# Composition and Inheritance

## Objectives

- Ofter the usage of exiting 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.

## 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.{cpp,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 << nationali
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

```cpp
#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 exiting 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

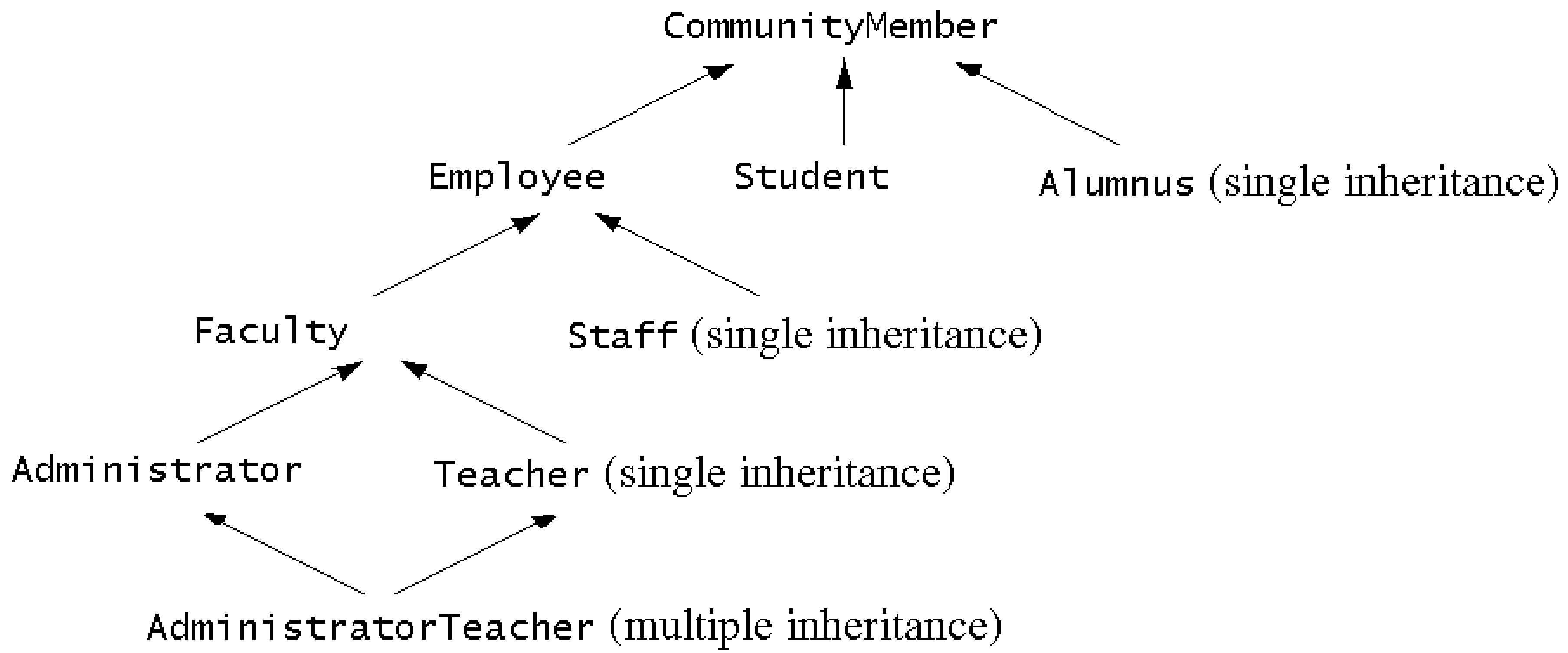

CommunityMember
Employee
Student
Alumnus (single inheritance)
Faculty
Staff (single inheritance)
Administrator
Teacher (single inheritance)
AdministratorTeacher (multiple inheritance)

Fig. 19.2 An inheritance hierarchy for university community members.

## Inheritance

- Another way to reuse exiting 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 exiting 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 `friend`s 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;
}
```

## 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

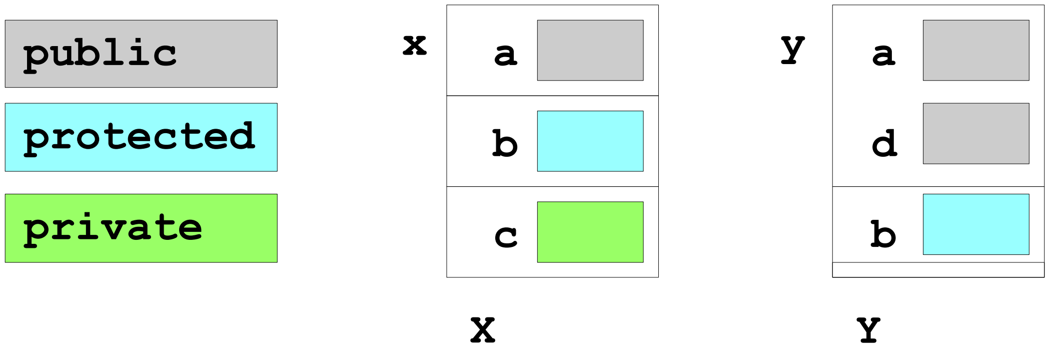
- The protected access category is a balance between `private` and `public` categories:
  - **`private`** members are accessible only from within the class itself and its `friend` classes;
  - **`protected`** members are accessible from within the class itself, its `friend` classes, its derived classes, 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 X : public Y
{
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()`.

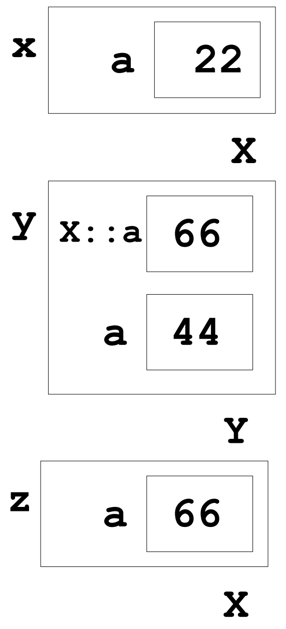
# Overriding and dominating inherited members

```
#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;                         //assign 44 to the a defined in Y
  y.X::a = 66;                      //assign 66 to the a defined in X
  y.f();                            //invokes the f() defined in X
  y.X::f();                         //invokes the f() defined in Y
  cout << "y.a = "<<y.a<<endl;
  cout << "y.X::a = " <<y.X::a << endl;
  X z = y;
  cout << "z.a = : "<< z.a << endl;
}
```

```
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→f()` would invoke the function `f()` defined in the base class.
  - Recall that `p→f()` is an alternate notation for `(*p).f()`
- This invokes the member function `f()` of the object to which `p` points.
- `p→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→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→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 ty
  p->f();
  p = &y;             // invokes Y::f()
  p->f();

}
```

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

This example illustrates **<u>polymorphism</u>**: the same call `p→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 <u>run time</u>. The rule that the pointer's statically defined type determines which member function gets invoked is overruled by declaring the member function `virtual`.

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.

Polymorphism.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) {}
  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

Polymorphism.cpp

## 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 the x and one for the y coordinates) and one set function (i.e. you set x and y with a single function). Override the "<<" operator with an ostream like [x,y].

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

Start from the Exercise 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
```

## Additional stuff

- This is not a backup: this section contains “random” useful stuff not mentioned up to now

## Overloading function

- C++ allows you to use the same name for different functions.
- As long as they have different parameter type lists, the compiler will regard them as different functions.
- To be distinguished, the parameter lists must either contain a different number of parameters, or there must be at least one position in their parameter lists where the tyes are different.

## Overloading `max()` function

```
#include <iostream>

using std::cout;   // program uses cout
using std::endl;   // program uses endl

int max(int, int);
int max(int, int, int);

int main(){
  cout<<"Max(10,20): " <<max(10,20)<<"\nMax(2,9,6): "<<max(2,9,6)<<endl;
  return 0;
}

int max(int x, int y){
  return (x>y ? x: y);
}

int max(int x, int y, int z){
  int m = (x>y ? x: y); // m = max(x,y)
  return (z>m ? z: m);
}
```

```
Max(10,20): 20
Max(2,9,6): 9
```

OverloadingMax.cpp

# Command line arguments in C/C++

- The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

- We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
- To pass command line arguments, we typically define main() with two arguments: first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **argc** (**ARG**ument **C**ount) is a `int` and stores number of command-line arguments passed by the user <u>including the name of the program</u>. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv**(**ARG**ument **V**ector) is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to <u>strings</u>.
- `argv[0]` is the name of the program , After that till argv[argc-1] every element is command-line arguments.

## Command line arguments in C/C++

```
#include <iostream>
#include <cstdlib>

using std::cout;  // program uses cout
using std::endl;  // program uses endl

int max(int, int);
int max(int, int, int);

int main(int argc, char* argv[]){

  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  int c = atoi(argv[3]);
  // Converting string type to integer type
  // using function "atoi( argument)"

  cout<<"Max("<<a<<","<<b<<"): " <<max(a,b)<<endl;
  cout<<"Max("<<a<<","<<b<<","<<c<<"): "<<max(a,b,c)<<endl;
  return 0;
}

int max(int x, int y){
  return (x>y ? x: y);
}

int max(int x, int y, int z){
  int m = (x>y ? x: y); // m = max(x,y)
  return (z>m ? z: m);
}
```

## Command line arguments in C/C++

```
g++ OverloadingMaxArg.cpp -o max
./max 4 5 6
Max(4,5): 5
Max(4,5,6): 6
```

Important: if you define a function that needs some argument, you have to specify them at the runtime, otherwise you will get a segmentation violation!!!

## File input

- To get data from a file, we have to create a stream that flows from the file into the program.
- We can do that using the `ifstream` constructor.

```
ifstream infile ("file-name");
```

- The argument for this constructor is a string that contains the name of the file you want to open.
- The result is an object named `infile` that supports all the same operations as `cin`, including `>>` and `getline`.

```
int x;
apstring line;
infile >> x;
getline (infile, line);
// get a single integer and store in x
// get a whole line and store in line
```

# File input

- To get data from a file, we have to create a stream that flows from the file into the program.
- We can do that using the `ifstream` constructor.

```
ifstream infile ("file-name");
```

- The argument for this constructor is a string that contains the name of the file you want to open.
- The result is an object named `infile` that supports all the same operations as `cin`, including `>>` and `getline`.

```
int x;
apstring line;
infile >> x;
getline (infile, line);
// get a single integer and store in x
// get a whole line and store in line
```

## File input

- If we know ahead of time how much data is in a file, it is straightforward to write a loop that reads the entire file and then stops.
- More often, though, we want to read the entire file, but don't know how big it is.
- There are member functions for `ifstreams` that check the status of the input stream; they are called **`good`**, **`eof`**, **`fail`** and **`bad`**.
- We will use `good` to make sure the file was opened successfully and `eof` to detect the "end of file."
- Whenever you get data from an input stream, you don't know whether the attempt succeeded until you check. If the return value from eof is true then we have reached the end of the file and we know that the last attempt failed.

## File input

- Here is a program that reads lines from a file and displays them on the screen:

```
apstring fileName = ...;
ifstream infile (fileName.c_str());
if (infile.good() == false) {
     cout << "Unable to open the file named " <<
fileName;
   exit (1);
 }
while (true) {
   getline (infile, line);
   if (infile.eof()) break;
   cout << line << endl;
}
```

## File input

- Here is a program that reads lines from a file and displays them on the screen:

```
apstring fileName = ...;
ifstream infile (fileName.c_str());
if (infile.good() == false) {
     cout << "Unable to open the file named " <<
fileName;
   exit (1);
 }
while (true) {
   getline (infile, line);
   if (infile.eof()) break;
   cout << line << endl;
}
```

The function `c_str` converts an `apstring` to a native C string. Because the `ifstream` constructor expects a C string as an argument, we have to convert the apstring.

- `good` function: return value is false if the system could not open the file (most likely because it does not exist, or you do not have permission to read it)
- The statement `while(true)` is an idiom for an infinite loop: usually there will be a break statement somewhere in the loop so that the program does not really run forever.
- The `break` statement allows us to exit the loop as soon as we detect the end of file.
- It is important to exit the loop between the input statement and the output statement, so that when `getline` fails at the end of the file, we do not output the invalid data in line.

## File output

- Sending output to a file is similar. For example, we could modify the previous program to copy lines from one file to another.

```
ifstream infile ("input-file");
ofstream outfile ("output-file");
if (infile.good() == false || outfile.good() == false) {
     cout << "Unable to open one of the files." << endl;
     exit (1);
}
while (true) {
     getline (infile, line);
     if (infile.eof()) break;
     outfile << line << endl;
}
```

# Esercitazione 9

## Esercizio 2

- Write a macro where the function min(a,b) and min(a,b,c) are overloaded. The argument of the function can be with the cin operator.
- Repeat the exercise passing the argument from the command line.

```
./min
Insert how many numbers you want to compare
2
Insert the first number: 3
Insert the second number: 6
The minimum is: 3

./minArg 3 3 4 6
Min(3,4,6): 3
```