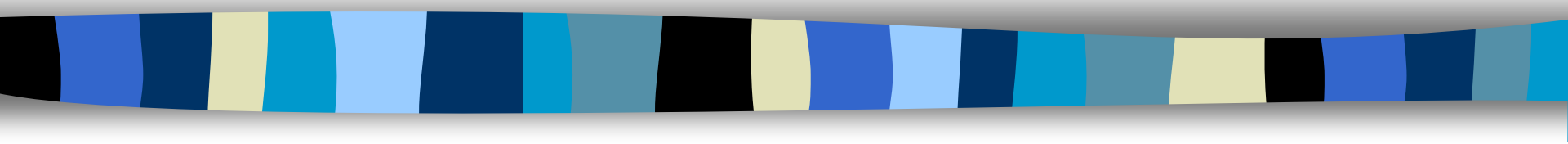
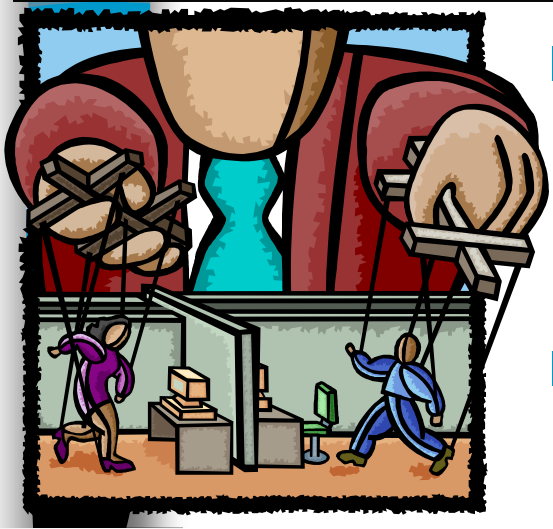
Inheritance



Miri Kopel Ben-Nissan

```
class Employee
{
private:
   char* m_sFirstName, m_sLastName;
   Date  m_hiringDate;
   short m_siDepartment;
   //…
public:
   //…
   void SetHiringDate(int dd,int mm,int yy);
   void SetDepartment(short department);
   //…
};
```



- **How should we describe a Manager?**
- **A manager is also an employee.**
- **Has more attributes than Employee.**

How should we describe the Manager?

- **Copy the attributes and methods from Employee and add the extra information.**
  - Inefficient: doubling of code.
- **Add an Employee object to the Manager's attributes (Composition):**

```
class Manager
{
private:
    //manager's employee information.
    Employee     m_Emp;
    //people managed.
    Employee*    m_employeeGroup;
    short        m_siLevel;
    //...
};
```

But…

- **We need to add set & get methods to the `Manager` in order to access its `Employee` data.**

- **There is nothing that tells the compiler and other tools that `Manager` is also an `Employee`.**
  - A **`Manager*`** is not an **`Employee*`**.
  - We cannot put a **`Manager`** onto a list of **`Employees`** without writing special code.

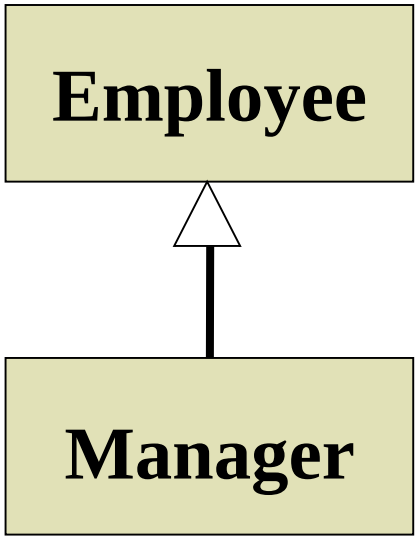
Solution: Inheritance

- The correct approach is to explicitly state that a `Manager` is an `Employee`, with a few pieces of information added.
- When you use inheritance, you are saying, "This new class is like that old class".

```
class Manager : public Employee
```

- When you do this, you automatically get all the data members and member functions in the base class.

```
class Manager : public Employee
{
private:
   Employee* m_employeeGroup;
   short     m_siLevel;
//...
};
```



Base class
(super class)

Derived class
(subclass)

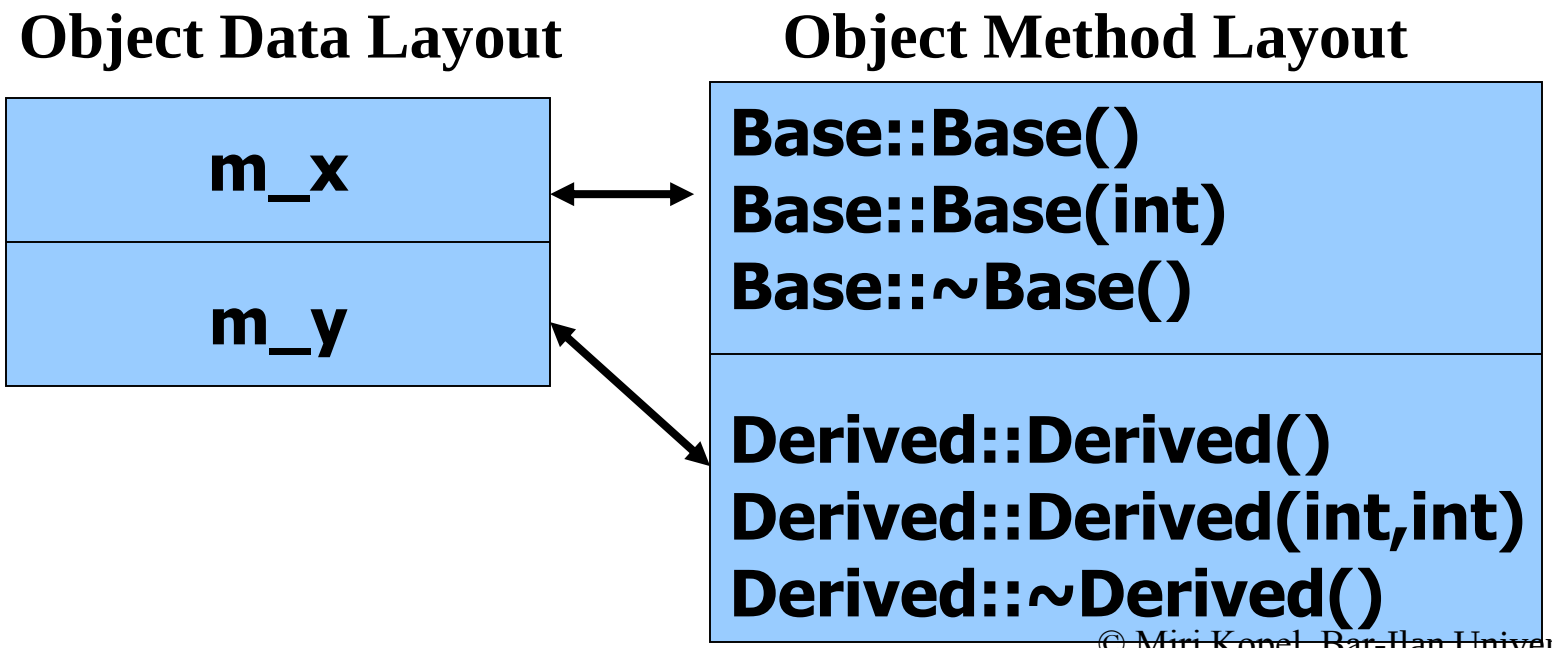
- Deriving `Manager` from `Employee` in this way makes `Manager` a subtype of `Employee` so that a `Manager` can be used wherever an `Employee` is acceptable.

How does the derived object look like?

- **Inheritance results in a "composite" object at run-time.**
  - Public methods are combined into one set of public methods, but with distinct names qualified by the class name in which they are defined.
  - Data layout is combined as well, with compile-time access control (e.g., private) still applies.

class Base
int m_x;
Base(); Base(int x); ~Base(); //...

class Derived : public Base
int m_y;
Derived(); Derived(int x, int y); ~Derived(); //...



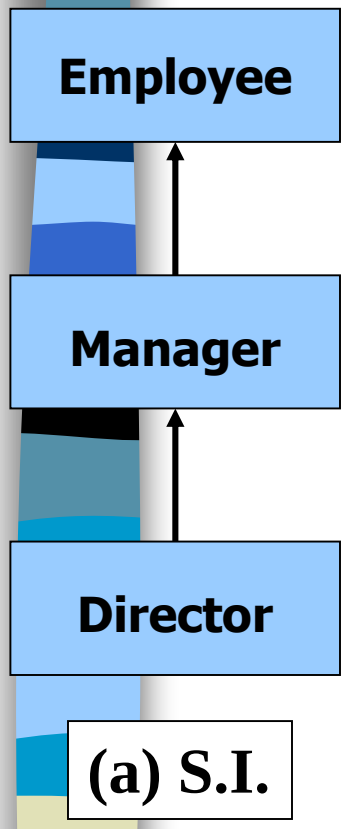
Constructors and Destructors

- **Class objects are constructed from the bottom up:**
  1. The base.
  2. The members.
  3. The derived class itself.
- **They are destroyed in the opposite order:**
  1. The derived class itself.
  2. The members.
  3. The base.

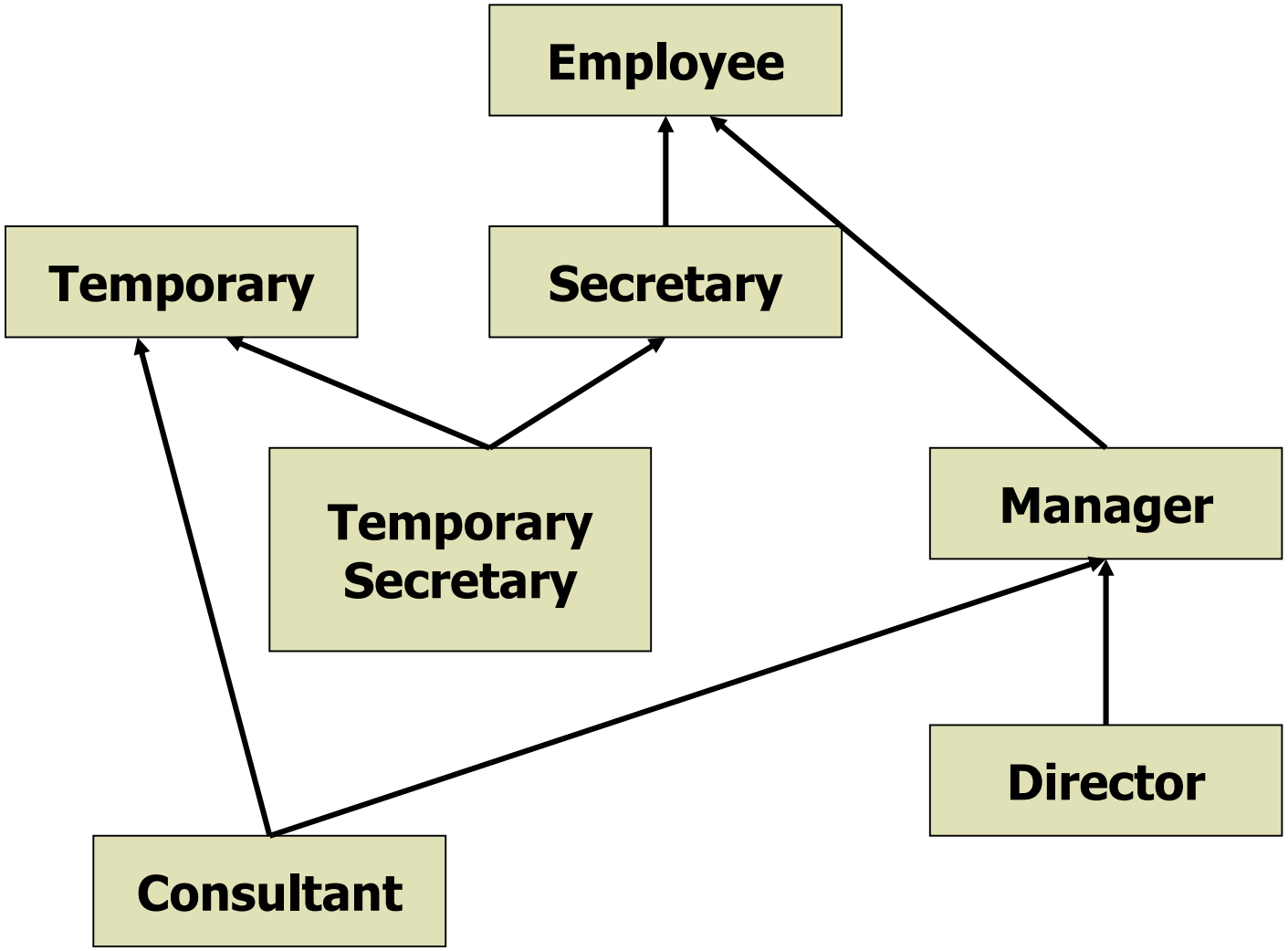
Inheritance comes in two forms, depending on number of parents a subclass (derived class) has:

- **Single Inheritance (SI)**
  - Only one parent per derived class.
  - Form an inheritance tree.
  - SI requires a small amount of run-time overhead when used with dynamic binding.
- **Multiple Inheritance (MI)**
  - More than one parent per derived class.
  - Forms an inherited DAG (Directed Acyclic Graph).
  - Compared with SI, MI adds additional run-time overhead (also involving dynamic binding).

- A derived class can itself be a base class.



(a) S.I.



(b) M.I.

Types of Inheritance

- **<u>Subtyping/Interface Inheritance:</u> creating a subtype of an existing class for purpose of setting up dynamic binding.**
  - **Circle** is a subclass of **Shape** (i.e., Is-A relation).
  - A **Birthday** is a subclass of **Date**.

- **Code Reuse /Implementation Inheritance:**
  reusing an implementation to create a new class type.
  - Class **Clock** that inherits from class **Window** the ability to represent itself graphically. A stack is not really a subtype or specialization of Vector.
  - In this case, inheritance makes implementation easier, because there is no rewrite and debug existing code.
  - This is called using inheritance for reuse, i.e., a pseudo-Has-A relation.

Member Functions

- A member of a derived class can use the public (and protected) members of its base class as if they were declared in the derived class itself.
- However, a derived class cannot use the base class' private names.
- A protected member is like a public member to a member of a derived class, yet it is like a private member to other functions.

Member Functions (cont…)

- **You can inherit a method as is and call it as though it were one of your own methods.**
- **You can overload a method with a new implementation.**
- **You can extend them with a method of the same name that calls the inherited method at some point.**
- **Private methods of the base class are not accessible to a derived class.**
  - unless the derived class is a friend of the base class.

Functions you don't inherit:

- constructors and destructors don't inherit and must be created specially for each derived class.
  - However, they may be called automatically where necessary.
- The operator = doesn't inherited because it performs a constructor-like activity.
- These functions are synthesized by the compiler if you don't create them yourself.

Upcasting

- **The most important aspect of inheritance is not that it provides member functions for the new class, but the relationship expressed between the new class and the base class.**

- **This relationship can be summarized by saying, "*This new class is a type of the existing class*".**

- **This description is supported directly by the compiler.**

Upcasting (cont…)

- **Inheritance means that all the functions in the base class are also available in the derived class.**
- **Thus, any message you can send to the base class can also be sent to the derived class.**
- **A pointer to a derived class may always be assigned to a pointer to a base class that was inherited publically (but not vice versa).**

- For example, we can now create a list of Employees, some of whom are Managers:

```
void f(Employee& emp , Manager& mng)
{
    Employee* empArray = new Employee[2];
    empArray[0] = &emp;
    empArray[1] = &mng;
}
```

- Upcasting is safe because you're going from a more specific type to a more general type – the only thing that can occur to the class interface is that it can lose member functions, not gain them.

Inheritance or Composition ?

- **In C++ you reuse code by creating new classes, but instead of creating them from scratch, you use existing classes that someone else has built and debugged.**
  - Composition.
  - Inheritance.

- **<u>Composition</u> is generally used when you want the feature of an existing class inside your new class, but not its interface.**
  - you embed an object to implement features you've defined rather than the interface from the original class.
- **<u>Inheritance</u> is used when you're making a new type from an existing one, and you want your new type to have exactly the same interface as the existing type (plus other member functions you want to add), so you can use it everywhere you'd use the existing type.**

- One of the ways to determine whether you should be using composition or inheritance is by asking whether you'll ever need to upcast from your new class.

- We say an object X has-a Y if Y is a part-of X (composition).

- We say an object X is-a Y, if everywhere you can use an object of type Y you can use instead of object of type X (inheritance).

For example:

- Would you say that an **Email Message** is-a **Header** or an **Email Message** has-a **Header**?
- Would you say that a **Stack** is-a **Vector** or a **Stack** has-a **Vector**?
- Would you say that **Circle** is-a **Shape** or a **Circle** has-a **Shape**?

Private Inheritance

- **When you inherit privately you're creating a new class that has the data and functionality of the base class, but the functionality is hidden, so it's only part of the underlying implementation.**
  - When you inherit privately, all the public members of the base class become private.
  - The class user has no access to the underlying functionality.
  - An object cannot be treated as a instance of the base class. It is illegal to assign the address of a derived object to a pointer to a base object.

Private inheritance or Composition?

- **You'll usually want to use composition rather than private inheritance.**
- **Use private inheritance when:**
  - you want to produce part of the same interface as the base class and disallow the treatment of the object as if it were a base-class object.
  - In implementation inheritance.

Derived Class Access to Base Class Members

Base Class Access Control	Inheritance mode		
	public	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	none	none	none

Inheritance and Assignment operator

- Consider we have class Base and class Derived (that derived publically from Base). Consider we declare two objects of Derived:

```
Derived d1, d2;
```

And now we want to do:

```
d1 = d2;
```

- If the derived class didn't implemented the operator=, than the Base's operator= is called.
- If the derived class implemented the operator=, than only this operator is called.

- **If we want also to activate the Base's assignment operator, we need to call it explicitly:**

```
class Derived : public Base
{
  //...
public:
  //...
  Derived& operator=(const Derived& d)
  {
   //call the Base version of operator =:
   Base::operator=(d);
   //do the assignment concerning the
   //Derived's attributes.
  }
};
```

The danger of Implementation Inheritance:

- **Using inheritance for reuse may sometimes be a dangerous misuse of the technique.**
  - Operations that are valid for the base type may not appeal to the derived type at all.
    - e.g., Stack that inherits class Vector: the operator[] is valid for the Vector, not to the Stack.
  - Thus, you can use **private** base class or use composition instead.

Liabilities of Multiple Inheritance:

- **A base class may legally appear only once in a derivation list.**
  - e.g., `class A : public B, public B` is an error.

- **A base class may appear multiple times within a derivation hierarchy.**
  - Can cause method and data member ambiguity.
  - Causes unnecessary duplication of storage.

Function name hiding

- When you inherit a class and provide a new definition for one of its member functions, If you change the member function argument list or return type in the derived class, all the other versions are automatically hidden in the new class.

Function name overriding

- When you inherit a class and provide a new definition for one of its member functions, If you provide the exact signature and return type in the derived class definition as in the base class definition, we call it “redefinition” or “overriding”.

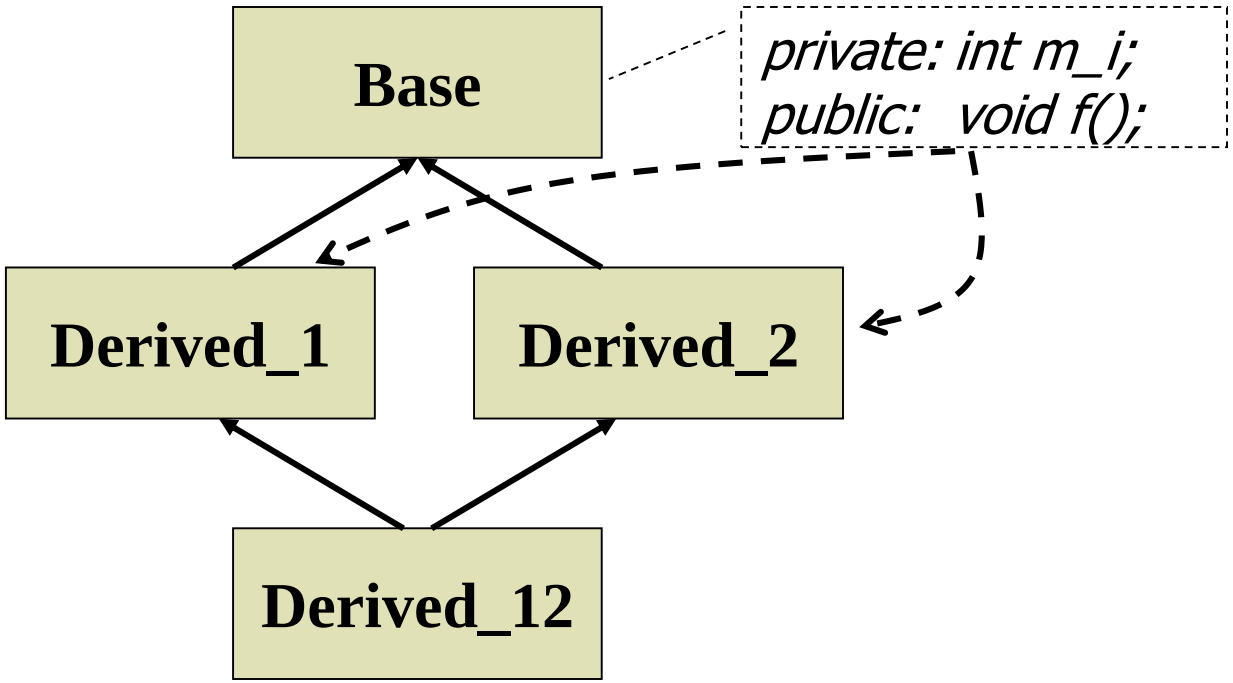
Example of hiding names:

```
class A
{
public:
   void f(int i) {cout<<"A::f(int)";}
   float* g() {return nullptr;}
};

class B : public A
{
public:
   void f(int i, int j) {cout<<"B::f(int,int)";}
   int*  g() {return nullptr;}
};

int main()
{
   B b;
   b.f(2); //error: function does not take 1 parameters
   float* pF = b.g(); //error: can't convert int* to float*
}
```

Virtual Inheritance



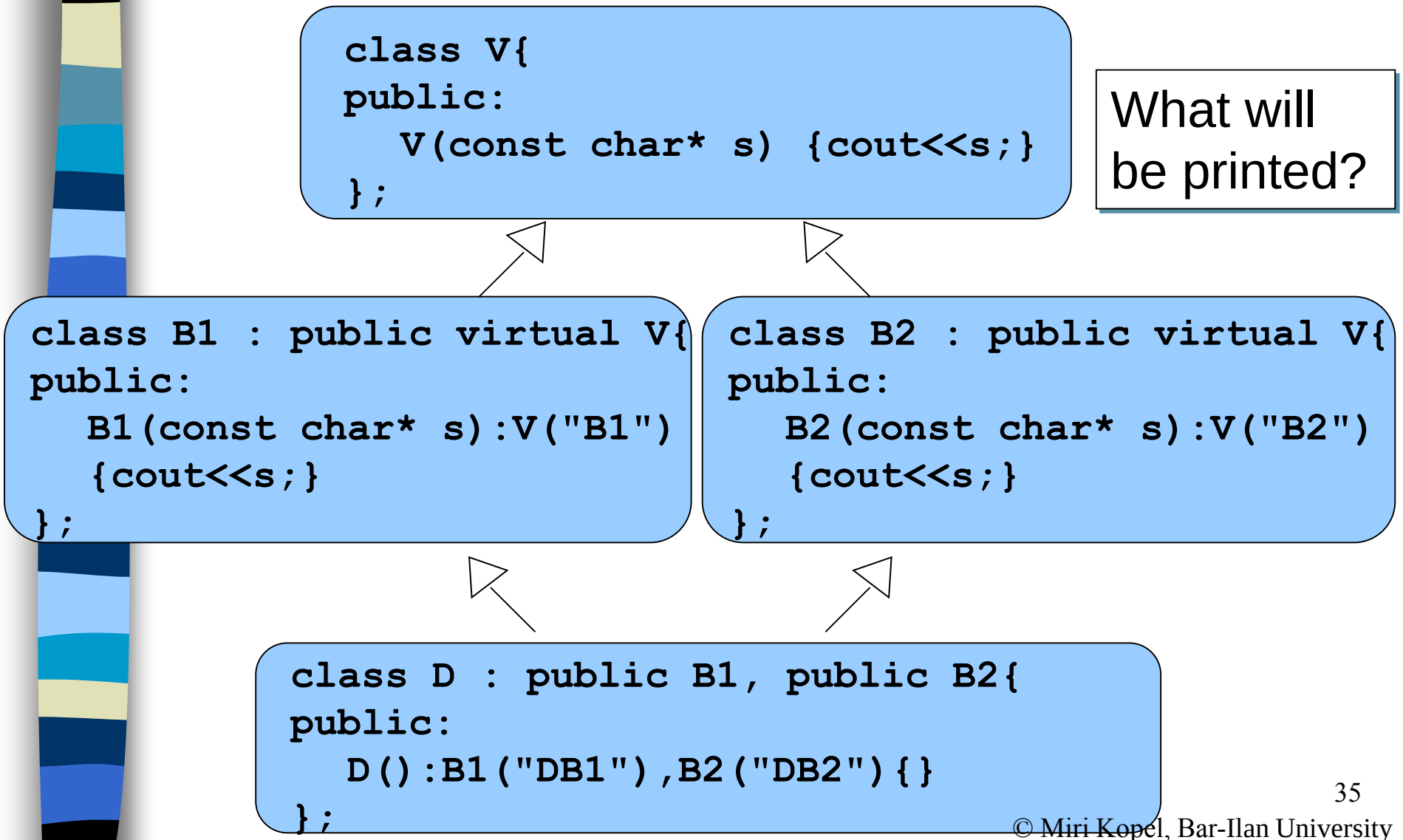
- **With virtual inheritance there is only one copy of each object even if (because of multiple inheritance) the object appears more than once in the hierarchy.**

```
class Derived_12 : virtual public Derived_1,
                   virtual public Derived_2
{...};
```

```
class base{
  public: int i;
};
class derived1a: virtual public base {
  public: int j;
};
class derived1b: virtual public base {
  public: int k;
};
class derived2:  public derived1a,
                 public derived1b
{
  public: int product() {return i*j*k;}
};
int main()
{
  derived2 obj;
  obj.i = 10; obj.j = 3; obj.k = 5;
  cout<<"product is"<< obj.product() <<'\n';
}
```

constructors and Virtual inheritance

- Consider the following situation:



- **Answer: Compilation Error.**

- **Solutions:**
  - **Declare a default constructor in V.**
  - **Call V's constructor directly from D's:**
    - **Virtual bases are initialized by the initializer of the most derived class, other initializers are ignored.**
    - **Virtual bases may be initialized in a constructor of a derived class even if they are not immediate base classes.**