1;
20         cout << "Month " << mn << " invalid. Set to month 1.\n";
21     }
22
23     year = yr; // should validate yr
24     day = checkDay( dy ); // validate the day
25
```

date1.cpp

# Composition: Objects as Members of Classes

```
26     // output Date object to show when its constructor is called
27     cout << "Date object constructor for date ";
28     print();
29     cout << endl;
30
31 } // end Date constructor
32
33 // print Date object in form month/day/year
34 void Date::print() const
35 {
36     cout << month << '/' << day << '/' << year;
37
38 } // end function print
39
40 // output Date object to show when its destructor is called
41 Date::~Date()
42 {
43     cout << "Date object destructor for date ";
44     print();
45     cout << endl;
46
47 } // end destructor ~Date
48
```

Output to show timing of constructors

No arguments; each member function contains implicit handle to object on which it operates.

Output to show timing of destructors.

date1.cpp

# Composition: Objects as Members of Classes

```
49 // utility function to confirm proper day value based on
50 // month and year; handles leap years, too
51 int Date::checkDay( int testDay ) const
52 {
53     static const int daysPerMonth[ 13 ] =
54         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
55
56     // determine whether testDay is valid for specified month
57     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
58         return testDay;
59
60     // February 29 check for leap year
61     if ( month == 2 && testDay == 29 &&
62         ( year % 400 == 0 ||
63           ( year % 4 == 0 && year % 100 != 0 ) ) )
64         return testDay;
65
66     cout << "Day " << testDay << " invalid. Set to day 1.\n";
67
68     return 1; // leave object in consistent state if bad value
69
70 } // end function checkDay
```

date1.cpp



# Composition: Objects as Members of Classes

```
1 // Fig. 7.8: employee1.h
2 // Employee class definition.
3 // Member functions defined in employee1.cpp.
4 #ifndef EMPLOYEE1_H
5 #define EMPLOYEE1_H
6
7 // include Date class definition from date1.h
8 #include "date1.h"
9
10 class Employee {
11
12 public:
13     Employee(
14         const char *, const char *, const Date &, const Date & );
15
16     void print() const;
17     ~Employee(); // provided to confirm destruction order
18
19 private:
20     char firstName[ 25 ];
21     char lastName[ 25 ];
22     const Date birthDate; // composition: member object
23     const Date hireDate; // composition: member object
24
25 }; // end class Employee
26
27 #endif
```

Using composition; **Employee** object contains **Date** objects as data members

employee1.h

```

1 // Fig. 7.9: employee1.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring> // strcpy and strlen prototypes
9
10 #include "employee1.h" // Employee class definition
11 #include "date1.h" // Date class definition
12
13 // constructor uses member initializer list to pass initializer
14 // values to constructors of member objects birthDate and
15 // hireDate [Note: This invokes the so-called "default copy
16 // constructor" which the C++ compiler provides implicitly.]
17 Employee::Employee( const char *first, const char *last,
18 const Date &dateOfBirth, const Date &dateOfHire )
19 : birthDate( dateOfBirth ) // initialize birthDate
20 hireDate( dateOfHire ) // initialize hireDate
21 {
22 // copy first into firstName and be sure to
23 int length = strlen( first );
24 length = ( length < 25 ? length : 24 );
25 strncpy( firstName, first, length );
26 firstName[ length ] = '\0';
27
28 // copy last into lastName and be sure that it fits
29 length = strlen( last );
30 length = ( length < 25 ? length : 24 );
31 strncpy( lastName, last, length );
32 lastName[ length ] = '\0';
33
34 // output Employee object to show when constructor is called
35 cout << "Employee object constructor: "
36 << firstName << ' ' << lastName << endl;

```

Member initializer syntax to initialize **Date** data members **birthDate** and **hireDate**; compiler uses default copy constructor.

Output to show timing of constructors.

employee1.cpp

# Composition: Objects as Members of Classes

```
38     } // end Employee constructor
39
40     // print Employee object
41     void Employee::print() const
42     {
43         cout << lastName << ", " << firstName << "\nHired: ";
44         hireDate.print();
45         cout << " Birth date: ";
46         birthDate.print();
47         cout << endl;
48
49     } // end function print
50
51     // output Employee object to show when
52     Employee::~Employee()
53     {
54         cout << "Employee object destructor: "
55             << lastName << ", " << firstName << endl;
56
57     } // end destructor ~Employee
```

Output to show timing of destructors.



employee1.cpp

# Composition: Objects as Members of Classes

```
1 // Fig. 7.10: fig07_10.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "employee1.h" // Employee class definition
9
10 int main()
11 {
12     Date birth( 7, 24, 1949 );
13     Date hire( 3, 12, 1988 );
14     Employee manager( "Bob", "Jones", birth, hire );
15
16     cout << '\n';
17     manager.print();
18
19     cout << "\nTest Date constructor with invalid values:\n";
20     Date lastDayOff( 14, 35, 1994 ); // invalid month and day
21     cout << endl;
22
23     return 0;
24
25 } // end main
```

Create **Date** objects to pass to **Employee** constructor.

useemployee1.cpp

# Composition: Objects as Members of Classes

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones
```

```
Jones, Bob
Hired: 3/12/1988 Birth date: 7/24/1949
```

```
Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994
```

```
Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

Note two additional **Date** objects constructed; no output since default copy constructor used.

Destructor for host object **manager** runs before destructors for member objects **hireDate** and **birthDate**.

Destructor for **Employee**'s member object **hireDate**.

Destructor for **Employee**'s member object **birthDate**.

Destructor for **Date** object **hire**.

Destructor for **Date** object **birth**.

useemployee1.cpp

# Example: A Person Class

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

class Person
{public:
    Person(string n="", string nat="U.S.A.", int s=1)
        : name(n), nationality(nat), sex(s) {}
    void printName ()    { cout << name; }
    void printNationality () { cout << nationality; }
private:
    string name, nationality;
    int sex;
};

int main()
{
    Person creator("Bjarne Stroustrup", "Denmark");
    cout << "The creator of C++ was ";
    creator.printName();
    cout << " who was born in ";
    creator.printNationality();
    cout << ".\n";
    return 0;
}
```

Person.h  
UsePerson.cpp

The creator of C++ was Bjarne Stroustrup who was born in Denmark

# Example: A Person Class

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

class Person
{public:
    Person(string n="", string nat="U.S.A.", int s=1)
        : name(n), nationality(nat), sex(s) {}
    void printName ()    { cout << name; }
    void printNationality () { cout << nationality; }
private:
    string name, nationality;
    int sex;
};

int main()
{
    Person creator("Bjarne Stroustrup", "Denmark");
    cout << "The creator of C++ was ";
    creator.printName();
    cout << " who was born in ";
    creator.printNationality();
    cout << ".\n";
    return 0;
}
```

Composition of the **string** class  
with the **Person** class

The creator of C++ was Bjarne Stroustrup who was born in Denmark

# Example a Date Class

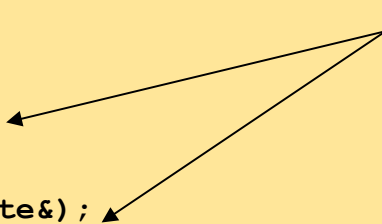
```
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::istream;
using std::ostream;
using std::string;

class Date {
    friend istream& operator>>(istream&, Date&);
    friend ostream& operator<<(ostream&, const Date&);
public:
    Date(int m=0, int d=0, int y=0) : month(m), day(d), year(y) { }
    void setDate(int m, int d, int y) { month = m; day = d; year = y;}
private:
    int month, day, year;
};

istream& operator>>(istream& in, Date& x)
{in >> x.month >> x.day >> x.year;
 return in;
}

ostream& operator<<(ostream& out, const Date& x)
{static string monthName[13] = {"", "January", "February", "March",
    "April", "May", "June", "July", "August",
    "September", "October", "November", "December"};
 out << monthName[x.month] << " " << x.day << ", " << x.year;
 return out;
}
```

Overload of the << and >> operators



Date.h, cpp



# Example a Date Class

```
int main()
{Date peace(11,11,1918);
  cout << "World War I ended on " << peace << ".\n";
  peace.setDate(8,14,1945);
  cout << "World War II ended on " << peace << ".\n";
  cout << "Enter month, day, and year: ";
  Date date;
  cin >> date;
  cout << "The date is "<< date << ".\n";
}
```

```
World War I ended on November 11, 1918.
World War II ended on August 14, 1945.
Enter month, day, and year: 10 10 2010
The date is October 10, 2010.
```

useDate.cpp

# Composition of Date class with Person Class

```
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

#include "Date.h"

class Person1
{public:
    Person1(string n="", string nat="U.S.A.", int s=1)
        : name(n), nationality(nat), sex(s) {}
    void setDOB(int m, int d, int y){dob.setDate(m,d,y);}
    void setDOD(int m, int d, int y){dod.setDate(m,d,y);}
    void printName ()    { cout << name; }
    void printNationality () { cout << nationality; }
    void printDOB(){cout<<dob;}
    void printDOD(){cout<<dod;}

private:
    string name, nationality;
    Date dob, dod;
    int sex;
};
```

Note that a member function of one class is used to define member functions of the composed class

Person1 . { cpp , h }

# Composition of Date class with Person Class

```
int main()
{
    Person1 author("Thomas Jefferson", "USA", 1);
    author.setDOB(4,13,1743);
    author.setDOD(7,4,1826);
    cout << "The author of the Declaration of Independence is ";
    author.printName();
    cout << ".\n He was born in ";
    author.printNationality();
    cout << " on ";
    author.printDOB();
    cout<<" and died on ";
    author.printDOD();
    cout<<".\n"<<endl;
    return 0;
}
```

The author of the Declaration of Independence is Thomas Jefferson.  
He was born in USA on April 13, 1743 and died on July 4, 1826.

usePerson1.cpp

# Composition

- Composition is often referred to as a “**has-a**” relationship because the objects of the composite class “have” objects of the composed class as members.
- Each object of the Person class “has a” **name** and a **nationality** which are **string** objects.

# Inheritance

- Another way to reuse existing software to create a new one is by means of inheritance (also called *specialization* or *derivation*)
- This is often referred as an “**is-a**” relationship because every object of the class being defined “**is**” also an object of the inherited class.
-

# Inheritance

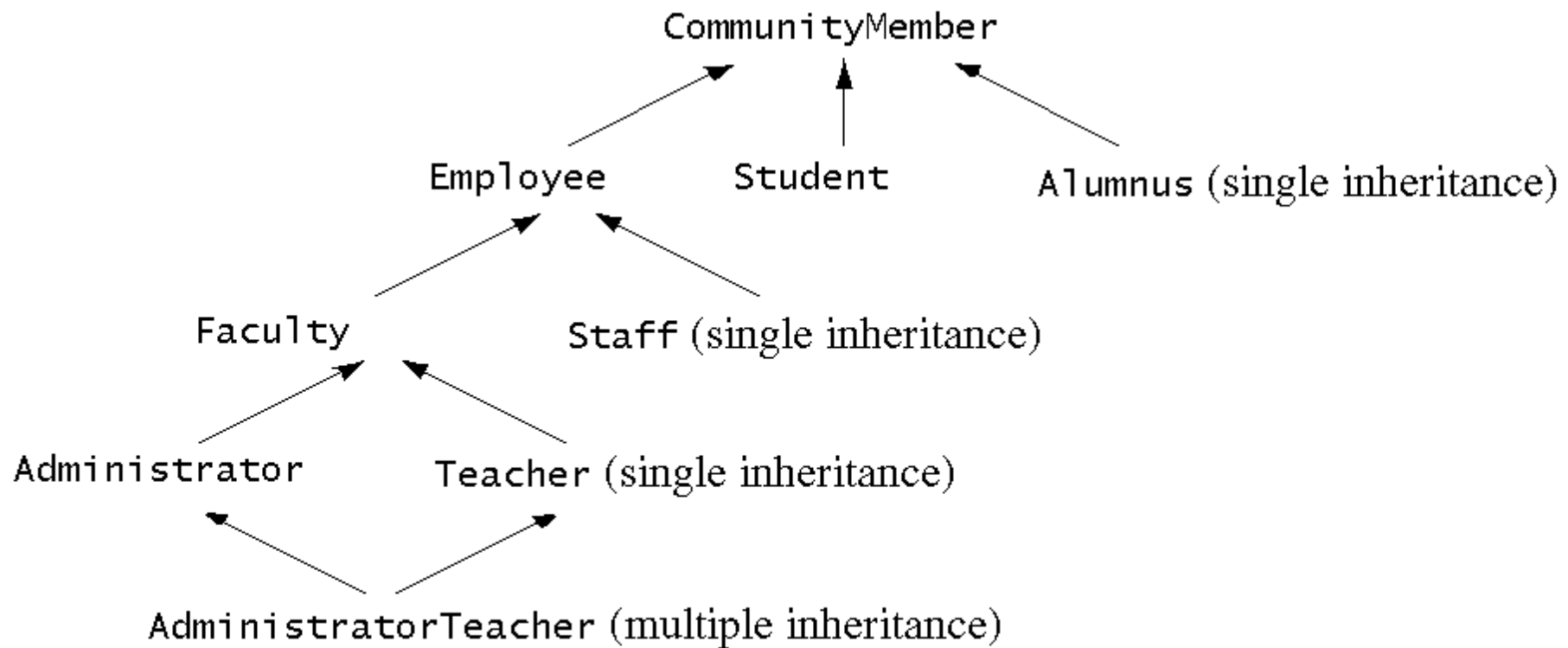


Fig. 19.2 An inheritance hierarchy for university community members.

# Inheritance

- Another way to reuse existing software to create a new one is by means of inheritance (also called *specialization* or *derivation*)
- This is often referred as an “**is-a**” relationship because every object of the class being defined “**is**” also an object of the inherited class.
- The common syntax for the deriving class Y from class X is

```
class Y : public X {  
    // ...  
};
```

# Inheritance

- Another way to reuse existing software to create a new one is by means of inheritance (also called *specialization* or *derivation*)
- This is often referred as an “**is-a**” relationship because every object of the class being defined “**is**” also an object of the inherited class.
- The common syntax for the deriving class Y from class X is

Derived class (or subclass)

Base class (or superclass)

```
class Y : public X {  
// ...  
};
```

public specify *public inheritance* →  
public members of the base class are  
public members of the derived class



# Inheritance

- Inheritance
  - **Single** Inheritance : Class inherits from one base class
  - **Multiple** Inheritance: Class inherits from multiple base classes
  - Three types of inheritance:
    - **public**: Derived objects are accessible by the base class objects
    - **private**: Derived objects are inaccessible by the base class
    - **protected**: Derived classes and **friends** can access protected members of the base class

# Base and Derived Classes

- Implementation of **public** inheritance

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

Class **CommissionWorker** inherits from class **Employee**

- **friend** functions not inherited
- **private** members of base class not accessible from derived class

# Deriving a Student Class from Person1 Class

```
// Definizione Classe STUDENT

#ifndef STUDENT_H
#define STUDENT_H
#include <iostream>

using std::ostream;
using std::istream;

#include "Person1.h"
#include "Date.h"

class Student : public Person1
{
public:
    Student(string n="", string id="", int s=1);
    void setDOM(int m, int d, int y){dom.setDate(m,d,y)};
    void printDOM();
private:
    string id;        //student identification
    Date dom;        //student date of matriculation
    int credits;     //course credit
    float gpa;       // grade-point average
    //name and sex are implemented in Person
};

#endif // STUDENT_H
```

# Deriving a Student Class from Person1 Class

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

#include "Student.h"

Student::Student(string n, string id, int s)
    : Person1(n, "Italy", s), id(id), credits(0) {}

void Student::setDOM(int m, int d, int y)
{
    dom.setDate(m, d, y);
}

void Student::printDOM() {
    cout<<dom;
}
```

Note how initialization works



# Deriving a Student Class from Person1 Class

```
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

// #include "Date.h"
// #include "Person1.h"
#include "Student.h"

int main()
{
    Student x("Anna Rossi", "123456789K" ,0);
    x.setDOB(4,13,1996);
    x.setDOM(7,9,2016);
    x.printName();
    cout<<" Born on ";
    x.printDOB();
    cout<<" Matriculated on ";
    x.printDOM();
    cout<<".\n"<<endl;
    return 0;
}
```

Anna Rossi Born on April 13, 1996 Matriculated on July 9, 2016.

# protected class members

- The `Student` class in has a significant problem: it cannot directly access the private data members of its `Person1` superclass: `name`, `nationality`, `DOB`, `DOD`, and `sex`.
- The lack of access on the first four of these is not serious because these can be written and read through the `Person` class's constructor and public access functions.
- However, there is no way to write or read a student's `sex`.
- One way to overcome this problem would be to make `sex` a data member of the `Student` class. But that is unnatural: `sex` is an attribute that all `Person` objects have, not just `Students`.
- A better solution is to change the private access specifier to **protected** in the `Person` class.
- That will allow access to these data members from derived classes.

# Person class with protected Data Members

```
// Definizione Classe PERSON2

#ifndef PERSON2_H
#define PERSON2_H
#include <iostream>
#include "Date.h"

using std::ostream;
using std::istream;

class Person2
{
public:
    Person2(string n="", string nat="U.S.A.", int s=1);
    void setDOB(int m, int d, int y);
    void setDOD(int m, int d, int y);
    void printName() ;
    void printNationality();
    void printDOB();
    void printDOD();
protected: ←
    string name, nationality;
    Date dob, dod;
    int sex;
};

#endif // PERSON2_H
```

Person2.cpp

# Person class with protected Data Members

```
// Definizione Classe STUDENT

#ifndef STUDENT_H
#define STUDENT_H
#include <iostream>

using std::ostream;
using std::istream;

#include "Person2.h"
#include "Date.h"

class Student : public Person2
{
public:
    Student(string n="", string id="", int s=1);
    void setDOM(int m, int d, int y);
    void printDOM();
    void printSex();
protected:
    string id;        //student identification
    Date dom;        //student date of matriculation
    int credits;     //course credit
    float gpa;       // grade-point average
    //name and sex are implemented in Person
};

#endif // STUDENT_H
```

Student.{h,cpp}



# Person class with protected Data Members

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

#include "Student.h"

Student::Student(string n, string id, int s)
    : Person2(n,"Italy",s), id(id), credits(0) {}

void Student::setDOM(int m, int d, int y)
{
    dom.setDate(m,d,y);
}

void Student::printDOM(){
    cout<<dom;
}

void Student::printSex(){
    cout<<(sex? "male":"female");
}
```

useStudent1.cpp

# Person class with protected Data Members

```
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

// #include "Date.h"
// #include "Person1.h"
#include "Student.h"

int main()
{
    Student x("Anna Rossi", "123456789K" ,0);
    x.setDOB(4,13,1996);
    x.setDOM(7,9,2016);
    x.printName();
    cout<<"\nSex ";
    x.printSex();
    cout<<" Born on ";
    x.printDOB();
    cout<<" Matriculated on ";
    x.printDOM();
    cout<<"\n"<<endl;
    return 0;
}
```

Anna Rossi

Sex female Born on April 13, 1996 Matriculated on July 9, 2016.

# protected Data Members

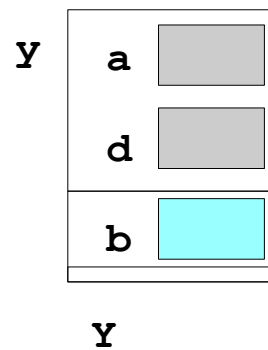
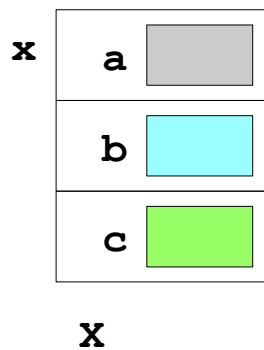
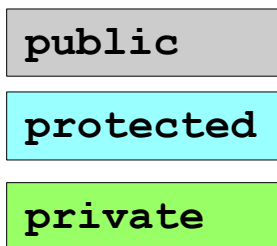
- The protected access category is a balance between `private` and `public` categories:
  - **private** members are accessible only from within the class itself and its `friend` classes;
  - **protected** members are accessible from within the class itself, its `friend` classes, its derived classes, and their `friend` classes;
  - **public** members are accessible from anywhere within the file.
- In general, `protected` is used instead of `private` whenever it is anticipated that a sub-class might be defined for the class.
  - A subclass inherits all the `public` and `protected` members of its base class. This means that, from the point of view of the subclass, the `public` and `protected` members of its base class appear as though they actually were declared in the subclass.

# protected Data Members

```
class X
{
public:
    int a;
protected:
    int b;
private:
    int c;
};

class Y : public X
{
public:
    int d;
};
```

```
X x;
Y y;
```



# Overriding and dominating inherited members

- If  $Y$  is a subclass of  $X$ , then  $Y$  objects inherit all the public and protected member data and member functions of  $X$ .
- In some cases, you might want to define a local version of an inherited member. For example, if  $a$  is a data member of  $X$  and if  $Y$  is a subclass of  $X$ , then you could also define a separate data member named  $a$  for  $Y$ .
- In this case, the  $a$  defined in  $Y$  **dominates** the  $a$  defined in  $X$ . Then a reference  $y.a$  for an object  $y$  of class  $Y$  will access the  $a$  defined in  $Y$  instead of the  $a$  defined in  $X$ .
- To access the  $a$  defined in  $X$ , one would use  $y.x::a$ .
- The same rule applies to member functions: if a function named  $f()$  is defined in  $X$  and another function named  $f()$  with the same signature is defined in  $Y$ , then  $Y.f()$  invokes the latter function, and  $y.x::f()$  invokes the former.
- In this case, the local function  $y.f()$  **overrides** the  $f()$  function defined in  $X$  unless it is invoked as  $y.x::f()$ .

# Overriding and dominating inherited members

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

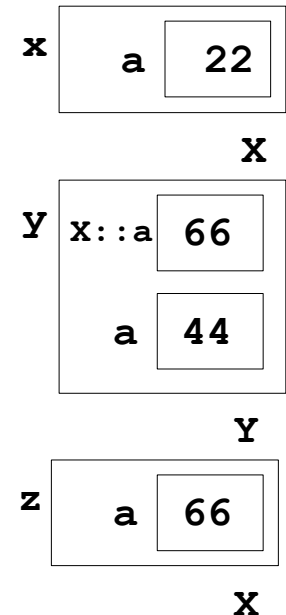
class X {
public:
    void f() {cout<<"X::f() executing\n";}
    int a;
};

class Y : public X {
public:
    void f() { cout << "Y::f() executing\n";}
    int a;
};

int main(){
    X x;
    x.a = 22;
    x.f();
    cout << "x.a = " <<x.a<<endl;
    Y y;
    y.a = 44;
    y.X::a = 66;
    y.f();
    y.X::f();
    cout << "y.a = " <<y.a<<endl;
    cout << "y.X::a = " <<y.X::a << endl;
    X z = y;
    cout << "z.a = : " << z.a << endl;
}
```

```
//assign 44 to the a defined in Y
//assign 66 to the a defined in X
//invokes the f() defined in X
//invokes the f() defined in Y
```

```
X::f() executing
x.a = 22
Y::f() executing
X::f() executing
y.a = 44
y.X::a = 66
z.a = : 66
```



DominatingOverriding.cpp

# virtual functions and polymorphism

- One of the most powerful features of C++ is that it allows objects of different types to respond differently to the same function call.
- This is called **polymorphism** and it is achieved by means of `virtual` functions.
- Polymorphism is rendered possible by the fact that a pointer to a base class instance may also point to any subclass instance

```
class X
{ //
};

class Y : public X // Y is a subclass of X
{ //
};

int main() {
    X* p;           // p is a pointer to object of class X
    Y y;
    p = &y;        // p can also point to object of subclass Y
}
```

# virtual functions and polymorphism

- If  $p$  has type  $X^*$  (“pointer to type  $x$ ”), then  $p$  can also point to any object whose type is a subclass of  $X$ . However, even when  $p$  is pointing to an instance of a subclass  $Y$ , its type is still  $X^*$ .
- An expression like  $p \rightarrow f()$  would invoke the function  $f()$  defined in the base class.
  - Recall that  $p \rightarrow f()$  is an alternate notation for  $(*p) . f()$
- This invokes the member function  $f()$  of the object to which  $p$  points.
- $p \rightarrow f()$  will always execute  $x :: f()$  because  $p$  had type  $X^*$ .
- The fact that  $p$  happens to be pointing at that moment to an instance of subclass  $Y$  is irrelevant; it’s the statically defined type  $X^*$  of  $p$  that normally determines its behavior.



# virtual functions and polymorphism

```
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;

class X {
public:
    void f() {cout<<"X::f() executing\n";}
};
class Y : public X {
public:
    void f() { cout << "Y::f() executing\n";}
};

int main(){
    X x;
    Y y;
    X *p = &x;           // invokes X::f() because p has type X*
    p->f();
    p = &y;             // invokes X::f() because p has type X*
    p->f();
}
```

```
X::f() executing
X::f() executing
```

Two function calls  $p \rightarrow f()$  are made. Both calls invoke the same version of  $f()$  that is defined in the base class  $X$  because  $p$  is declared to be a pointer to  $X$  objects. Having  $p$  point to  $y$  has no effect on the second call  $p \rightarrow f()$ .

virtualfunction.cpp

# virtual functions and polymorphism

```
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
using std::string;

class X {
public:
    virtual void f() {cout<<"X::f() executing\n";}
};
class Y : public X {
public:
    void f() { cout << "Y::f() executing\n";}
};

int main(){
    X x;
    Y y;
    X *p = &x;           // invokes X::f() because p has type X
    p->f();
    p = &y;              // invokes Y::f()
    p->f();
}
```

This example illustrates **polymorphism**: the same call  $p \rightarrow f()$  invokes different functions. The function is selected according to which class of object  $p$  points to. This is called **dynamic binding** because the association (i.e., binding) of the call to the actual code to be executed is deferred until run time. The rule that the pointer's statically defined type determines which member function gets invoked is overruled by declaring the member function **virtual**.

```
X::f() executing
Y::f() executing
```

virtualfunction1.cpp

# virtual functions and polymorphism

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

class Person
{public:
    Person(string n=""): name(n) {}
    void print ()      { cout << "My name is: " << name << endl; }
protected:
    string name;
};

class Student : public Person
{public:
    Student(string n="", float g = 0): Person(n), gpa(g) {}
    void print ()      { cout << "My name is: " << name << " and my gpa is: " << gpa << endl; }
private:
    float gpa;
};

class Professor : public Person
{public:
    Professor(string n="", int p = 0): Person(n), publs(p) {}
    void print ()      { cout << "My name is: " << name << " and I have: " << publs << " publications " << endl; }
private:
    float publs;
};
```

Polymorphism.cpp

# virtual functions and polymorphism

```
int main()
{
    Person *p;
    Person x("Bob");
    p = &x;
    p->print();
    Student y("Tom",28.8);
    p = &y;
    p->print();
    Professor z("Ann",52);
    p = &z;
    p->print();
    return 0;
}
```

```
My name is: Bob
My name is: Tom
My name is: Ann
```

The `print()` function defined in the base class is not virtual. So the call `p->print()` always invokes that same base class function `Person::print()` because `p` has type `Person*`. The pointer `p` is *statically bound* to that base class function at compile time.

# virtual functions and polymorphism

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

class Person
{public:
    Person(string n=""): name(n) {}
    virtual void print () { cout << "My name is: "<<<name<<endl; }
protected:
    string name;
};

class Student : public Person
{public:
    Student(string n="", float g = 0): Person(n),gpa(g) {}
    void print () { cout << "My name is: "<<<name<<" and my gpa is:
"<<gpa<< endl; }
private:
    float gpa;
};

class Professor : public Person
{public:
    Professor(string n="", int p = 0): Person(n),publs(p) {}
    void print () { cout << "My name is: "<<<name<<" and I have: "<<<publs<<
" publications "<<endl; }
private:
    float publs;
};
```

Polymorphism.cpp

# virtual functions and polymorphism

```
int main()
{
    Person *p;
    Person x("Bob");
    p = &x;
    p->print();
    Student y("Tom",28.8);
    p = &y;
    p->print();
    Professor z("Ann",52);
    p = &z;
    p->print();
    return 0;
}
```

My name is: Bob

My name is: Tom and my gpa is: 28.8

My name is: Ann and I have: 52 publications

Now the pointer `p` is *dynamically bound* to the `print()` function of whatever object it points to. The call `p->print()` is **polymorphic** because its meaning changes according to the circumstance

# Esercitazione 9

## Esercizio 1

Implement a Cerchio class that inherits from the Punto class.

An object of the Punto class will be the center of the circle. For the Punto class, implement two get functions (one for the x and one for the y coordinates) and one set function (i.e. you set x and y with a single function).

For the Cerchio class implement the functions SetRadius() and GetRadius() and GetArea().

Start from the Exercise6 in Esercitazione7 for the implementation of the Punto class

Example of execution:

```
./cerchio
```

```
Center and Radius
```

```
x: 1 y: 1 r: 4
```

```
New Center Coordinates [3, 2]
```

```
Area : 50.24
```