Inheritance (advanced topics)

- The general form of class inheritance:

class derived_class_name : *access_specifier* base_class_name
{
}

access_specifier: *public* *private* *protected*

- The access specifier defines how members of a base class are inherited from a derived class. If an access specifier is not specified, it defaults to a derived class declared with the *class* keyword as private; with the keyword *struct* - as public.

- *public*:
    - The public members of the base class become public members of the derived class (directly accessible inside and outside the derived class).
    - The private members of the base class remain the private members of the derived class. This means that access to these members is possible only through public function of the get_val () type, both in the derived class and outside the derived class.

- The protected members of the base class become protected members of the derived class - directly accessible members in the derived class, yet they remain inaccessible outside in the derived class and in the base class.

- *private*:

  - The public members of the base class become the private members of the derived class. Inside the derived class, direct access to these members exists, outside - it does not exist. A derived class can expose the public members of the base class through its get_derived () public methods or restore it again as public (see slide 4).

  - The private members of the base class remain the private members of the derived class. A derived class does not have direct access to these members (is possible an access only through base class public methods), outside access to hidden base class members through base class methods is impossible (the base class methods become hidden members of the derived class).

  - The protected members of the base class become private members of the derived class - they are available directly in the derived class and are not accessible outside the derived class.

- *protected*:
  - The public members of the base class become protected members of the derived class.
  - The private members of the base class remain the private members of the derived class.
  - The protected members of the base class remain the protected members of the derived class.
- Reversal of the public components of the base class when inherited as private:

```
class B
{
        int i;
public:
        int j;
.....................................
};

class D : private B
{
public:
        B::j;     //restoration of the public member of the base class
                  //to the status of public when inherited as private.
        B::i;     //error! i - private member of the base class
};

int main()
{
    D ob;
    ob.j = 20; //OK
    return 0;
}
```

The access specifier for a member of the base class	The derived class is inherited as		
	: public	: private	: protected
public: (It is accessible outside the base class - ob.i =… // OK)	**inside the derived class**: is accessible	**inside the derived class**: is accessible	**inside the derived class**: is accessible
	outside the derived class: is accessible	**outside the derived class:** is inaccessible	**outside the derived class:** is inaccessible
private: (It is inaccessible outside the base class - ob.i = … // !OK)	**inside the derived class**: is inaccessible	**inside the derived class**: is inaccessible	**inside the derived class**: is inaccessible
	outside the derived class: is inaccessible	**outside the derived class:** is inaccessible	**outside the derived class:** is inaccessible
protected: (It is inaccessible outside the base class - ob.i = … // !OK)	**inside the derived class**: is accessible	**inside the derived class**: is accessible	**inside the derived class**: is accessible
	outside the derived class: is inaccessible	**outside the derived class:** is inaccessible	**outside the derived class:** is inaccessible

➢ The specifiers that determine how a base class is inherited by a derived class do not affect how the members of the base class are accessed in the derived class. Access to base class members in a derived class is determined only by what access specifier those members are declared in the derived class.

Constructors, destructors and inheritance

➢ If the base class and the derived class have constructors and destructors, the order of calls is as follow:

- Base class constructor
- Derived class constructor
- Derived class destructor
- Base class destructor

➢ The base class knows nothing about the existence of the derived class - initialization in the base class is performed independently of the derived class. Base class initialization can be the basis for derived class initialization. For this, the constructor of the base class is called before the constructor of the derived class.

- On destroying objects, destroying the base class would damage the derived class object. For this, the destructor of the derived class is to be called earlier than the destructor of the base class.

- Passing arguments to the constructor of the base class.

    - All arguments of the derived class, as well as the base class, are passed to the derived class constructor.

    - The base class arguments remain passed to the base class.

derived_class_constructor(full_argument_list) : *base_class_constructor*(argument_list_to_base_class)
{
    //body of constructor of derived class
}

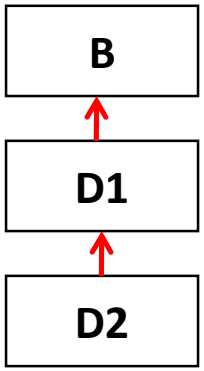
full_argument_list – the argument list of the derived class and the base class.
argument_list_to_base_class – the base class argument list.

- **Example 37.**

- Example:

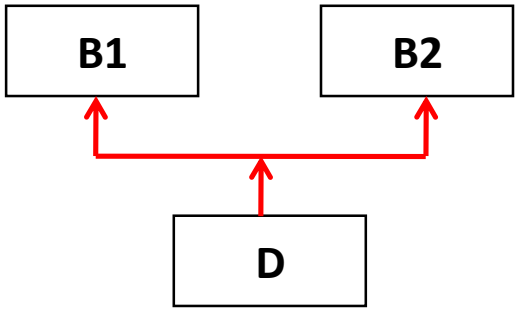
```
class base
{
        int i;
public:
        base(int ii) { i = ii;}
};

class derived : public base
{
        int j;
public:
        derived(int ii) : base(ii) { j=0; }    //passing an argument to the base class
                                                //derived class does not require the arguments
};
```

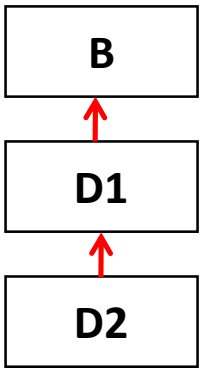
Multi-inheritance



Multilevel hierarchy of classes

The derived class inherits several base classes



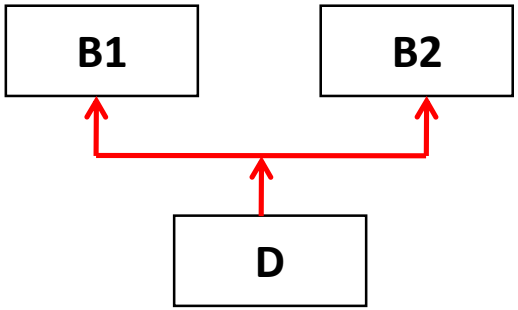


➢ In a multi-level class hierarchy, constructors are called in the order in which these classes inherit, and destructors - in the reverse order.

- constructor B
- constructor D1
- constructor D2
- destructor D2
- destructor D1
- destructor B

Definitions of the derived classes:

```
class D1 : public B
 {
 };

 class D2 : public D1
 {
 };
```



➢ If a derived class inherits several base classes, the derived class declaration is a follow:

```
class D : public B1, public B2
{
};
```

```
class derived_class_name: access specifier  base_class_name_1,
                          access specifier  base_class_name_2,
                          ……..……..……..……..……..……..……..……..……..……..
                          access specifier  base_class_name_N
{
};
```

➢ Constructors are called in the order in which the classes were created, in order from left to right, which is as the base classes were listed in the inheritance list. Destructors are called in reverse order, right to left:

```
constructor B1
constructor B2
constructor D
destructor D
destructor B2
destructor B1
```

➢ In that case, the constructors and destructors will be called like this:

```
class  D : private B2 , private B1
{
}

constructor B2
constructor B1
constructor D
destructor D
destructor B1
destructor B2
```

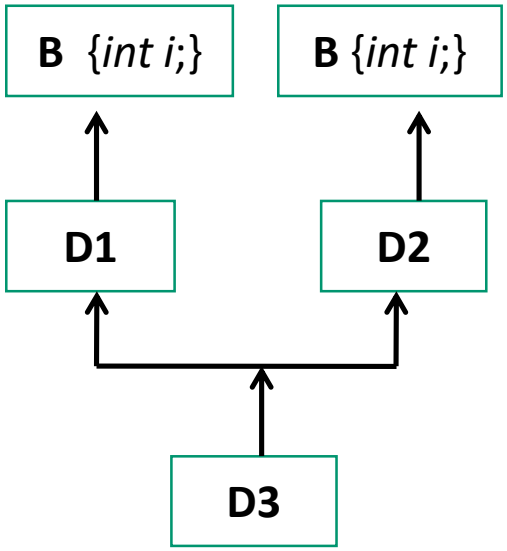
- Passing arguments to constructors of base classes.

```
derived_class_constructor(list_argum_full) : base_class_name_1(list_argum_1) ,
                                             base_class_name_2(list_argum_2) ,
                                             ……,
                                             base_class_name_N(list_argum_N)
{
      // derived class constructor
}
```

list_argum_full – the general argument list for the derived class and all base classes.
list_argum_1 – the argument list for base class 1,
list_argum_2 – the argument list for base class 2,
…….
list_argum_N – the argument list for base class N

Example W38, W39

- We will consider such a situation:



➢ There is ambiguity in this class hierarchy. Let class B have an *int i* member. Then classes D1, D2 (derived from class B, so each of these classes contains its own copy of the base class) will have two different instances of the member *i*, which are at different memory addresses. The D3 class inherits from D1, D2. So in class D3 there are 2 instances of component *i*. Which instance of the component *i* should we taken?

➢ Virtual classes prevent the creation of two instances of variable *int i* in this situation:

```
class D1 : virtual public B
{
}

class D2 : virtual public B
{
}

class D3 : public D1, public D2
{
}
```

Example W40.

- If the base class B has been inherited by D1, D2 as a virtual class, a single copy of this base class is created inside the derived class.

- The difference between normal and virtual class occurs when an object inherits from the base class more than once.

- **Przykład W41 (from Schildt)) !**