Cairo University, Faculty of Computers and AI

CS213 – 2022 / 2023

Programming II

Lecture 4: OOP – III – Inheritance

By

Dr. Mohammad El-Ramly

Lecture Objectives

1. Separate Compilation

   `#ifndef ….. #endif`

2. Introduction to Inheritance
3. Implementing Inheritance in C++
4. More details about Inheritance

1. Separate Compilation

- A way to organize and manage your code.
- C++ allows you to divide a program into parts
  - Each part can be stored in a separate file
  - Each part can be compiled separately
  - All the files can be linked manually or with a **make** file.

Why are Separate Files Desirable?

- Human mind can only grasp a page or two of code at a time
- By separate compilation
  - Program is more understandable.
  - Classes can be reused in a new program
  - Teams of programmers can work on the same program separately.

Example

```
#include "helloworld.hpp"
int main(){
  hello_world();
  return 0;
}
```

```
#ifndef _HELLOWORLD_H
#define _HELLOWORLD_H

#include <iostream>
void hello_world();

#endif
```

```
#include "helloworld.h"
#include <iostream>
using namespace std;
void hello_world(){
 cout << "Hello World!\n";
}
```

file: **main.cpp**

file: **helloworld.hpp**

file: **helloworld.cpp**

Naming The Header File - red box

- A header file contains declarations and other basic information needed by a program to use a library of functions (or classes).
- It separates information that would have to be repeated for different parts of the program.
- They have the suffix **.h** or **.hpp**.
- To use a header file you must include it

```
#include "XXX.hpp"
```

#include " " or < > ?

- To include a predefined header file use < and >

```
#include <iostream>
```

  - < and > tells the compiler to look where the system stores predefined header files
- To include a header file you wrote, use the double quotes

```
#include "XXX.hpp"
```

  - The double quotes usually cause the compiler to look in the current directory for the header file

The Implementation File – blue box

- Contains the definitions of the functions / classes you wish to separate from the main function.
- Often has the same name as the header file but a different suffix
    - Since our header file is named XXX.hpp, the implementation file often is named XXX.cpp

The Application File - browbox

- The Application file is the file that contains the program that is usually the main function.
  - It is also called a driver file
  - Must use an include directive to include the interface file:

```
#include "XXX.hpp"
```

Running The Program

- Basic steps required to run a program: (Details vary from system to system!)
  - Compile the implementation file
  - Compile the application file
  - Link the files to create an executable program using a utility called a linker
    - Linking is often done automatically

Compile XXX.hpp ?

- The interface file is not compiled separately
  - The preprocessor replaces any occurrence of #include "XXX.hpp" with the text of XXX.hpp before compiling
  - Both the implementation file and the application file contain #include "XXX.hpp"
    - The text of XXX.hpp is seen by the compiler in each of these files
    - There is no need to compile XXX.hpp separately

Introduction to **`#ifndef`**

- To prevent multiple declarations, we can use these directives:
  - **`#ifndef _DTIME_HPP`**
    checks to see if **`dtime.hpp`** has been defined
  - **`#define _DTIME_HPP`**
    adds **`dtime.h`** to a list indicating dtime.h has been seen
  - **`#endif`**
    If **`dtime.hpp`** has been defined, skip to **`#endif`**

Example

```
#ifndef _HELLOWORLD_HPP
#define _HELLOWORLD_HPP
#include <iostream>

void hello_world();

#endif
```

- First time a **`#include "helloworld.hpp"`** is found, **`helloworld.hpp`** is defined
- Next time a **`#include "helloworld.hpp"`** is found, lines between **`#ifndef`** and **`#endif`** are skipped

Why _HELLOWORLD_HPP?

- HELLOWORLD_HPP is the normal convention for creating an identifier to use with ifndef
  - It is the file name in all caps
  - Use ' _ ' instead of ' . '
  - Use '_' in the beginning
- You may use any other identifier, but will make your code more difficult to read

Defining You OWN Libraries

- You can create your own libraries of functions
  - If you have a collection of functions…
    - **Declare** them in a header file with their comments
    - **Define** them in an implementation file
    - **Use** the library files just as you use predefined libraries.

2. Introduction to Inheritance

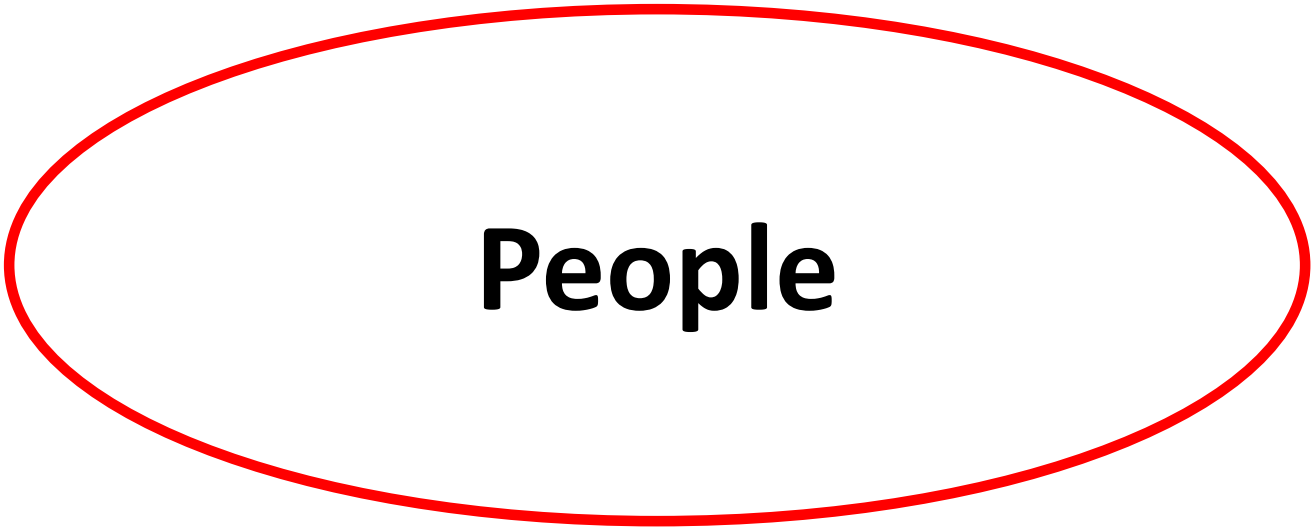
- Most powerful feature of OOP
- Similar to inheritance in real life
- New classes are created from existing classes
- New classes absorb all features of existing classes including their data and functions. Also enhance them by adding their own new features in form of new data members and new member functions

Introduction to Inheritance

- Existing classes are called base (super) classes
- New classes are called derived (sub) classes
- Objects of derived classes are more specialized as compared to objects of their base classes
- Inheritance provides us a mechanism of software reusability which is one of the most important principles of software engineering

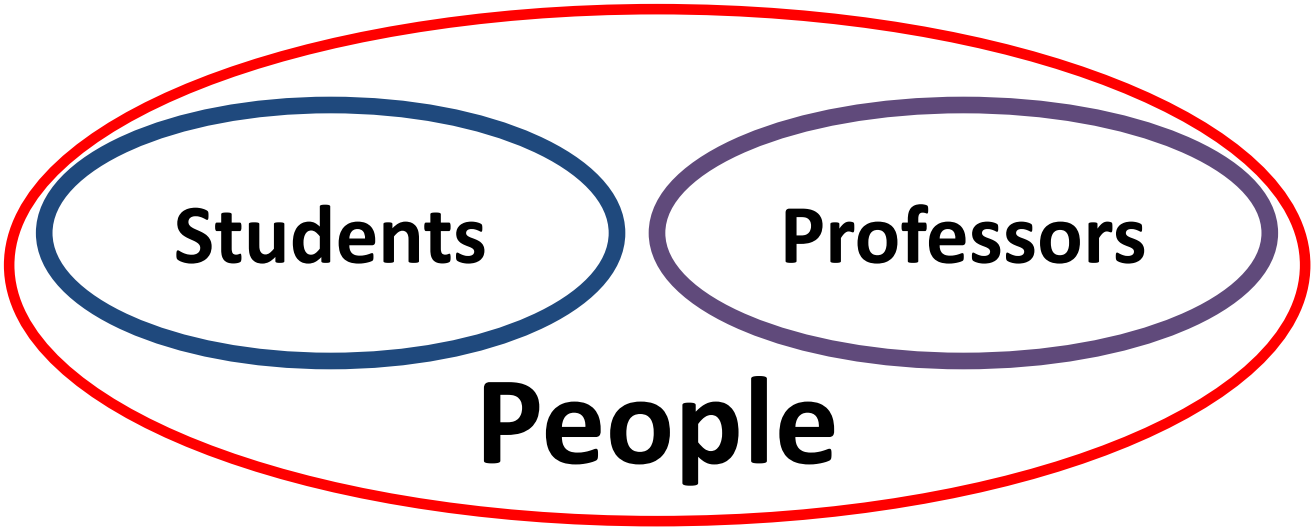
A Class is a Type

- A class defines a set of objects, or a type
- People



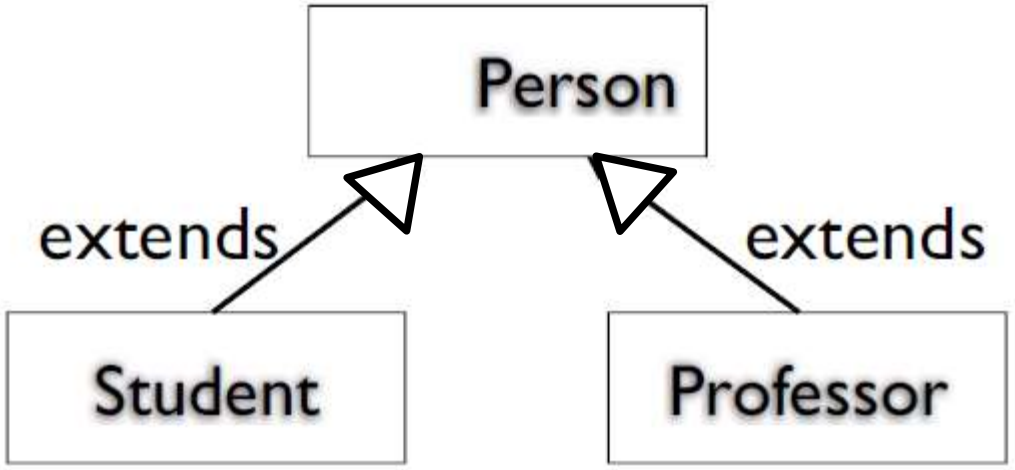
There May Be Subtypes

- Some objects are distinct from others in some ways
- Types of people



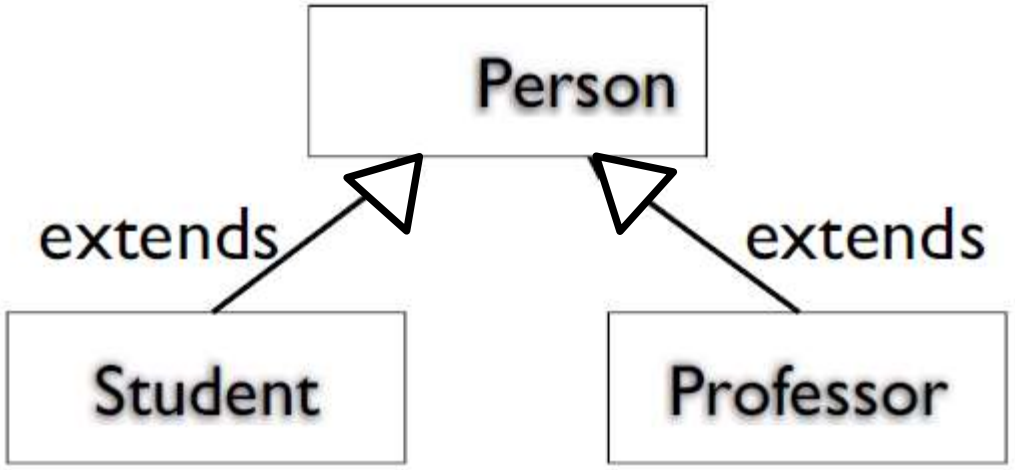
Type Hierarchy

- How are students and professors similar / different ?
- In attributes? In Behaviors?



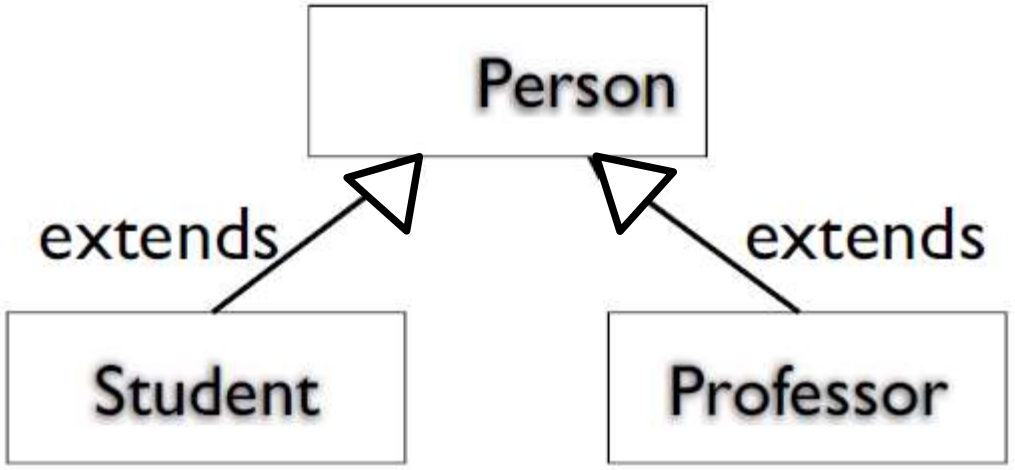
Common Attributes / Behaviors

- name, ID, address
- change address, display profile



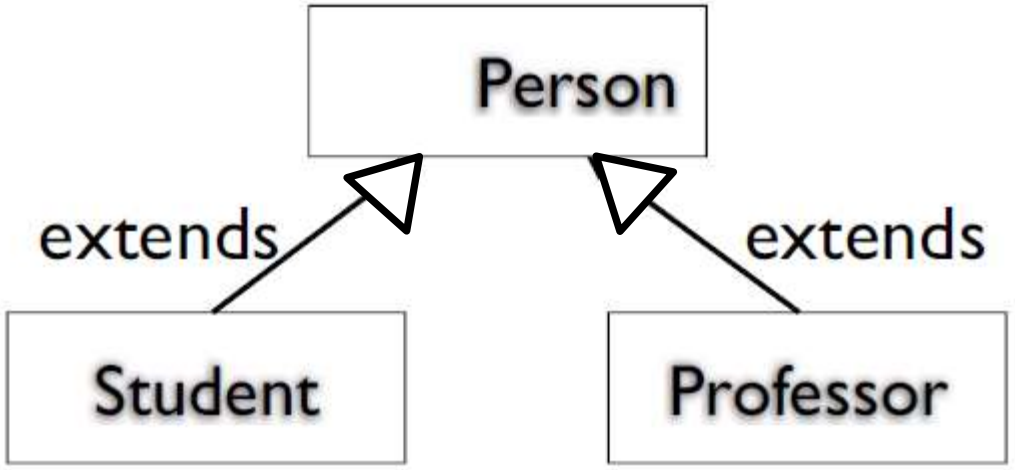
Different Attributes / Behaviors

- Students
  - course number, classes taken, year
  - add a class taken, change course



Different Attributes / Behaviors

- Professors
  - course number, classes taught, rank
  - add a class taught, promote



Inheritance

- A subtype inherits characteristics and behaviors of its base type.
- e.g. Each student has

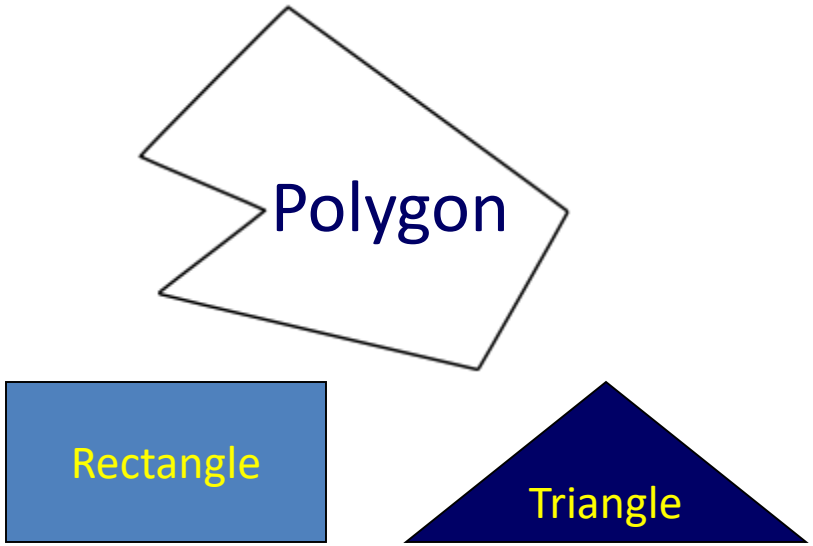
Characteristics:	Behaviors:
name	display profile
ID	change address
address	add a class taken
course number	change course
classes taken	
year	

Inheritance

- A subtype inherits characteristics and behaviors of its base type.
- e.g. Each Professor has

Characteristics:	Behaviors:
name	display profile
ID	change address
address	add a class taught
course number	promote
class taught	
rank	

Example on Inheritance

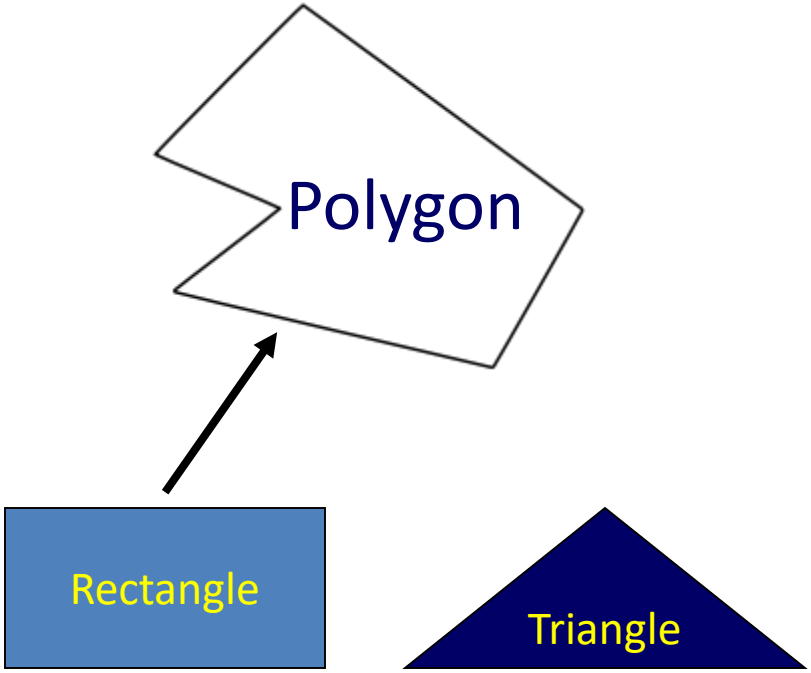



```
class Rectangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

```
class Polygon{
  private:
    int    numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

```
class Triangle{
  private:
    int    numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

Example on Inheritance

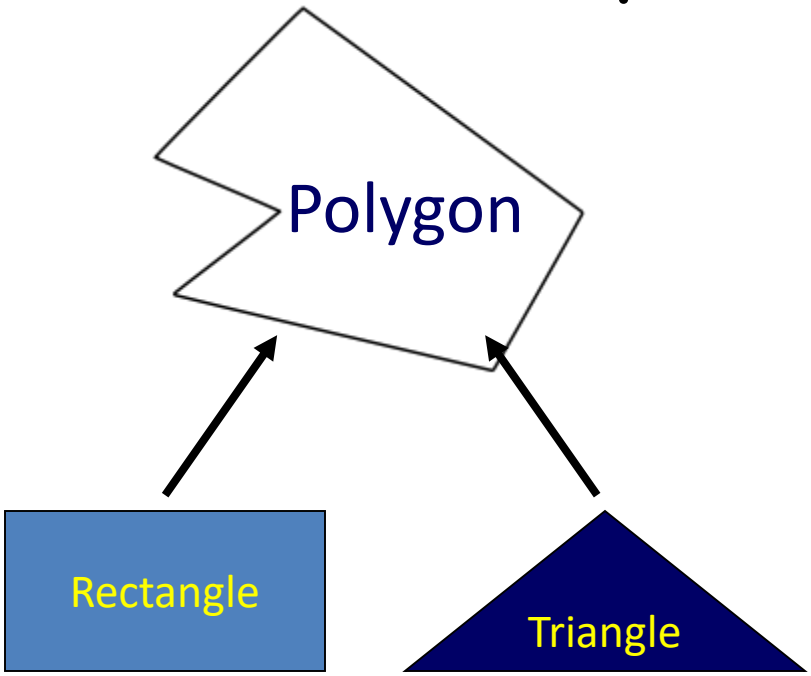



```
class Polygon{
    protected:
        int numVertices;
        float *xCoord, float *yCoord;
    public:
        void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
    public:
        float area();
};
```

```
class Rectangle{
    protected:
        int numVertices;
        float *xCoord, float *yCoord;
    public:
        void set(float *x, float *y, int nV);
        float area();
};
```

Example on Inheritance




```
class Polygon{
    protected:
        int numVertices;
        float *xCoord, float *yCoord;
    public:
        void set(float *x, float *y, int nV);
};
```

```
class Triangle : public Polygon{
    public:
        float area();
};
```

```
class Triangle{
    protected:
        int numVertices;
        float *xCoord, float *yCoord;
    public:
        void set(float *x, float *y, int nV);
        float area();
};
```

Why Inheritance ?

Inheritance is a mechanism for

- **building class types from existing class types**
- defining new class types to be a ***specialization*** of existing types

3. Implementing Inheritance

- In C++, we write
- **class** ChildClass **:** ***access modifier*** BaseClass { };
- This means that all members of BaseClass are now included in ChildClass
- ChildClass can have additional new members
- It can also override some of the parent's members.

Base Class: Person

```
#include <string>
class Person {
     protected:
          int id;
          std::string name;
          std::string address;
     public:
          Person (int id, std::string
               name, std::string address);
          void displayProfile();
          void changeAddress
               (std::string newAddress);
};
```

Access Control

Namespace Prefix

Access Control Modifiers

- **public:** accessible by anyone
- **protected:** accessible inside the class and by all of its subclasses
- **private:** accessible only inside the class, NOT including its subclasses

Base Class: Student

```
#include <iostream>
#include <vector>
#include "Person.h"
#include "Class.h"
class Student : public Person {
  protected:
     int course, year;
     std::vector<Class*> classesTaken;
  public:
     Student(int id, std::string name,
          std::string address, int course,
          int year);
     void displayProfile();
     void addClassTaken(Class* newClass);
     void changeCourse(int newCourse);
};
```

Inherit from parent class

Vector of pointers to Class

Base Class: Student

```
#include <iostream>
#include <vector>
#include "Person.h"
#include "Class.h"
class Student : public Person {
  protected:
     int course, year;
     std::vector<Class*> classesTaken;
  public:
     Student(int id, std::string name,
          std::string address, int course,
          int year) : Person (id, name, address);
     void displayProfile();
     void addClassTaken(Class* newClass);
     void changeCourse(int newCourse);
};
```

Pass some parameters to parent's constructor

Constructing an Object of Student

```
// in Student.cpp
Student::Student(int id, std::string name,
                         std::string address,
                         int course, int year)
                  : Person(id, name, address) {
      this->course = course;
      this->year = year;
}
// in MITPerson.cpp
Person::Person (int id, std::string name,
                        std::string address) {
      this->id = id;
      this->name = name;
      this->address = address;
}
```

Call one of the constructors of base class

Creating a Student Object

```
// in main.cpp
Student ali (20170001, "Ali Taha",
             "Giza", 6, 2);

Student* ptrAli = new Student (20170001, "Ali
                  Taha", "Giza", 6, 2);
```

Modes / Types of inheritance

- public
- private
- protected

Public Inheritance

- With public inheritance,
  - public and protected members of the base class become respectively public and protected members of the derived class.

Protected Inheritance

- Public and protected members of the base class become protected members of the derived class.

Private Inheritance

- With private inheritance, public and protected members of the base class become private members of the derived class.

Modes of Inheritance in C++

		Inheritance Mode		
		public	protected	private
Members in Base Class	public	public	protected	private
	protected	protected	protected	private
	private	X	X	X
		Members in derived class		

- https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm

4. More on Inheritance

- Inheritance is a way to organize and manage your code.
- It is a design choice you make wile modeling your domain.
- It is a way of extending existing classes with new functions.
- Or making special cases of the general classes.

Inheritance is ....

- Another mechanism for code reuse.
- A process of deriving classes from a base class without disturbing the implementation of the base class.
- Inheritance models the Is-A relationship. In a Is-a relationship the derived class is a variation of the base class.
- Ex: Vehicle ***is a*** Car

  Car is a Vehicle => Car derived from Vehicle

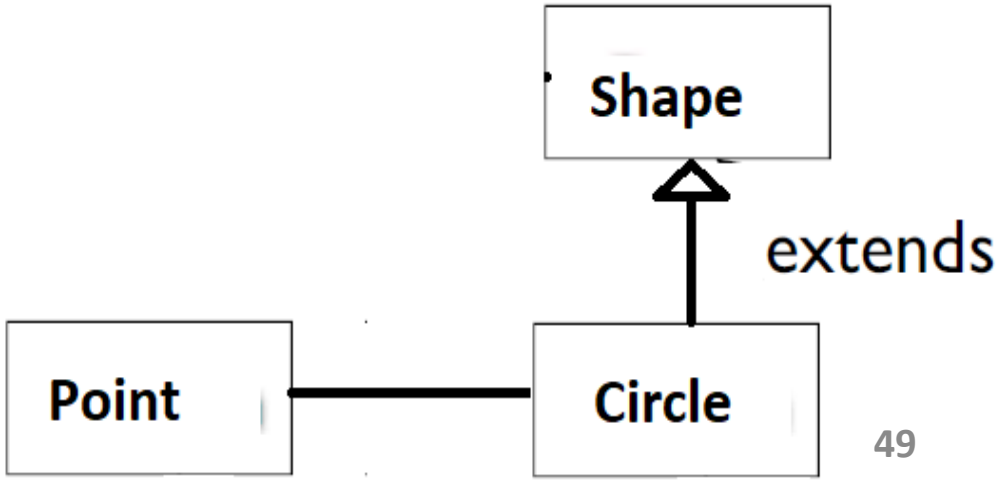
Inheritance vs. Association

- Inheritance: "is a"

```
class Circle : public Shape {
    ......
};
```

- Association: "has a"

```
class Circle {
private:
    Point center;
};
```



Some Decisions to Make

- What classes model my system? How Abstract?
- Attributes and operations of each class
- Public, private, protected levels of visibility
  - Public: visible everywhere
  - Protected: within class and subclass declarations
  - Private: visible only in class where declared
- Friend functions and classes
  - Careful attention to visibility and data abstraction

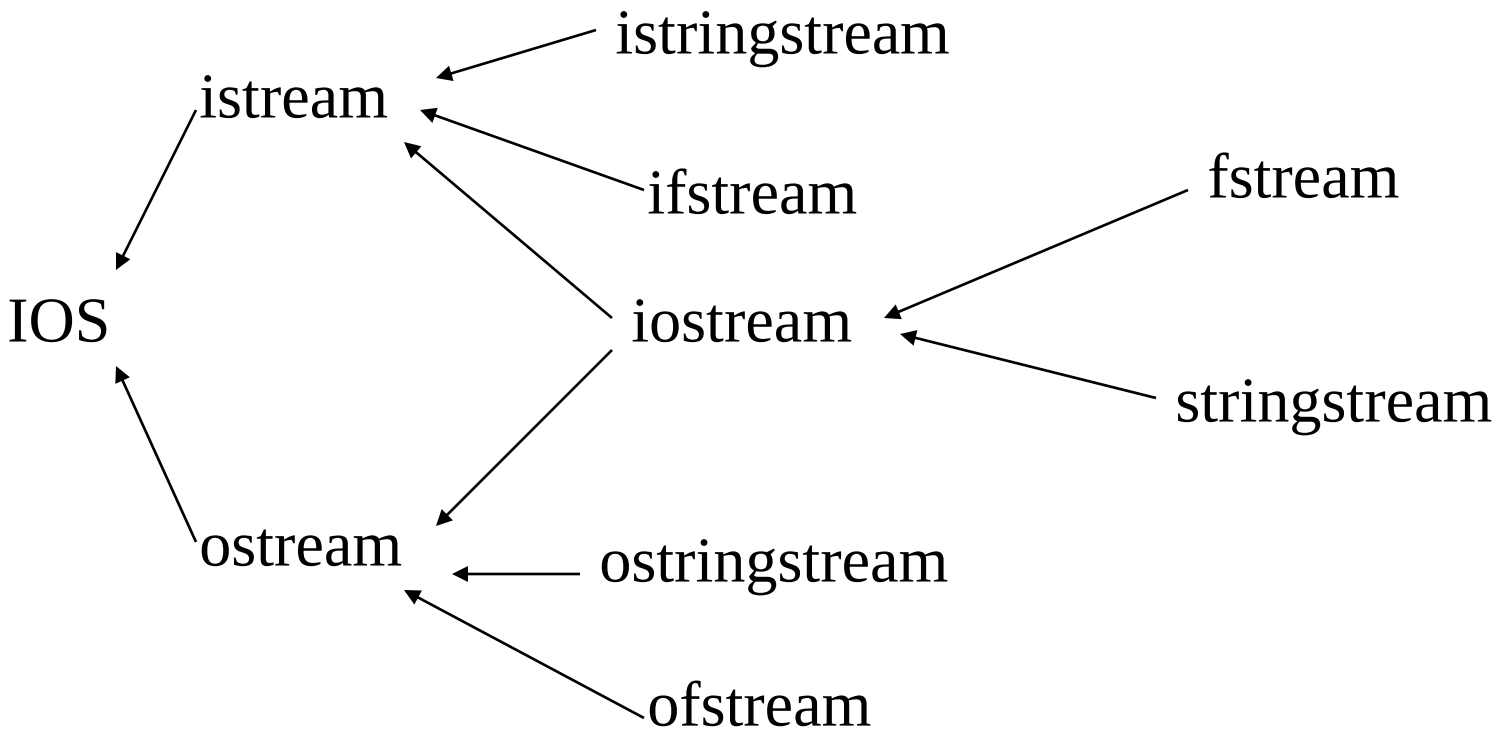
Access control

- `Public:` visible to everyone
- `Private:` visible only to the implementer of this particular class
- `Protected:` visible to this class and derived classes
- Good rule of thumb:
  - member functions `public` or `protected`
  - member variables `private`

Class relationships

- OK:
  - A has a data member of type B
  - A calls function from B
  - A creates B
- Bad:
  - A uses data directly from B (without using B's interface)
- Even worse:
  - A directly manipulates data in B

Hierarchy of C++ Stream Classes



What to inherit?

- In principle, every member (but not private) of a base class is inherited by a derived class
  - just with different access permission

Constructor Rules for Derived Classes

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
  public:
    A ( )
     {cout<< "A:default"<<endl;}
    A (int a)
     {cout<<"A:parameter"<<endl;}
};
```

```
class B : public A
{
  public:
    B (int a)
       {cout<<"B"<<endl;}
};
```

```
B test(1);
```

output:

```
A:default
B
```

Constructor Rules for Derived Classes

You can also specify an constructor of the base class other than the default constructor

DerivedClassCon (derivedClass args) : BaseClassCon (baseClass args)
{ DerivedClass constructor body }

```
class A {
  public:
    A ( )
     {cout <<  "A:default" << endl;}
    A (int a)
     {cout << "A:parameter" << endl;}
};
```

```
class C : public A {
  public:
    C (int a) : A(a)
       {cout<<"C"<<endl;}
};
```

```
C test(1);
```

output:

```
A:parameter
C
```

Even more …

- A derived class can override methods defined in its parent class. With overriding,
  - the method in the subclass has the identical signature to the method in the base class.
  - a subclass implements its own version of a base class method.

```
class A {
  protected:
    int x, y;
  public:
    void print ()
          {cout<<"From A"<<endl;}
};
```

```
class B : public A {
  public:
    void print ()
        {cout<<"From B"<<endl;}
};
```

Default class functions

- By default, each class has member functions:
  - constructor `Date();`
  - destructor `~Date();`
  - copy constructor
    ```
    Date(const Date& other);
    ```
  - assignment operator
    ```
    Date& operator=(const Date& other);
    ```
- These call the appropriate functions on each member variable
- **Be careful**: If this is not what you want, then either override or disallow (by making private)