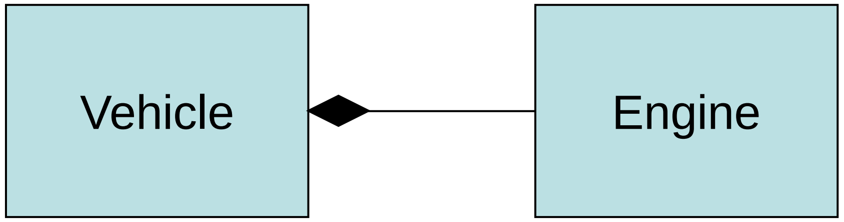
Object-Oriented Programming

Lesson 6

Inheritance and composition
Access control
Polymorphism
Virtual functions

Composition

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. In composition the new class is composed of objects of existing classes.



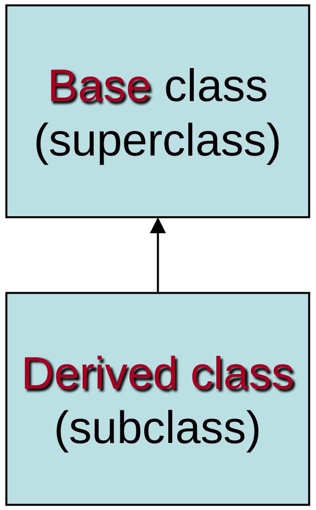
Composition is often referred to as a "has-a" relationship, as in "a car has an engine."

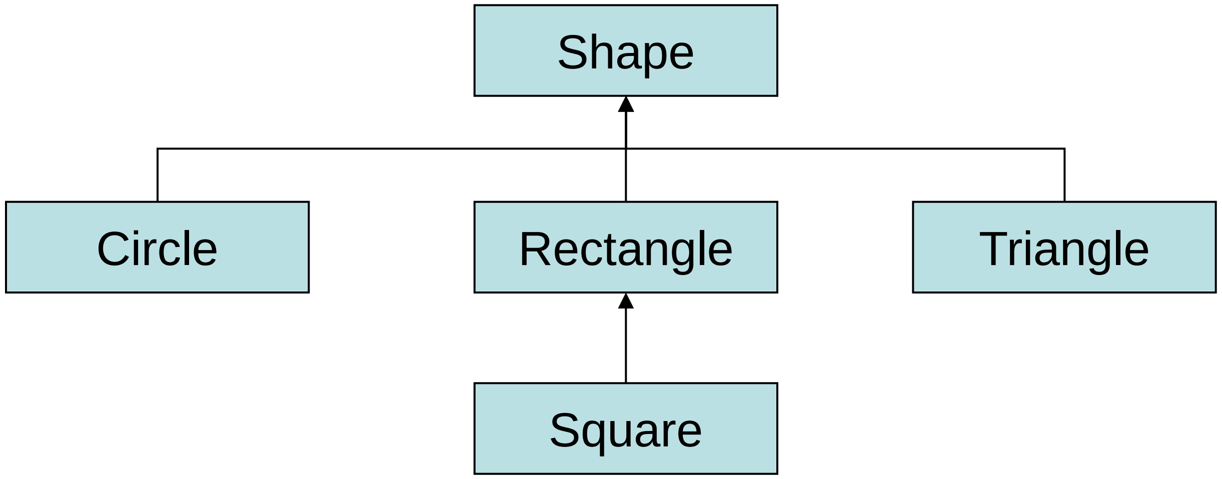
```
class Engine
{
   // definition of the engine
};

class Vehicle
{
   Engine engine;
   // definition of the vehicle
};
```

The Vehicle class only has access to the public members of Engine.

Inheritance

Inheritance permits new classes to be derived from the existing classes, reusing their interface and adding new functionality.





```
class Shape
{       //definition of the class
};

class Triangle : public Shape
{       //definition of the class
};
```

"is-a" relationship, because you can say "a triangle is a (specific) shape."

Inheritance

When you inherit from an existing type, you create a new type.

The new class has all the members of the old class (although the **private** ones are hidden away and inaccessible), and has the same interface.

All the messages you can send to objects of the base class you can also send to objects of the derived class.

You have two ways to differentiate your new derived class from the original base class:

1. Add brand new functions to the derived class.
2. Change the behaviour of an existing base-class function. This is referred to as overriding that function.

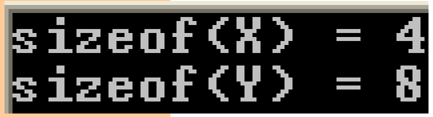
Inheritance

```
class X
{       int i;
public:
        X() { i = 0; }

        void set(int ii)
          { i = ii; }

        int read() const
          { return i; }
};
```

```
class Y : public X
{
        int i;
public:
        Y() { i = 0; }

        void set(int ii)
        {
            i = ii;
            X::set(ii);
        }
};
```

overriding

```
int main()
{       cout << "sizeof(X) = " << sizeof(X) << endl;
        cout << "sizeof(Y) = " << sizeof Y << endl;

        Y D;

        D.read();

        D.set(12);
}
```



```
sizeof(X) = 4
sizeof(Y) = 8
```

Constructor initializer list

The constructor initializer list occurs only in the definition of the constructor and is a list of “constructor calls” that occur after the function argument list and a colon, but before the opening brace of the constructor body.

This is to remind you that the initialization in the list occurs before any of the main constructor code is executed.

When an object is created, the compiler guarantees that constructors for all of its subobjects are called.

Constructor initializer list

```
class A
{
public:
        A ()  { b = CBook (“Test”, 2010); }
private:
        CBook b;
        int n;
};
```

The constructor initializer list is empty =>

1) A default CBook constructor will be called to initialize the subobject b;
2) A temporary object will be constructed with the call of `CBook (“Test”, 2010);`
3) The operator = will assign the temporary object to b;
4) The temporary object will be destroyed.

```
A () : b (“Test”, 2010), n (10)  { };
```

Here, the object b is initialized with just one constructor call!

Constructor initializer list

```
class A
{
public:
        A () : b (“Test”, 2010), n (10)  { };
private:
        CBook b;
        int n;
};
```

Pseudo-constructor

If you do not call the pseudo-constructor explicitly, no initialization will be done.

```
int i (3);
int* ip = new int (12);
```

Initialization of the base class

The constructor initializer list should contain a call to the base-class constructor.

```
class CPerson
{
       std::string m_sName;
       unsigned m_uAge;
public:
       CPerson(std::string name, unsigned age);
};
```

```
class CStudent : public CPerson
{       int m_iMecNum;
public:
       CStudent(std::string name, unsigned age, int num)
      : CPerson (name, age) { m_iMecNum = num; };
};
```

Constructors and destructors

Construction starts at the very root of the class hierarchy, and at each level the base class constructor is called first, followed by the member object constructors.

```
CStudent ("Ana Lopes", 23, 12345);
```

1. Constructor of `CPerson`
2. Constructor of `CStudent`

The destructors are called in exactly the reverse order.

1. Destructor of `CStudent`
2. Destructor of `CPerson`

Function overloading

Anytime you redefine an overloaded function name from the base class, all the other versions are automatically hidden in the new class.

```
class Base
{
public:
  int f();
  int f(int flag);
};

class Derived1 : public Base
{
};

class Derived2 : public Base
{
public:
  int f();
};
```

```
int main()
{
  Derived1 d1;

  d1.f();
  d1.f(0);

  Derived2 d2;
  d2.f();
  //  d2.f(0); error
}
```

Functions that don’t automatically inherit

Not all functions are automatically inherited from the base class into the derived class. Functions that don’t automatically inherit:

1. Constructors
2. Destructors
3. Operator =

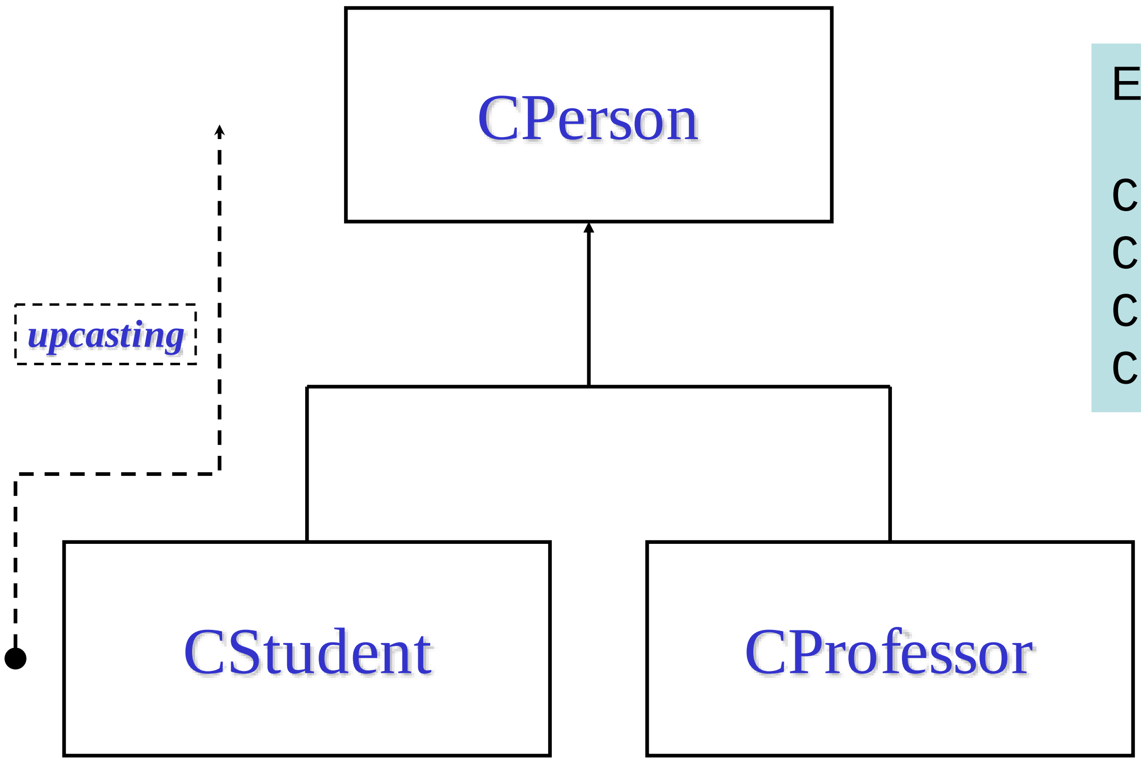
If the derived class does not have any constructor, the compiler will synthesize the default constructor.

The synthesized constructor will call the default constructor of the base class.

If the derived class does not have the destructor, the compiler will synthesize one.

Upcasting

Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type is called upcasting because of the way inheritance trees are drawn with the base class at the top.



Example:

```
CStudent s;
CProfessor p;
CPerson* ps = &s;
CPerson& rp = p;
```

Access to the base class members

The base class can be declared as either private, protected or public, for example:

```
class DERIVED_PUBLIC : public BASE {};
class DERIVED_PROTECTED : protected BASE {};
class DERIVED_PRIVATE : private BASE {};
```

When you’re inheriting, the base class defaults to private.

The access specifier controls:

1) access to the members of the base class;
2) pointer and reference upcasting.

Public inheritance

In public inheritance:
- public members of the base class remain public members in the derived class;
- protected members of the base class remain protected members in the derived class;
- private members of the base class are inaccessible in the derived class.

If a class B is the public base of the class D, then public members of B can be used by anyone and protected members of B can be used by members and friends of D.

Anyone can upcast D* to B*.

Public inheritance

```
class c1
{ public:
        int pub_1;
protected:
        int prot_1;
private:
        int priv_1;
};
```

```
class c2 : public c1
{ public:
        int pub_2;
        c2() { pub_1; prot_1; priv_1; }
protected:
        int prot_2;
private:
        int priv_2;
};
```

```
c1 a;   a.pub_1; a.prot_1; a.priv_1;
c2 b;   b.pub_1; b.pub_2;  b.prot_2; b.priv_2;
c1* p = new c2;
```

Protected inheritance

In protected inheritance:
- public members of the base class transform into protected members in the derived class;
- protected members of the base class remain protected members in the derived class;
- private members of the base class are inaccessible in the derived class.

If a class B is the protected base of the class D, then public and protected members of B can only be used by members and friends of D.

Only members of D can upcast D* to B*.

Protected inheritance

```
class c1
{ public:
        int pub_1;
protected:
        int prot_1;
private:
        int priv_1;
};
```

```
class c2 : protected c1
{ public:
        int pub_2;
        c2() { pub_1; prot_1; priv_1;
                c1* p = new c2; }
protected:
        int prot_2;
private:
        int priv_2;    };
```

```
c1 a;   a.pub_1;  a.prot_1; a.priv_1;
c2 b;   b.pub_1;  b.pub_2;  b.prot_2; b.priv_2;
c1* p = new c2;
```

Private inheritance

In private inheritance:
- public and protected members of the base class transform into private members in the derived class;
- private members of the base class are inaccessible in the derived class.

If a class B is the private base of the class D, then public and protected members of B can only be used by members and friends of D.

Only members of D can upcast D* to B*.

Private inheritance

```
class c1
{ public:
        int pub_1;
protected:
        int prot_1;
private:
        int priv_1;
};
```

```
class c2 : private c1
{ public:
        int pub_2;
        c2() { pub_1; prot_1; priv_1;
               c1* p = new c2; }
protected:
        int prot_2;
private:
        int priv_2;   };
```

```
c1 a;   a.pub_1;  a.prot_1;  a.priv_1;
c2 b;   b.pub_1;  b.pub_2;   b.prot_2;  b.priv_2;
c1* p = new c2;
```

Choosing the type of inheritance

Normally, you'll make the inheritance public so the interface of the base class is also the interface of the derived class. Public derivation means "*is-a*" for derived classes and friends. (list ↔ sorted list)

When you inherit privately, you're "*implementing in terms of*;" that is, you're creating a new class that has all of the data and functionality of the base class, but that functionality is hidden, so it's only part of the underlying implementation. The class user has no access to the underlying functionality, and an object cannot be treated as a instance of the base class. (list ↔ stack)

Protected derivation means "*implemented-in-terms-of*" to other classes but "*is-a*" for derived classes and friends. It's something you don't use very often, but it's in the language for completeness.

Function call binding

```
class CPerson
{
    public:
            void print () const
            {
                cout << “I am a person!" << endl;
            }
};
```

```
class CStudent : public CPerson
{
    public:
            void print () const
            {
                cout << “I am a student!" << endl;
            }
};
```

```
int main (int argc, char* argv[])
{
        CStudent s1;
        CPerson& r = s1;
        r.print();

        return 0;
}
```

```
I am a person!
```

?

Connecting a function call to a function body is called binding.

early
(before the program is run)

late
(at runtime)

Virtual functions

To cause late binding to occur for a particular function, C++ requires that you use the virtual keyword when declaring the function in the base class.

Late binding occurs only with virtual functions, and only when you're using an address of the base class where those virtual functions exist.

person.h

```
class CPerson
{
    public:
            virtual void print () const;

};
```

person.cpp

```
void CPerson::print() const
{
    cout << “I am a person!" << endl;
};
```

If a function is declared as virtual in the base class, it is virtual in all the derived classes. The redefinition of a virtual function in a derived class is usually called overriding.

Virtual functions

person.h
```
class CPerson
{
    public:
            virtual void print () const;

};
```

student.h
```
class CStudent : public CPerson
{
    public:
            void print () const;

};
```

person.cpp
```
void CPerson::print() const
{
    cout << “I am a person!" << endl;
};
```

student.cpp
```
void CStudent::print() const
{
    cout << “I am a student!" << endl;
};
```

```
int main (int argc, char* argv[])
{
        CStudent a1;
        CPerson& r = s1;
        r.print();

        return 0;
}
```

```
I am a student!
```

Extensibility

With **print()** defined as virtual in the base class, you can add as many new types as you want without changing the code that calls **print()** for objects of different types.

post_gr.h

```
class CPostGradStudent : public CStudent
{
    public:
            void print () const;

};
```

post_gr.cpp

```
void CPostGradStudent::print() const
{
    cout << “I am a postgraduate"
             “ student!" << endl;
};
```

```
int main (int argc, char* argv[])
{
        CPostGradStudent a2;
        CPerson& r = a2;
        r.print();
        return 0;
}
```

```
I am a postgraduate student!
```

Such a program is extensible because you can add new functionality by inheriting new data types from the common base class.

The functions that manipulate the base class interface will not need to be changed at all to accommodate the new classes.

post_gr.h

```
class CPostGradStudent : public CStude
{
    public:
            void print () const;

};
```

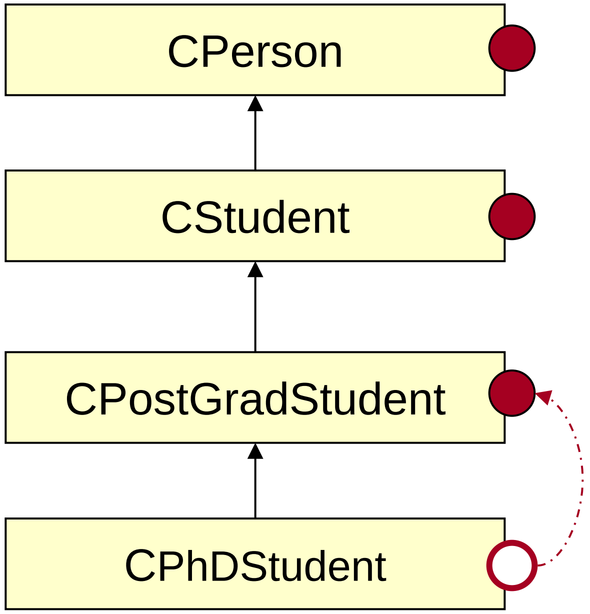
post_gr.cpp

```
void CPostGradStudent::print() const
{
     cout << " I am a postgraduate "
             “student!" << endl;
};
```

PhD_student.h

```
class CPhDStudent : public CPostGradStudent
{

};
```

```
int main (int argc, char* argv[])
{
            CPhDStudent s3;
            CPerson& r = s3;
            r.print();

            return 0;
}
```



```
I am a postgraduate student!
```

How C++ implements late binding

The keyword **virtual** tells the compiler it should not perform early binding.

The typical compiler creates a single table (called the **VTABLE**) for each class that contains **virtual** functions.

The compiler places the addresses of the virtual functions for that particular class in the **VTABLE**.

In each class with **virtual** functions, it secretly places a pointer, called the **vpointer** (abbreviated as **VPTR**), which points to the **VTABLE** for that object.

When you make a virtual function call through a base-class pointer (that is, when you make a **polymorphic call**), the compiler quietly inserts code to fetch the **VPTR** and look up the function address in the **VTABLE**, thus calling the correct function and causing late binding to take place.

How C++ implements late binding

```
int: 4
NoVirtual: 4
void*: 4
OneVirtual: 8
TwoVirtuals: 8
```

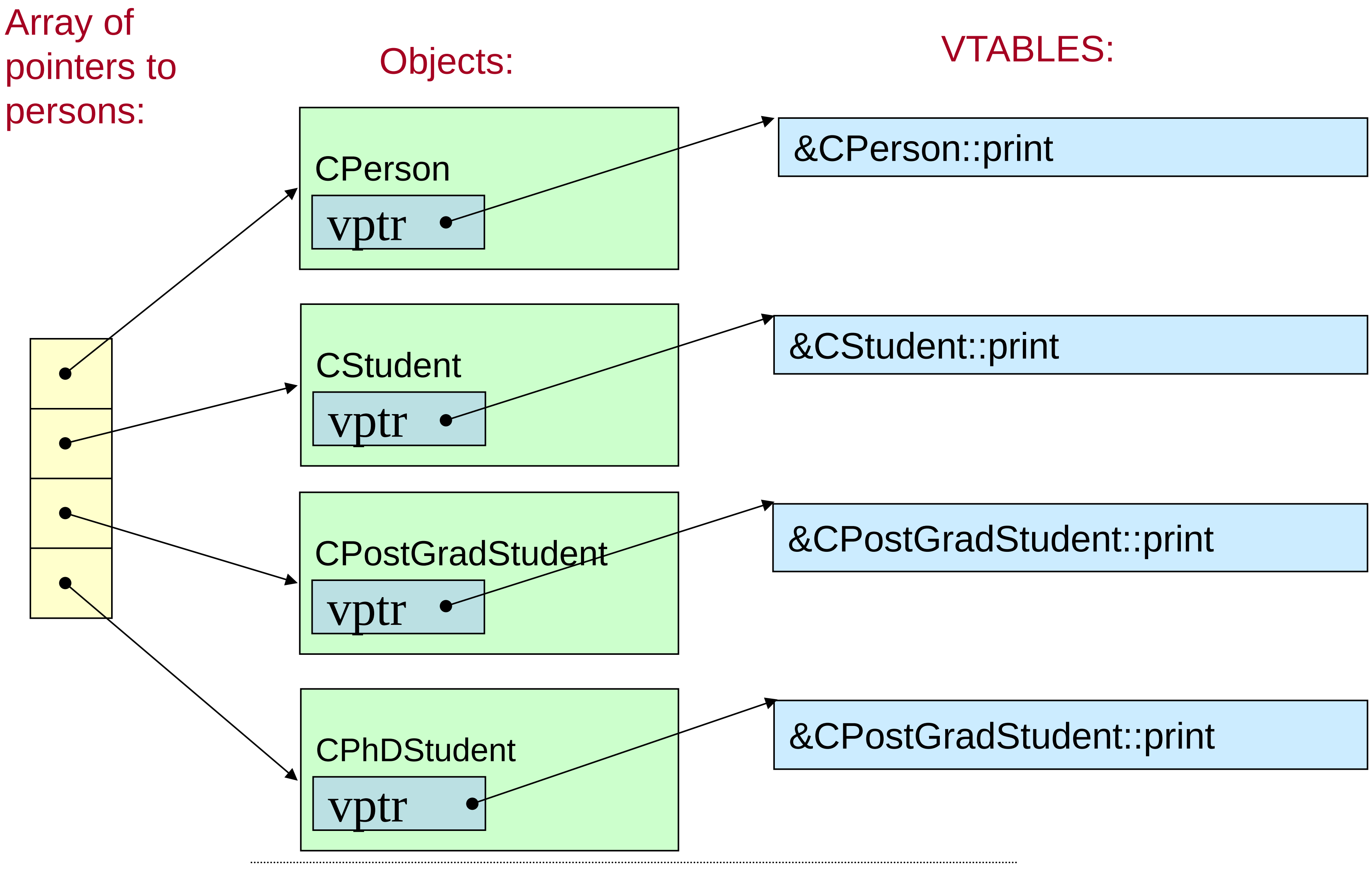
```
class NoVirtual
{
        int a;
public:
        void x() {}
        int i() { return 1; }
};
```

```
class OneVirtual
{
        int a;
public:
        virtual void x() {}
        int i() { return 1; }
};
```

```
class TwoVirtuals
{
        int a;
public:
        virtual void x() {}
        virtual int i() { return 1; }
};
```

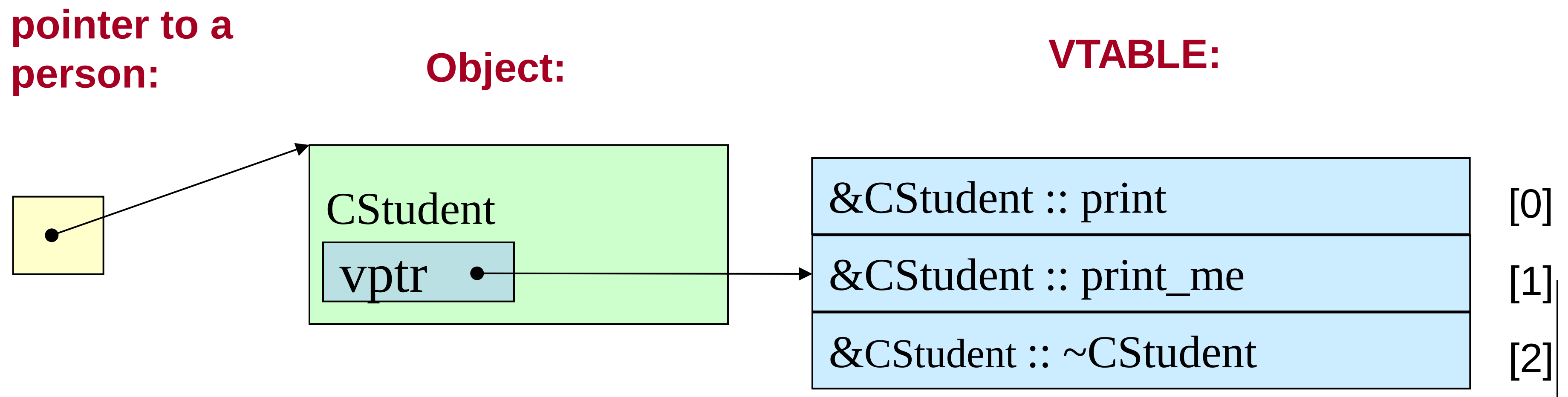
```
int main()
{
    cout << "int: " << sizeof(int) << endl;
    cout << "NoVirtual: " << sizeof(NoVirtual) << endl;
    cout << "void* : " << sizeof(void*) << endl;
    cout << "OneVirtual: " << sizeof(OneVirtual) << endl;
    cout << "TwoVirtuals: " << sizeof(TwoVirtuals) << endl;

    return 0;
}
```

```
CPerson* A [] = { new CPerson, new CStudent, new CPostGradStudent,
                  new CPhDStudent };
```



```
class CPerson
{
        public:
                virtual void print () const
                {
                        cout << “I am a person!" << endl;
                }
                virtual void print_me() const;
                virtual ~CPerson();
};
```

```
class CStudent : public CPerson
{
        public:
                void print () const
                {
                        cout << “I am a student!" << endl;
                }
                void print_me() const;
                ~CStudent();
};
```



```
CStudent s; CPerson& pr = s;  pr.print_me();
```

<u>no virtual</u>:
call the function CPerson::print_me

```
mov  ecx,dword ptr [ebp-14h]
call  @ILT+160(CPessoa::print_me) (004010a5)
```

<u>virtual</u>:
call the function at the address VPTR[1]

```
mov   ecx,dword ptr [ebp-14h]
mov   edx,dword ptr [ecx]
mov   esi,esp
mov   ecx,dword ptr [ebp-14h]
call   dword ptr [edx+4]
cmp   esi,esp
call   __chkesp (004038d0)
```

The vpointer is inicialized to the proper VTABLE in the constructor.

Upcasting deals only with addresses. If the compiler has an object, it knows the exact type and therefore will not use late binding for any function calls.

```
int main()
{
        CStudent s;
        s.print();          ←  static binding

        CPerson& pr = s;
        pr.print();         ←  dynamic binding

        CPerson* pp = &s;
        pp.print();         ←  dynamic binding

        return 0;
}
```

Overhead

Virtual destructor

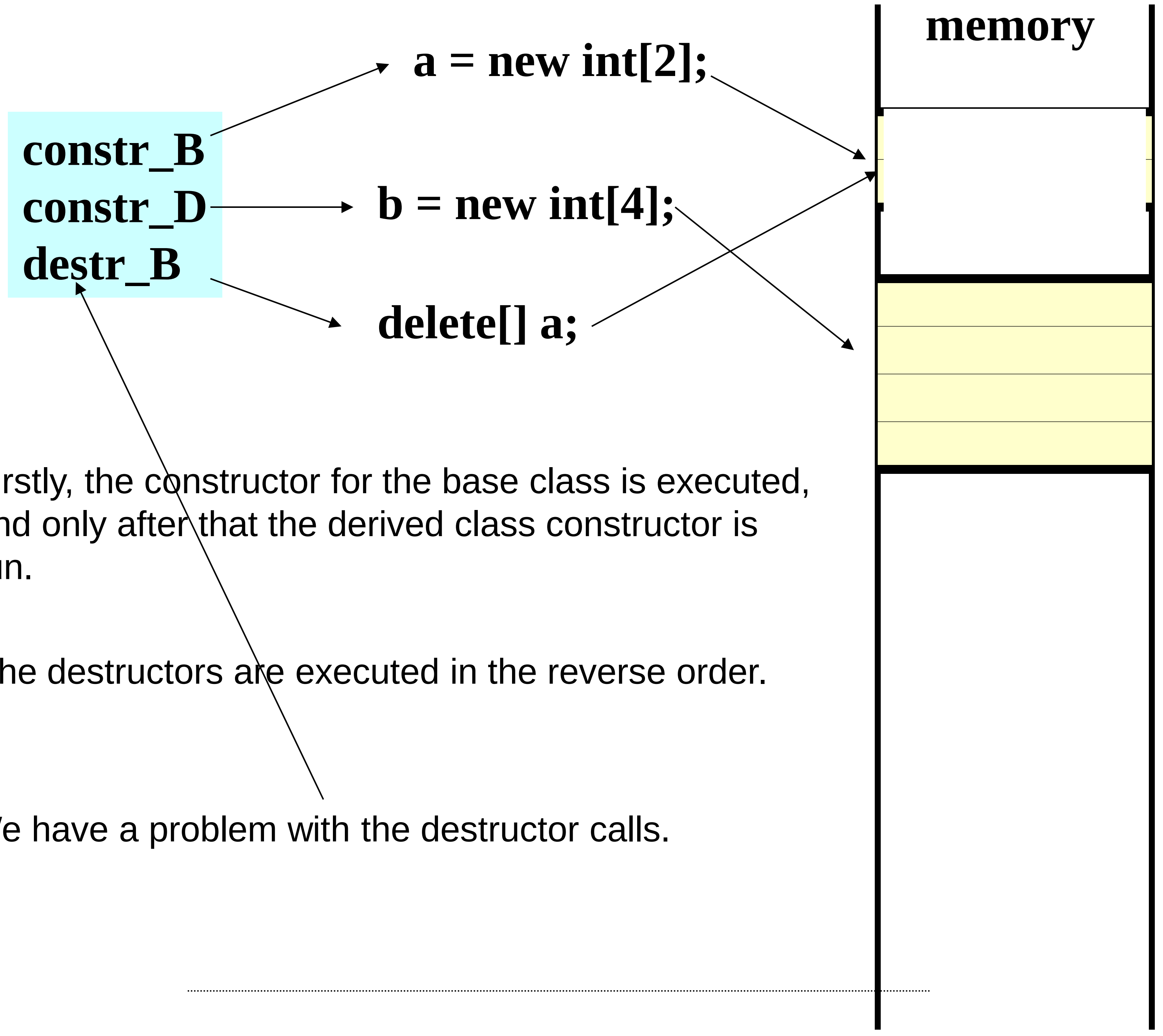
```
class B
{   int* a;
public:
   B() {  a = new int[2];
          cout << "constr_B\n"; }
  ~B(){  delete[] a;
          cout << "destr_B\n";   }
};
```

```
class D : public B
{   int* b;
public:
   D() {  b = new int[4];
          cout << "constr_D\n";  }
  ~D(){  delete[] b;
          cout << "destr_D\n";   }
};
```

```
int main(int argc, char* argv[])
{         B* pb = new D;
          delete pb;
          return 0;                    }
```

The results:

```
constr_B
constr_D
destr_B
```



Firstly, the constructor for the base class is executed, and only after that the derived class constructor is run.

The destructors are executed in the reverse order.

We have a problem with the destructor calls.

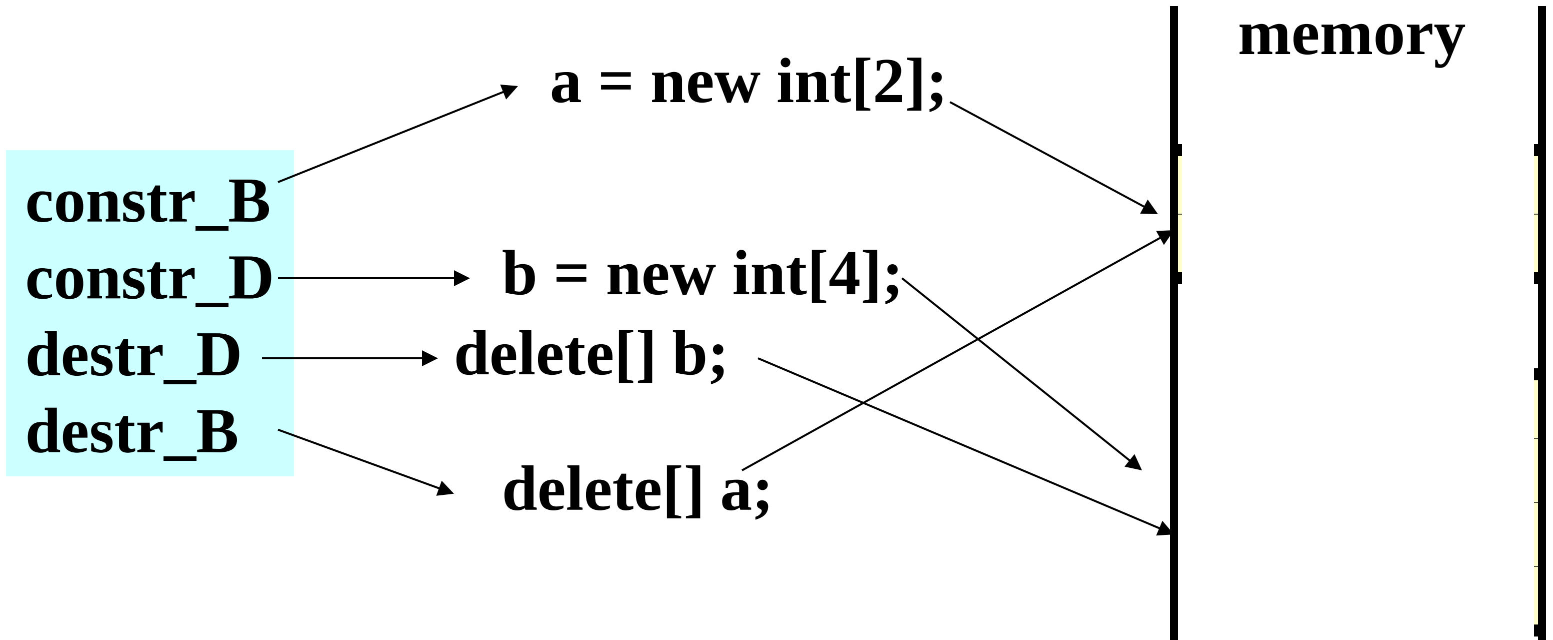
The problem is resolved with virtual destructors.

```
class B
{   int* a;
public:
   B() {  a = new int[2];
          cout << "constr_B\n"; }
  virtual ~B(){  delete[] a;
          cout << "destr_B\n";   }
};
```

```
class D : public B
{   int* b;
public:
   D() {  b = new int[4];
          cout << "constr_D\n";  }
  virtual ~D(){  delete[] b;
          cout << "destr_D\n";   }
};
```

The results:

```
constr_B
constr_D
destr_D
destr_B
```



It is possible for the destructor to be virtual because the object already knows what type it is (whereas it doesn't during construction). Once an object has been constructed, its VPTR is initialized, so virtual function calls can take place.

Forgetting to make a destructor virtual is an insidious bug because it often doesn't directly affect the behavior of your program, but it can quietly introduce a memory leak.

Constructors, destructors and virtual functions

Constructors cannot be virtual.

Destructors can and often must be virtual.

If you're inside an ordinary member function and you call a virtual function, that function is called using the late-binding mechanism.

Inside a constructor or a destructor, only the "local" version of the member function is called; the virtual mechanism is ignored.

The reason is that you'd be calling a function that might manipulate members that hadn't been initialized yet or had already been destroyed!

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Chapters 14, 15