Object Oriented Programming - Lecture 4

Diana Borza - diana.borza@ubbcluj.ro

March 21, 2022

Content

- OOP features (cont'd)
- Friend elements
- What's new in C++? (cont'd)

OOP - features

- **Abstraction**: separating an object's specification from its implementation.
- **Polymorphism**: allows an object to be one of several types, and determining at runtime how to "process" it, based on its type.
- **Inheritance**: organize classes to be arranged in a hierarchy that represents "IS A" relationships $\rightarrow$ easy re-use of the code, in addition to potentially mirroring real-world relationships in an intuitive way.
- **Encapsulation**: binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.

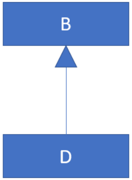
What's the object oriented way to get rich?

What's the object oriented way to get rich?

Inheritance

Inheritance I

- used to *derive a more specific* concept from a more general one;
- the specific concept must have the characteristics of the general concept, but it can have more;
- *derived* class inherits all of the data members and member functions of the *base* class (with the exception of constructors, destructor, and assignment operators).



Inheritance II

- Allows defining a new class (*subclass*) by using the definition of another class (*superclass*).
- Inheritance makes code *reusability* possible. Reusability refers to using already existing code (classes).
- The time and effort needed to develop a program are reduced, the software is more robust.
- The existing class is **not** modified. The new class can use all the features of the old one and add **new features** of its own.
- Inheritance can be used if there is a kind of or IS A relationship between the objects.

Inheritance

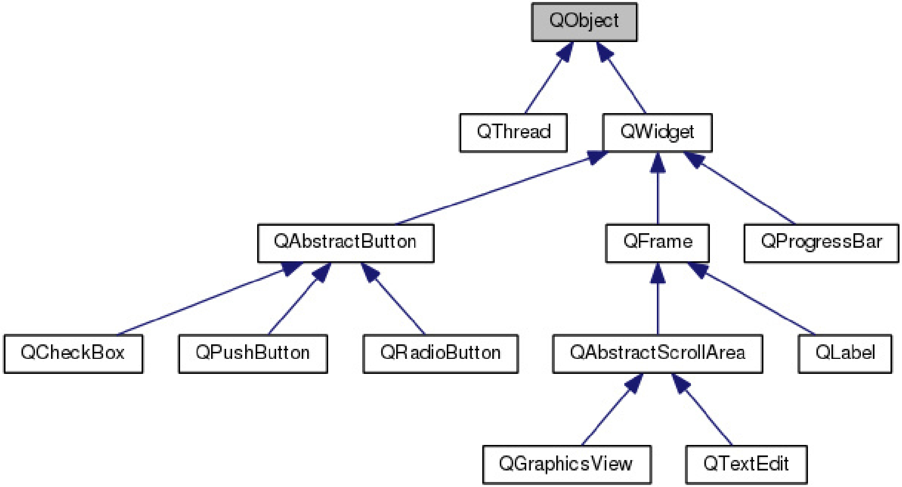
Let B (base) and D (derived) be two classes:

- D inherits from B - class D has all variables and methods of class B;
- D is derived from B - class D may redefine methods of class B;
- D is a specialization of B - class D may add new members besides the ones inherited from B.

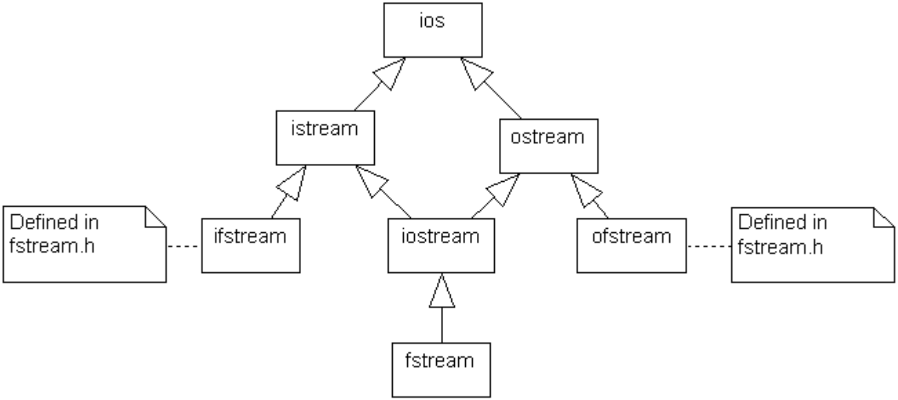
Inheritance - terminology

- class B = superclass, base class, parent class.
- class D = subclass, derived class, descendent class.
- inherited member (function, variable) = a member defined in B, and used unchanged in D.
- overridden member = defined in B and D.
- added member (new) = defined only in D.

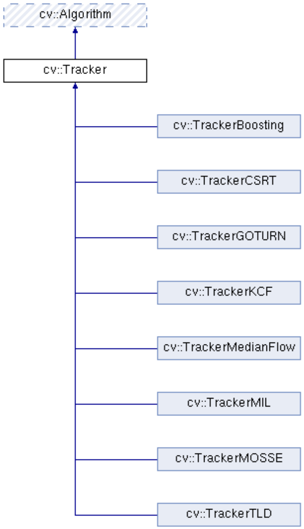
Inheritance - real world examples I - Qt framework

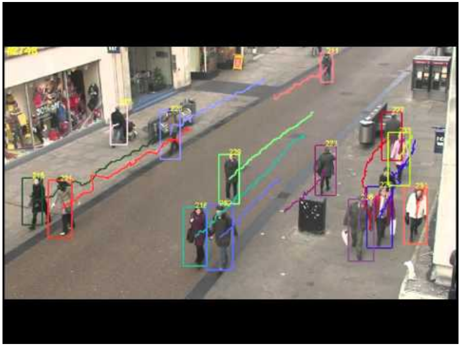


Inheritance - real world examples II - stl - IO



Inheritance - real world examples III - tracking

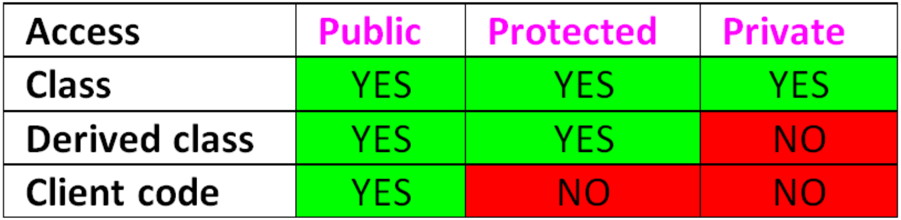




Access modifiers I

- Access modifiers define *from where* we can access the members of a class (fields or methods).
  - public: public members can be accessed from anywhere.
  - private: private members can be accessed from within the class or from friend functions or classes.
  - protected: protected members can be accessed from within the derived classes; protected acts just like private, except that inheriting classes have access to protected members, but not to private members. Friend functions or classes can access protected members.

Access modifiers II



Access	Public	Protected	Private
Class	YES	YES	YES
Derived class	YES	YES	NO
Client code	YES	NO	NO

Friend elements I

- A non-member function can access the private and protected members of a class if it is declared a friend of that class.
- Friend function: the declaration of this external function is placed within the class and it is preceded with the keyword friend.
- The friend keyword does not need to be used when defining the function.

Demo

Friend functions (friend_functions.cpp)

Friend elements II

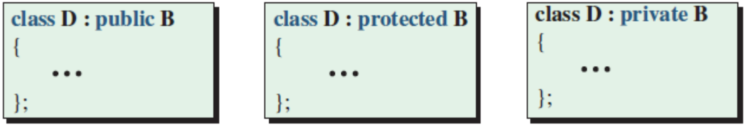
- A *friend* class is a class whose members have access to the private or protected members of another class.
- Friendship is not transitive: The friend of a friend is not considered a friend unless explicitly specified.
- Friendship is never corresponded (unless specified):
  Demo: Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle. (Rectangle members can access the protected and private members of Square, but not the other way around).

Demo

FriendClasses

Inheritance types I

- three choices of deriving a class from a base class in C++: *private inheritance*, *protected inheritance*, and *public inheritance*;
- the default type of inheritance is *private*



```
class D : public B
{
    ...
};
```

Public inheritance

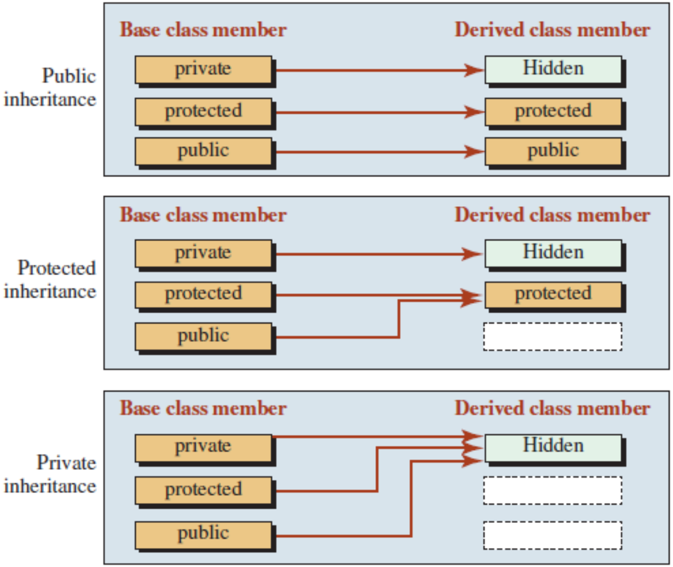
```
class D : protected B
{
    ...
};
```

Protected inheritance

```
class D : private B
{
    ...
};
```

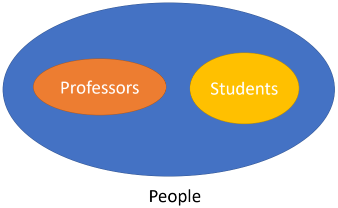
Private inheritance

Inheritance types II



Public inheritance - example I

- What characteristics/behaviors do people have in common?
  - name, ID, address
  - change address, display profile
- What things are special about students?
  - group number, classes taken, year
  - add a class taken, change course
- What things are special about professors?
  - course number, classes taught, rank (assistant, etc.)
  - add a class taught, promote



Public inheritance - example II

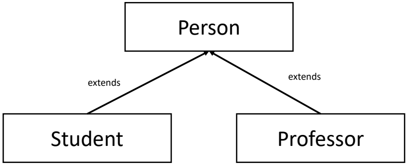
- A subtype inherits characteristics and behaviors of its base type.
- Each student has:

Characteristics:

- name
- ID
- address
- year
- classes
- group number

Behaviors:

- display profile
- change address
- add a class taken
- change group number



Methods overloading and methods overriding

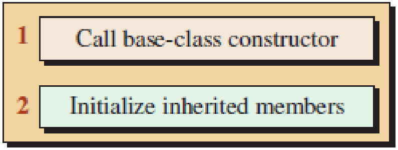
- A derived class may override (redefine) some methods of the base class.
- In defining derived classes, we only need to specify what is different about them from their base classes (programming by difference).
- Inheritance allows only overriding methods and adding new members and methods.
- We cannot remove functionality that was present in the base class.
- Use the scope resolution operator :: to access the overridden function of base class from derived class.

Methods not inherited I

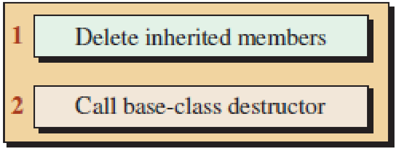
- member functions that are not inherited in the derived class: *default constructor*, *parameter constructor*, *copy constructor*, *destructor*, and *the assignment operator*;
- an object of a derived class naturally has more data members than a corresponding base class;
- the constructor of a derived class must construct more;
- the destructor of a derived class must destruct more.
- the constructor of the derived class cannot initialize the private data members of the base class;
- the destructor of a derived class cannot delete the private data members of the base class because they are hidden in the derived class.

Methods not inherited II

- solution → **INVOCATION**
- the constructor of the derived class **invokes** the constructor of the base class in its initialization and then initializes the data members of the derived class;
- the destructor of the derived class first deletes the data members of the derived class and then calls the destructor of the base class.
- destructors are called automatically *in the reverse order* of construction.



Constructors for derived class



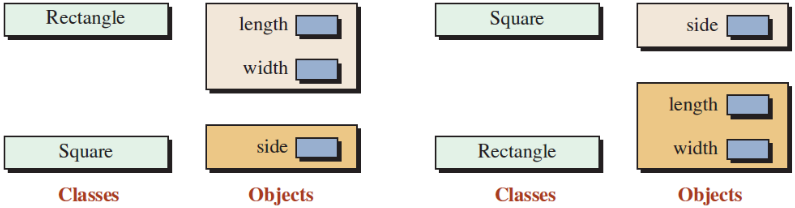
Destructors for derived class

Delegation of duty

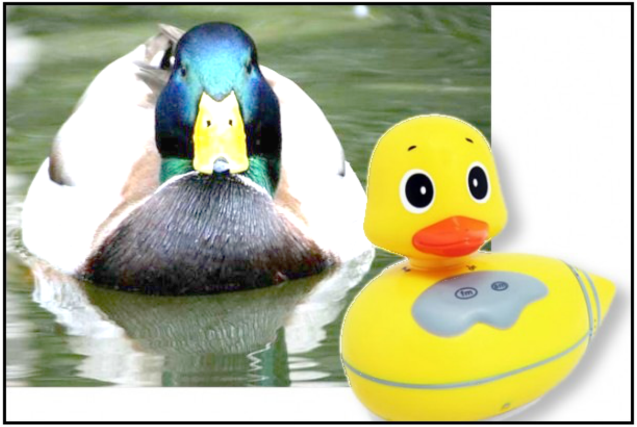
- An overloaded or overridden member function in a derived class can delegate part of its operation to a member function in a class in a higher level by calling the corresponding member function.
- In *delegation*, a derived member function delegates part of its duty to the base class using the class resolution operator (::)
- In *invocation*, the constructor of a derived class calls the constructor of the base class during initialization, which does not require the class resolution operator.

Liskov substitution principle I

- If S is a declared subtype of T, objects of type S should behave as objects of type T are expected to behave, if they are treated as objects of type T. *(Barbara H. Liskov and Jeannette M. Wing, A Behavioral Notion of Subtyping, ACM Transactions on Programming Languages and Systems, 1994.)*
- An object of the derived class (public inheritance) can be used in any context expecting an object of the base class (upcast is implicit).



Liskov substitution principle III

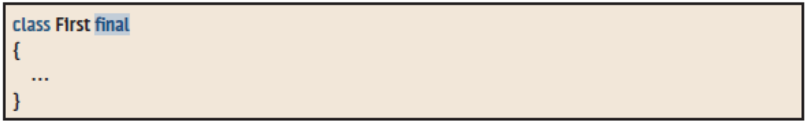


LISKOV SUBSTITUTION PRINCIPLE

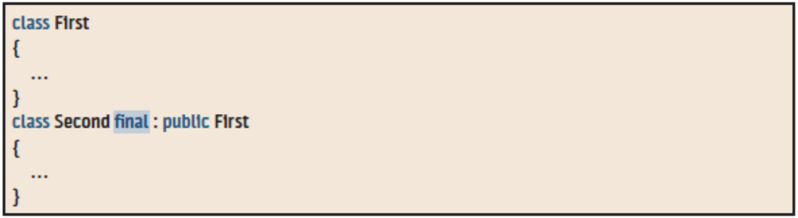
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Blocking inheritance - the final keyword

- block inheritance → we defined a class and do not want users to inherit from this class and create derived classes;
- use the final keyword



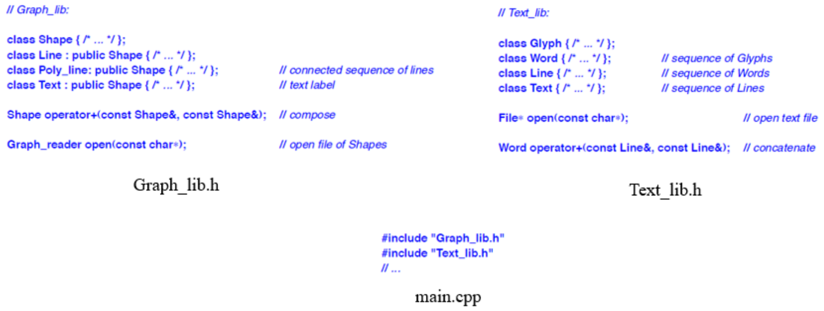
```
class First final
{
    ...
}
```



```
class First
{
    ...
}
class Second final : public First
{
    ...
}
```

What's new in C++ (cont'd) - namespaces I

- *modularity* - to keep separate things separate and to allow access to a "module" only through a well-specified interface.



```
// Graph_lib:

class Shape { /* ... */ };
class Line : public Shape { /* ... */ };
class Poly_line: public Shape { /* ... */ };        // connected sequence of lines
class Text : public Shape { /* ... */ };             // text label

Shape operator+(const Shape&, const Shape&);   // compose

Graph_reader open(const char*);                   // open file of Shapes
```

Graph_lib.h

```
// Text_lib:

class Glyph { /* ... */ };
class Word { /* ... */ };           // sequence of Glyphs
class Line { /* ... */ };           // sequence of Words
class Text { /* ... */ };           // sequence of Lines

File* open(const char*);                              // open text file

Word operator+(const Line&, const Line&);   // concatenate
```

Text_lib.h

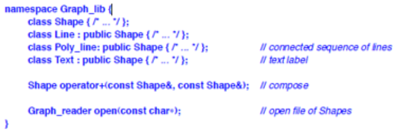
```
#include "Graph_lib.h"
#include "Text_lib.h"
// ...
```

main.cpp

- What is the problem with the previous code snippet?

What's new in C++ (cont'd) - namespaces II

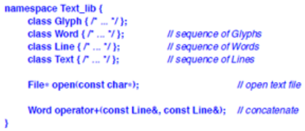
- Solution → namespaces



```
namespace Graph_lib {
    class Shape { /* ... */ };
    class Line : public Shape { /* ... */ };
    class Poly_line: public Shape { /* ... */ };        // connected sequence of lines
    class Text : public Shape { /* ... */ };            // text label

    Shape operator+(const Shape&, const Shape&);  // compose

    Graph_reader open(const char*);                     // open file of Shapes
}
```



```
namespace Text_lib {
    class Glyph { /* ... */ };
    class Word { /* ... */ };           // sequence of Glyphs
    class Line { /* ... */ };           // sequence of Words
    class Text { /* ... */ };           // sequence of Lines

    File* open(const char*);                        // open text file

    Word operator+(const Line&, const Line&);   // concatenate
}
```

What's new in C++ (cont'd) - namespaces III

- *namespace* is a (named) scope;
- a namespace is used to directly represent the notion of a set of facilities that directly belong together, for example, the code of a library;
- members of a namespace are in the same scope and can refer to each other without special notation, whereas access from outside the namespace requires explicit notation;
    - explicit qualification: e.g. std::string
    - using declarations: e.g. using std::string;
    - using directives: e.g. using namespace std; !!overuse can lead to exactly the name clashes that namespaces were introduced to avoid!!

What's new in C++ (cont'd) - namespaces III

- namespaces are open: you can add names to it from several separate namespace declarations;
- the members of a namespace need not be placed contiguously in a single file;

Demo

Namespaces (namespaces.cpp)

Summary - Inheritance

- Allows code to be reused between related types.
- Defines an IS A relationship.
- Constructors, assignment operators and destructors are not inherited.
- An object of the derived class (public inheritance) can be used in any context expecting an object of the base class (upcast is implicit), but not viceversa.
- Methods can be redefined (overriden) in derived classes.