

תכנות מתקדם

מצגת 6

הורשה ורב צורתיות

# הורשה

- הורשה נועדה להגדיר מחלקות שיש להם מכנה משותף
  - המחלקה המורשה נקראת מחלקת הבסיס
  - המחלקה היורשת נקראת המחלקה הנגזרת
- הורשת מימוש
  - מחלקת הבסיס מספקת למחלקה היורשת פונקציות ונתונים מוכנים
- הורשת ממשק
  - מאפשרת שימוש במחלקות היורשות השונות באמצעות הממשק של מחלקת הבסיס המשותפת
  - את המחלקות היורשות נקצה בזיכרון הדינמי באמצעות new
  - ניגש באמצעות מצביעים או משתני ייחוס למחלקת הבסיס

# מחלקה יורשת slicing

```
class Base {  
    int x;  
};  
class Derived : public Base {  
    int y;  
};  
  
int main()  
{  
    Base base;  
    Derived derived;  
    cout << sizeof(base) << '\n';  
    cout << sizeof(derived) << '\n';  
}
```

# מחלקה יורשת slicing

```
Base base2(base) ;           // copy constructor
Derived derived2(derived) ;

Base base3(derived) ;        // copy constructor
Derived derived3(base) ;     // Error
cout << sizeof(base3) << '\n';

base2 = derived;             // assignment operator
derived2 = base;             // Error
cout << sizeof(base2) << '\n';
}
```

slice.cpp

# מחלקה יורשת Upcasting and Downcasting

```
Base* basep = &derived;           // OK, upcasting
Base& baser = derived;             // OK, upcasting

Derived* derivedp = &base;         // Error, downcasting
Derived& derivedr = base;          // Error, downcasting

Derived* derivedp2 = basep;        // Error

cout << "basep: " << sizeof(*basep) << '\n';
```

# Upcasting and Downcasting

- Converting a **derived-class** pointer to a **base-class** pointer is called **upcasting**.
- It is always **allowed** without the need for an explicit type cast.
  - A **Derived object is a Base object** in that it inherits all the data members and member functions of a Base object.
  - Thus, anything that we can do to a Base object, we can do to a Derived class object.
- Converting a **base-class** pointer to a **derived-class** pointer is called **downcasting**
- It is **not allowed** without an explicit type cast.
  - That's because a derived class could add new data members, and the class member functions that used these data members wouldn't apply to the base class.

# Upcasting and Downcasting

```
class Employee {  
private:  
    int id;  
public:  
    void show_id() { }  
};  
  
class Programmer : public  
Employee {  
public:  
    void coding() { }  
};
```

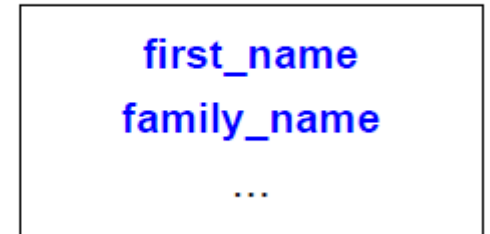
```
int main()  
{  
    Employee employee;  
    Programmer programmer;  
    Employee *pEmp =  
        &programmer; // upcast  
    Programmer *pProg = // down  
        (Programmer *) &employee;  
    pEmp->show_id();  
    pProg->show_id();  
    pEmp->coding(); // error  
    pProg->coding();
```

upcast.cpp

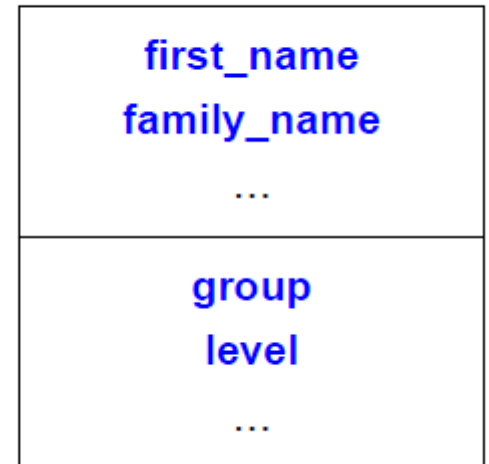
# מחלקה יורשת

```
class Employee {  
    string first_name , family_name;  
    Date hiring_date;  
    int department;  
};  
  
class Manager : public Employee {  
    list<Employee*> group;  
    int level;  
};
```

Employee:



Manager:





## שימוש בפונקציה של מחלקת הבסיס

```
class Employee {  
public:  
    void print() const;  
    // ...  
};  
  
class Manager:public Employee  
{  
public:  
    void print() const;  
    // ...  
};  
  
void Manager::print() const  
{  
    // print Employee info  
    Employee::print();  
    // print Manager info  
    cout << level;  
    // ...  
}
```

## הצורך בפונקציה וירטואלית

- A **base** class **pointer** (or **reference**) can point to a **derived** class object:

```
void f(Manager m1, Employee e1, Employee e2)
{
    list<Employee*> elist {&m1, &e1, &e2};
    print_list(elist);
}
```

- Problem, a base-class pointer (or reference) can invoke just base class methods

```
void print_list( const list<Employee*>& elist ) {
    for(auto x : elist)
        x -> print();
}
```

# פונקציה וירטואלית

- Virtual functions in a base class can be redefined in each derived class:
- The compiler ensures that the right **print()** for the given Employee object is invoked in each case

```
class Employee {  
public:  
    Employee(const string& name, int dept);  
    virtual void print() const;  
private:  
    string first_name , family_name;  
    short department;  
};
```

virtual.cpp

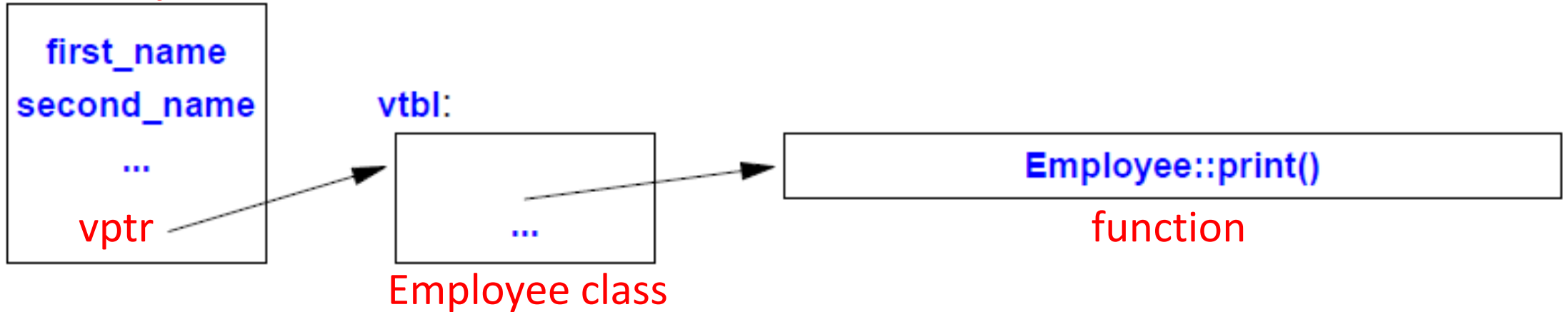
poly.cpp

## רב צורתיות

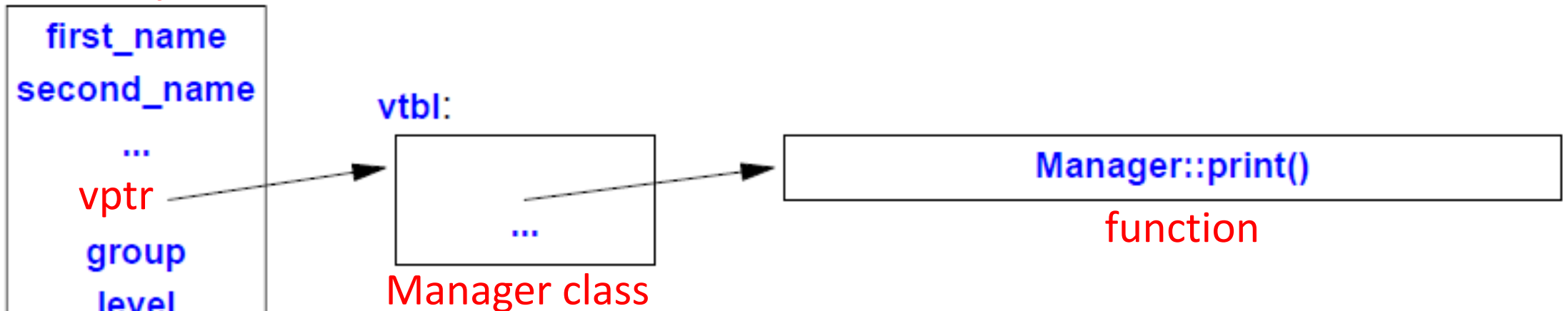
- כדי שתופעל הפונקציה של האובייקט שהמצביע מכיל את כתובתו ולא הפונקציה של סוג המצביע יש צורך בפולימורפיזם
- כדי לממש פולימורפיזם יש צורך לשמור בכל אובייקט את סוג האובייקט ואז אפשר לקרוא לפונקציה המתאימה
- לכל מחלקה שמכילה פונקציה וירטואלית יש טבלה `vtbl` של מצביעים לפונקציות
- לכל אובייקט של מחלקה וירטואלית יש מצביע `vptr` לטבלת הפונקציות של המחלקה שלו.
- שם הפונקציה משמש כאינדקס לטבלת המצביעים לפונקציות

# virtual function table

Employee: object

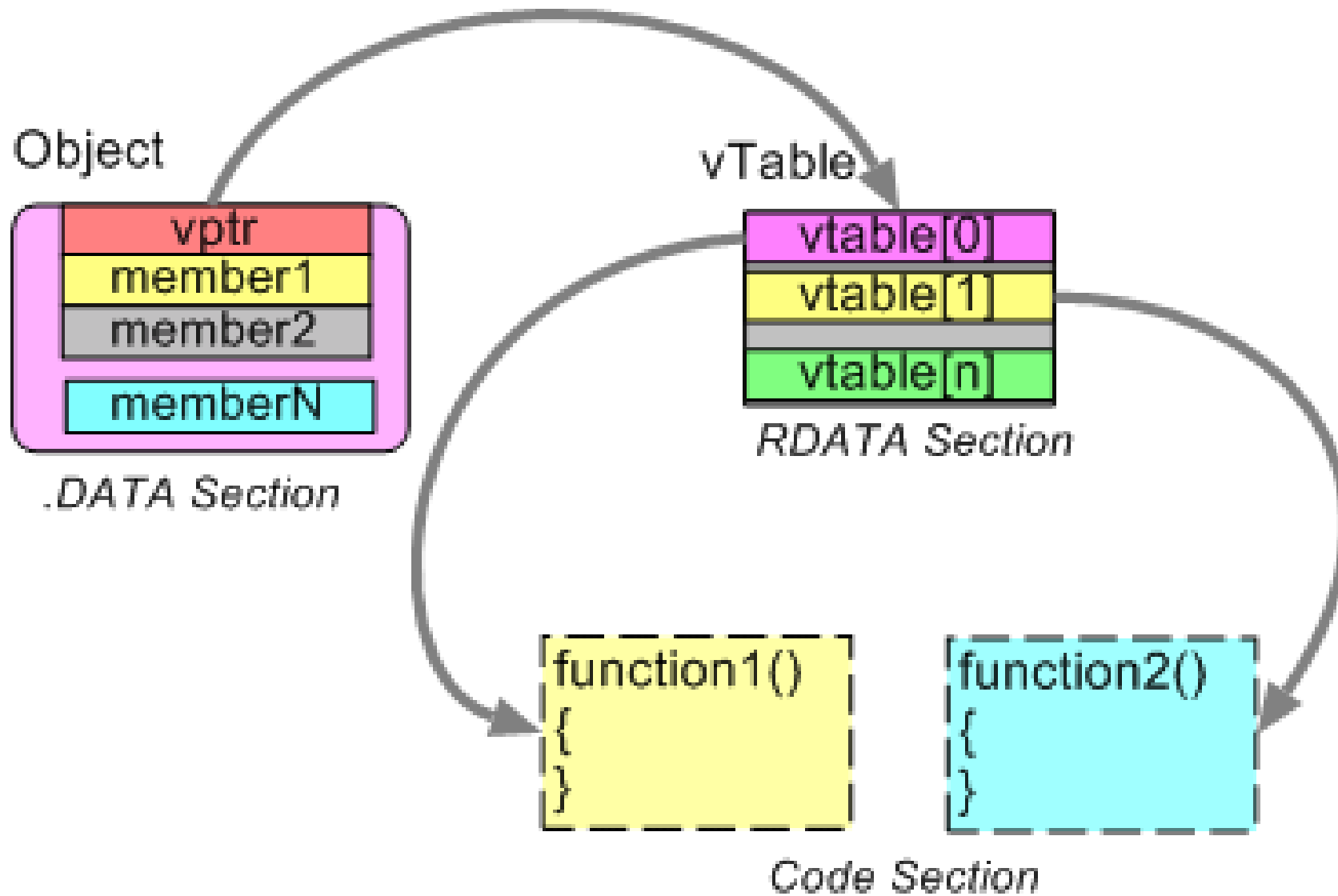


Manager: object

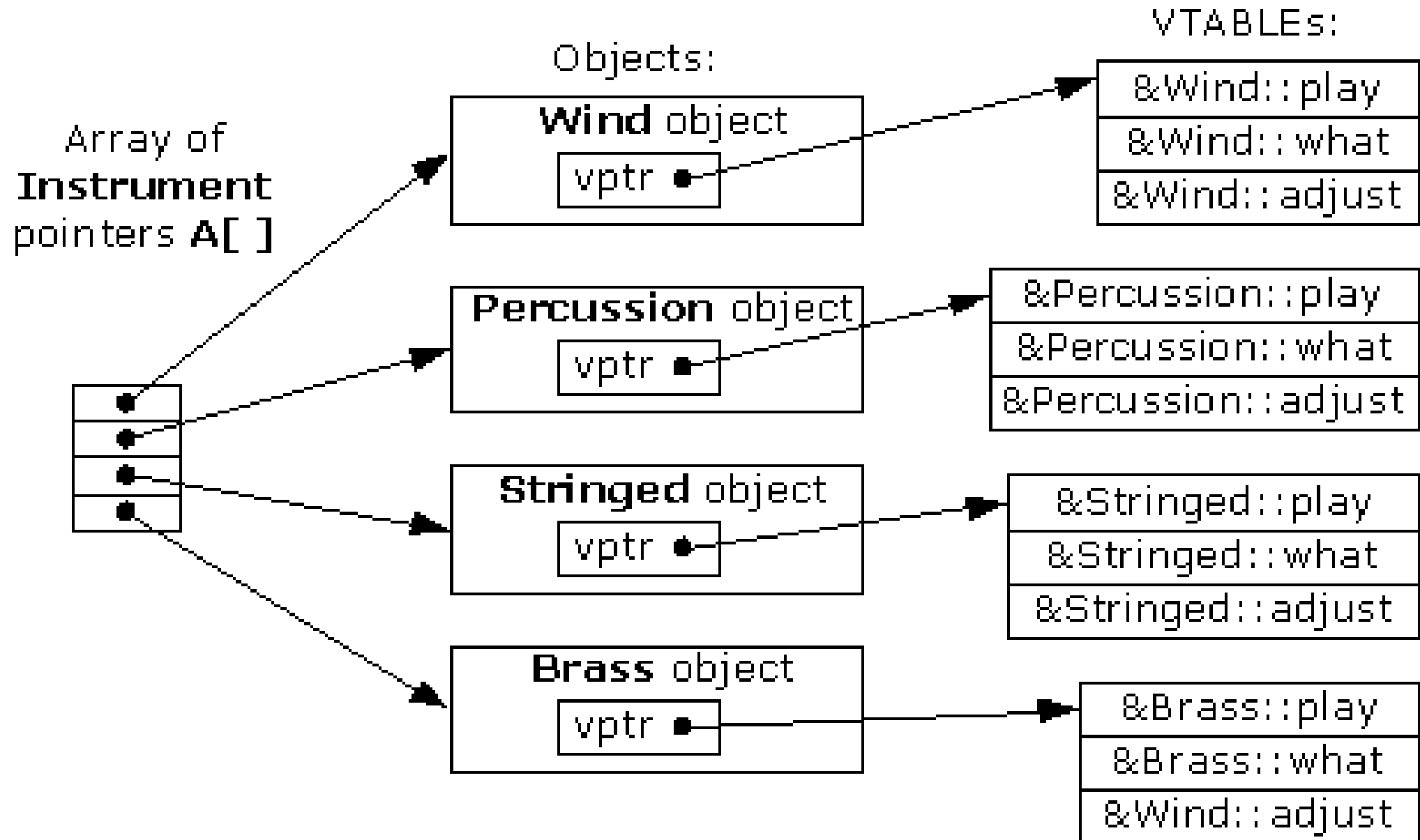


vptr.cpp

# Virtual Tables



# Virtual Tables



# Virtual Tables

For every class that contains virtual functions, the compiler constructs a **vtable** (virtual table)

The **vtable** contains an entry for each virtual function accessible by the class and stores a pointer to its definition

Entries in the **vtable** can point to either functions declared in the class itself, or virtual functions inherited from a base class.

Every time the compiler creates a **vtable** for a class, it adds class member which is a pointer to the corresponding virtual table, called the **vpointer**

When a call to a virtual function on an object is performed, the **vpointer** of the object is used to find the corresponding **vtable** of the class.

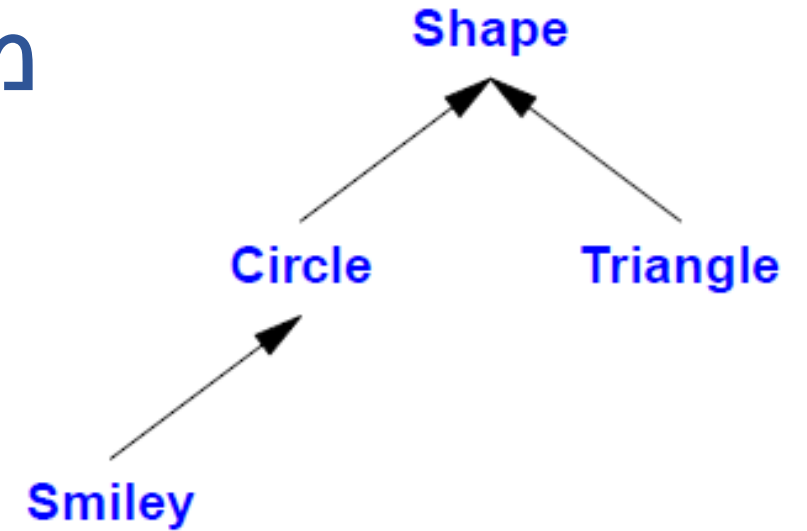
Next, the function name is used as index to the **vtable** to find the correct routine to be executed



# מחלקה אבסטרקטית

- A class with a **pure virtual function** is called an **abstract class**

```
class Shape {  
public:  
    virtual Point center() const = 0; // pure virtual  
    virtual void move(Point to) = 0;  
    virtual void draw() const = 0;  
    virtual void rotate(int angle) = 0;  
    virtual ~Shape() {} // virtual destructor is essential  
                        // since an object of a derived  
                        // class may be deleted  
                        // through a pointer to the base  
}
```



# הגדרת עיגול כמחלקה יורשת של צורה

```
class Circle : public Shape {  
public:  
    Circle(Point p, int rr); // constructor  
    Point center() const { return x; }  
    void move(Point to) { x = to; }  
    void draw() const;  
    void rotate(int) {} // nice simple algorithm  
private:  
    Point x; // center  
    int r; // radius  
};
```

# הגדרת סמיילי כמחלקה יורשת של עיגול

```
class Smiley : public Circle {
public:
    Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }
    ~Smiley() {delete mouth; for (auto p : eyes) delete p;}
    void move(Point to); void draw() const; void rotate(int);
    void add_eye(Shape* s) { eyes.push_back(s); }
    void set_mouth(Shape* s);
private:
    vector<Shape*> eyes; // usually two eyes
    Shape* mouth;
};
```

# override

- A function in a derived class **overrides** a virtual function in a base class if that function has **exactly the same name and type**
- A function with a slightly different name or a slightly different type may be intended to override but will define another function
- A programmer can explicitly state that a function is meant to **override**

```
class Smiley : public Circle {  
    // . . .  
    void move(Point to) override;  
    void draw() const override;  
    void rotate(int) override;
```

## רב צורתיות בפעולה

```
void rotate_all(vector<Shape*>& v, int angle)
{
    for (auto p : v) p->rotate(angle);
}

void user()
{
    std::vector<Shape*> v;
    while (cin) v.push_back(read_shape(cin));
    draw_all(v);           // call draw() for each element
    rotate_all(v,45);      // call rotate(45) for each element
    for (auto p : v) delete p;
}
```

shapes.cpp

```
enum class Kind { circle, triangle , smiley };  
Shape* read_shape(istream& is)  
{ // read shape header from is and find its kind k  
  switch (k) {          // Kind k;  
  case Kind::circle:    // read {Point,int}  
    return new Circle{p,r};  
  case Kind::triangle: // read {Point,Point,Point}  
    return new Triangle{p1,p2,p3};  
  case Kind::smiley:   // read {Point,int,Shape,Shape,Shape}  
    Smiley* ps = new Smiley{p,r};  
    ps->add_eye(e1); ps->add_eye(e2); ps->set_mouth(m);  
    return ps;  
  }  
}
```

# RTTI - Run-time Type Identification

- אם יש מצביע למחלקת הבסיס ורוצים לבצע פעולות שמוגדרות למחלקה יורשת
- `RTTI` מאפשר לקבוע בזמן ריצה לאיזה אובייקט הוא מצביע בפועל
- `dynamic_cast` ממיר ממצביע למחלקת הבסיס, למצביע למחלקה יורשת
- אם המצביע למחלקת הבסיס לא מצביע בפועל לאותה מחלקה יורשת, הערך המוחזר הוא `nullptr`
- בתכנית שכתובה היטב אין צורך ב- `RTTI`

# dynamic\_cast

```
class Base { virtual void f() {} };
class Derived : public Base { };

int main()
{
    Base *pBase1 = new Base;
    Base *pBase2 = new Derived;
    Derived *pD;

    pD = dynamic_cast<Derived*>(pBase1);    // pD = nullptr
    cout << (pD ? "OK" : "nullptr") << "\n";

    pD = dynamic_cast<Derived*>(pBase2);    // OK
    cout << (pD ? "OK" : "nullptr") << "\n";
}
```

dynamic\_cast.cpp



# dynamic\_cast

- We can use `dynamic_cast` to get type information

```
Shape* ps {read_shape(cin)};
// is a Smiley pointed to by ps ?
if (Smiley* p = dynamic_cast<Smiley*>(ps)) {
    // returns pointer to Smiley
    // yes, a Smiley
}
else {
    // returns nullptr
    // not a Smiley, try something else
}
```