Seminar 09
Inheritance

1. What is inheritance?

Source: http://www.cplusplus.com/

> ” *Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a base class and a derived class: The derived class inherits the members of the base class, on top of which it can add its own members.*

2. The *protected* modifier.

- Similar to public and private, the *protected* modifier is applied to member fields and member functions to a class. It's used when we don't want the outside world to know about a certain member, but the derived class should inherit it.

Access	**public**	**protected**	**private**
Inside the class	yes	yes	yes
Inside the derived class	yes	yes	no
The outside world	yes	no	no

2. What's being inherited?

- A publicly derived class inherits access to every member of a base class **except** for its private members, its friends, its constructors, its destructor and its assignment operators (operator=).

- *Note:* *Even though the constructors and the destructor of the base class are* ***not*** *inherited, they are automatically called just before the body of the derived class's constructors and destructor.*

- Unless otherwise specified, the constructors of the derived class call the **default** constructor of the base class. Specifying which constructor should be called can be done via the initializer list as follows:

```
Derived::Derived(param1, param2, …, paramN)
    : Base(paramP, …, paramQ) // Calling the base class's ctor
    , other_members
{
    // Body of the Derived class's constructor
}
```

3. Accessing the base class's members.

- Accessing the public and protected member **fields** of the base class can be done just by stating its identifier (name) - just as if it's a derived class member.

- Accessing the public and protected member functions (**methods**) of the base class can be done by stating the base class's name and then "::" as follows:

```
void Derived::foo()
{
    Base::bar(); // Calling the base class's method bar()
    // ... other code ...
}
```

4. Overloading vs overriding.

- **Overloading** a function means creating another function with the same name, but with *different* parameters. The same applies for methods and constructors.

- **Overriding** a member function (method) means creating a method in a *derived* class with the same name and parameters as in the base class. This effectively overrides the base class's implementation, and the derived class now uses this specific implementation instead of the inherited.

5. Casting to and from a publicly derived non-virtual class.

- Casting a derived class to a base class effectively uses narrowing conversion similar to converting a double to a float for example.

```
Derived dObj(...);
Base bObj = dObj;   // Copies only the base class's members
```

- Unless a conversion is defined, a base class cannot be casted to a derived class.

```
Base bObj(...);
```
~~`Derived dObj = bObj;`~~

- Casting a derived class pointer/address to a base class pointer can be done and the pointer will only have access to the base class's public members.

```
Derived dObj(...);
Base* bObj = &dObj;  // Doesn't create a new object, but
                          // bObj only has access to Base's members
```

6. Concepts.

Examples source: http://github.com/VasiPeycheva

- **Association** - our class **uses** another for example as a parameter of a method. The two objects **aren't linked** in any way.
  Example:
```
class Person {
    ...
    bool drive(const Vehicle& veh);
    ...
};
```

- **Aggregation** - our class is "**linked**" to an object of another, but isn't its owner. The two objects live **independently**.
  Example:
```
class Client {
public:
    Client(Vehicle* veh) : vehicle(veh) {}
    ...

private:
    Vehicle* vehicle;
};
```

- **Composition** - our class **owns** an object or an array of objects of another class. In this case our class is **responsible** for the object. Which means we need to take care of the memory - creating and deleting it when our object is created and destroyed.
  Example:
```
class Tree {
public:
    Tree(...) {
        fruits = new Fruit[...];
        ...
    }

    ~Tree() {
        delete[] fruits;
    }

private:
    Fruit* fruits;
};
```

7. Inheritance types.

- **Public** inheritance is used when there is a "**is-a**" relation between the derived class and the base class.

  Example: Every **Student** is a **Person**.

  ```
  class Student : public Person { ... }
  ```

- **Protected** inheritance, similarly to *composition*, is used when there is a "**has-a**" relation between the derived class and the base class.

  *Note: Very similar to private inheritance, but derived classes of the derived class **will** know about this inheritance.*

- **Private** inheritance is used when there is a "**is-implemented-in-terms-of**" relation between the derived class and the base class.

  *Note: Very similar to protected inheritance, but derived classes of the derived class **won't** know about this inheritance.*

  Example: Every **Car** can be implemented in terms of its **Engine**.

  ```
  class Car : private Engine {
  public:
      using Engine::start;  // inherit the Engine's start() method
      ...                   // publicly instead of privately
  };
  ```

The full example code can be found in the private-inheritance-example.cpp file.

- When should I prefer private inheritance over composition?

Source: Dinesh Khandelwal on Quora.com

" *Use composition whenever you can and use private inheritance whenever you must. When must you? Primarily when protected members and/or virtual functions enter the picture.*

Table of members' visibility conversion with different inheritance types.

Base class	Inheritance type	Derived class
public	: public →	public
protected		protected
private		-
public	: protected →	protected
protected		protected
private		-
public	: private →	private
protected		private
private		-