Object-oriented programming

Second semester

Lecture №11

Virtual Method Tables

Why? Good photo

- Exposition
- Diaphragm
- White balance
- luminous flux
- …

Polymorphism

```
class Base {
  Integer x;
  public Base(Integer v) {
    x = v;
  }
  public void print() {
    System.out.println("Base: " + x);
  }
}
class Child extends Base {
  Integer y;
  public Child(Integer v1, Integer v2) {
    super(v1);
    y = v2;
  }
  public void print() {
    System.out.println("Child: (" + x + "," + y + ")");
  }
}
```

```
class BaseTest {
  public static void main(String[] args) {
    Base base1 = new Base(45);
    Base base2 = new Child(567, 245);
    base1.print();
    base2.print();
  }
}
```

Output:

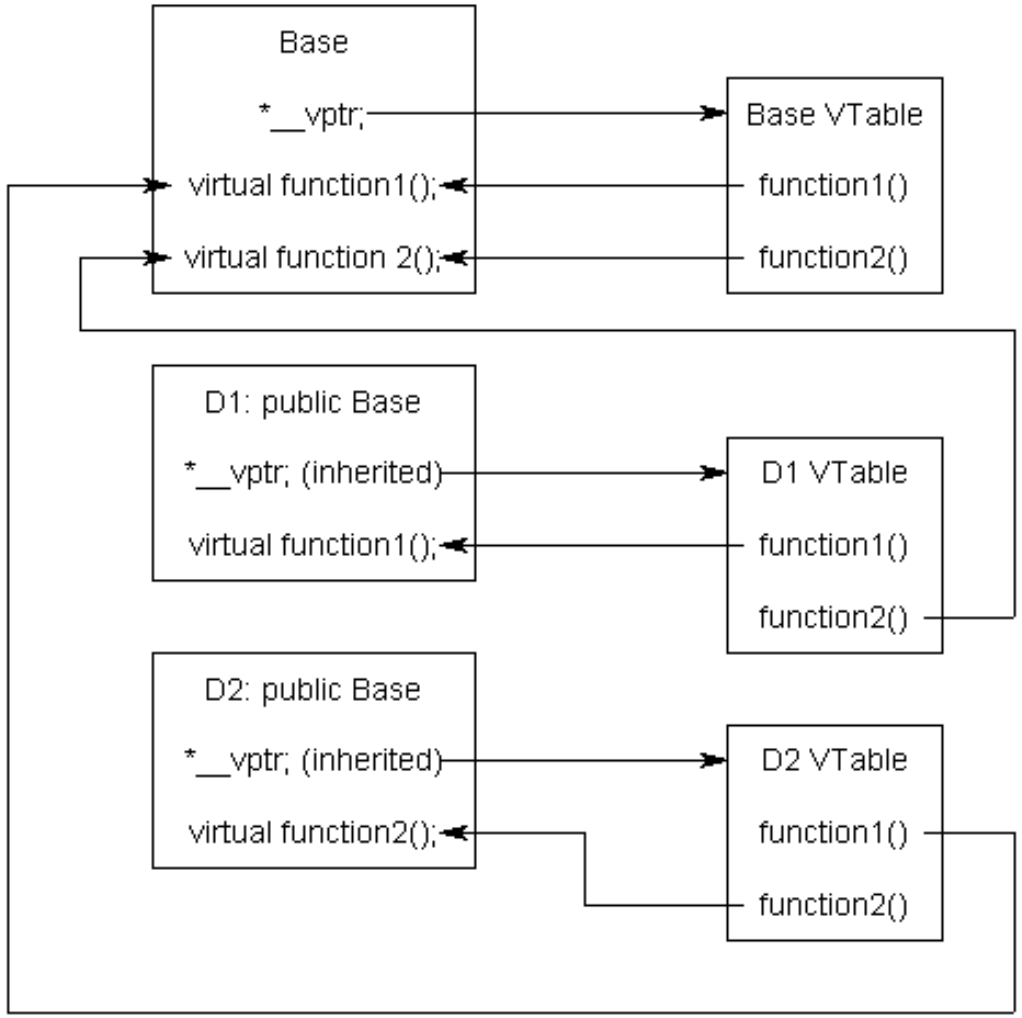
```
Base: 45
Child: (567,245)
```

Java -> C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Base {
  void (**vtable)();
  int x;
} Base;

void Base_print(Base* obj) {
  printf("Base: (%d)\n", obj->x);
}

void (*Base_Vtable[])() = { &Base_print };

Base* newBase(int v) {
  Base* obj = (Base*)malloc(sizeof(Base));
  obj->vtable = Base_Vtable;
  obj->x = v;
  return obj;
}
enum { Call_print };
void print(Base* obj) {
  obj->vtable[Call_print](obj);
}
```

```
typedef struct Child {
  void (**vtable)();
  int x; // inherited from Base
  int y;
} Child;

void Child_print(Child const* obj) {
  printf("Child: (%d,%d)\n", obj->x, obj->y);
}

void (*Child_Vtable[])() = { &Child_print };

Child* newChild(int v1, int v2) {
  Child* obj = (Child*)malloc(sizeof(Child));
  obj->vtable = Child_Vtable;
  obj->x = v1;
  obj->y = v2;
  return obj;
}

int main() {
  Base* base = newBase(45);
  Base* child = (Base*)newChild(567, 245);
  print(base);
  print(child);
}
```

Virtual Method Tables



ClassLoader

Multiple interfaces

```
C c  = new C();
B b =c;
I1 i1 = c;
I2 i2 = c;
i1.function1();
i2.function2();
```

idx	Function
0	B_function
1	I1_function1
2	I2_function2

?

```
class B{
   public void function(){}
}
interface I1 {
   void function1();
}
interface I2 {
   void function2();
}

Class C extends B implements I1,I2{
 public void function(){}
 public void function1(){}
 public void function2(){}
}
```

C++ solution

```
class Mother {
 public:
  virtual void MotherMethod() {}
  int mother_data;
};

class Father {
 public:
  virtual void FatherMethod() {}
  int father_data;
};

class Child : public Mother, public Father {
 public:
  virtual void ChildMethod() {}
  int child_data;
};
```

Child’s layout:

- _vptr$Mother
- mother_data (+ padding)
- _vptr$Father
- father_data
- child_data1

Virtual Machine solution

- No specification
- Smart pointers (Descriptor)
- Vtable && Itables

Performance

- A virtual call requires at least an extra indexed dereference and sometimes a "fixup" addition, compared to a non-virtual call, which is simply a jump to a compiled-in pointer. Therefore, calling virtual functions is inherently slower than calling non-virtual functions. An experiment done in 1996 indicates that approximately 6–13% of execution time is spent simply dispatching to the correct function, though the overhead can be as high as 50%. The cost of virtual functions may not be so high on modern CPU architectures due to much larger caches and better branch prediction.
- Furthermore, in environments where JIT compilation is not in use, virtual function calls usually cannot be inlined. In certain cases it may be possible for the compiler to perform a process known as devirtualization in which, for instance, the lookup and indirect call are replaced with a conditional execution of each inlined body, but such optimizations are not common.

call from super

```
public class D extends C {
    public String i;

    public D(String a, int b) {
        i = a;
        super.i = b;
    }

    public void print() {
        System.out.println("D.i = " + i);
        super.print();
    }
}
```