תכנות מתקדם
מצגת 4

הורשה ורב צורתיות

נושאים

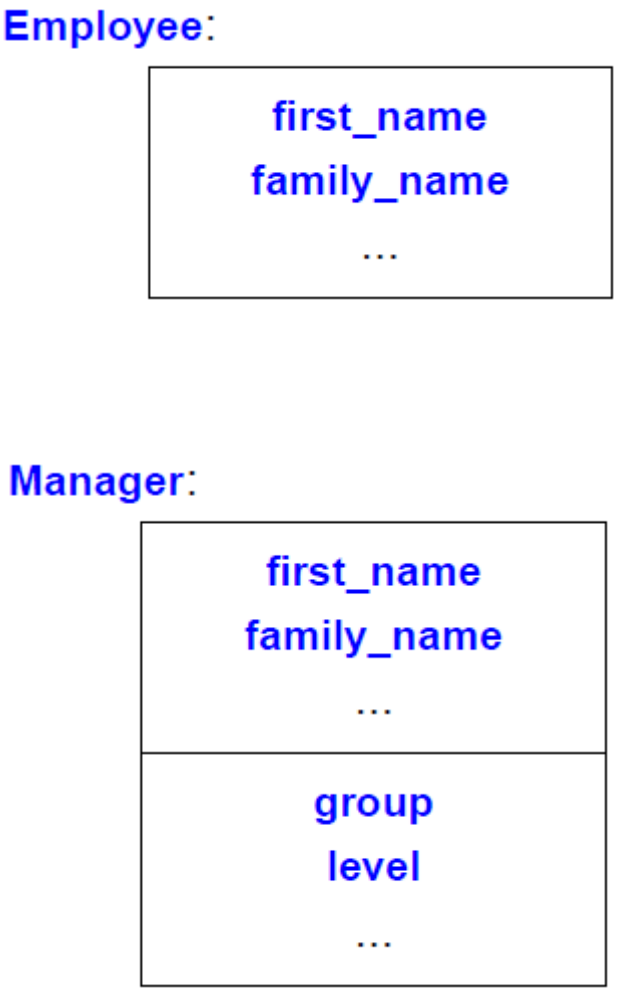
- הורשה
- פונקציה וירטואלית
- רב צורתיות
- vtbl
- מחלקה אבסטרקטית
- override

הורשה

- הורשה נועדה להגדיר מחלקות שיש להם מכנה משותף
  - המחלקה המורישה 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

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



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();
}
```

poly.cpp

פונקציה וירטואלית

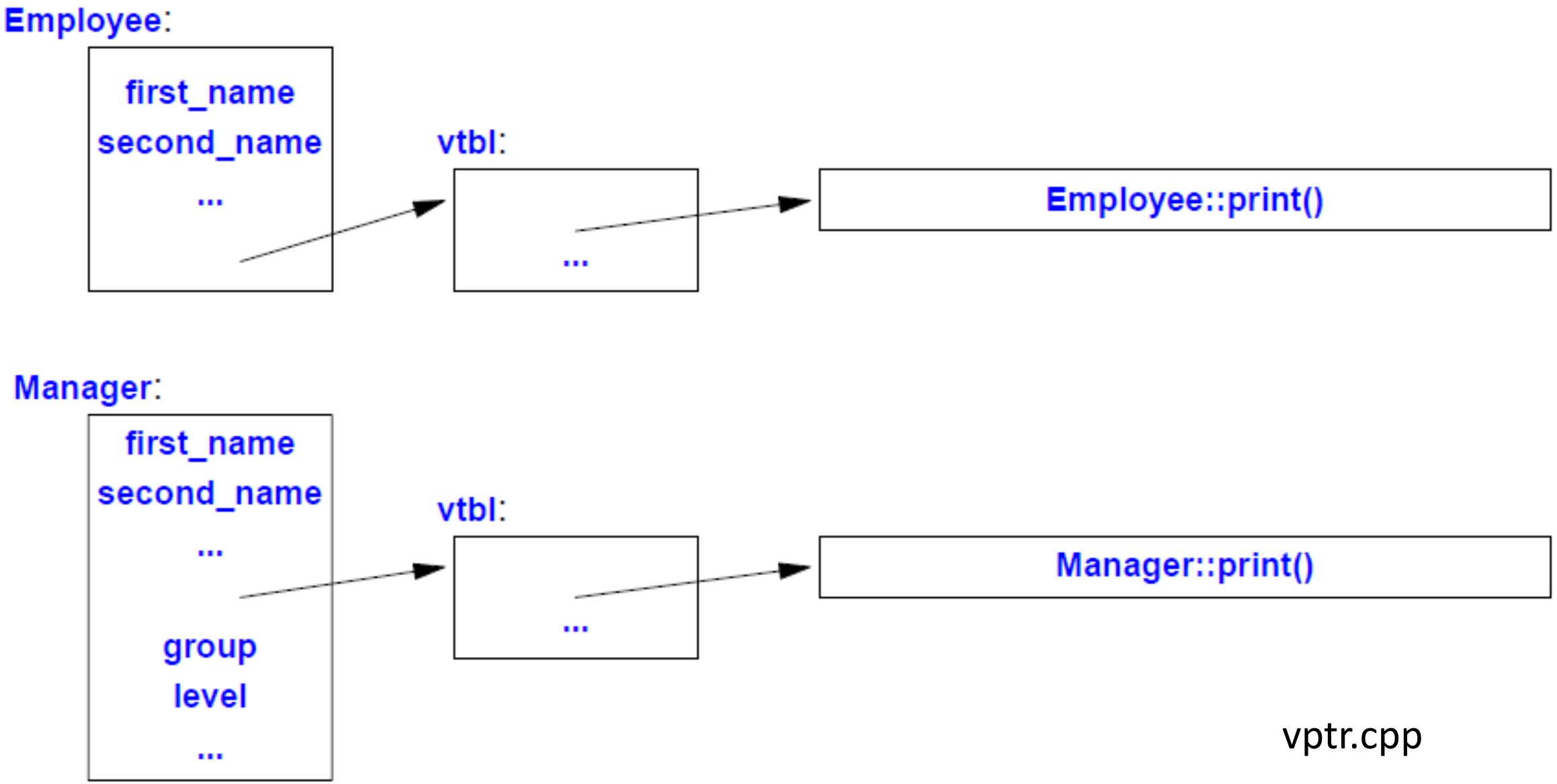
- Virtual functions in a base class can be redefined in each derived class:
- The compiler ensures that the right print() for the given Employee object is invoked in each case

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

רב צורתיות

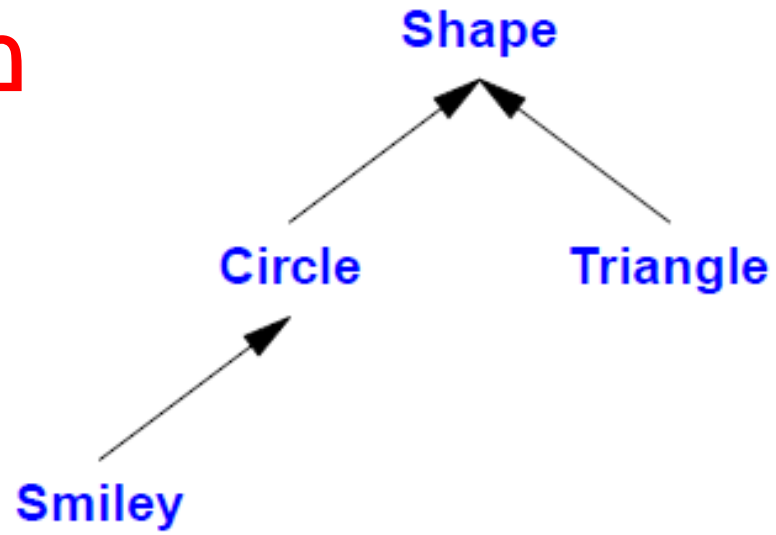
- Getting ‘‘the right’’ behavior from Employee’s functions independently of exactly what kind of Employee is actually used is called **polymorphism**
- Clearly, to implement **polymorphism**, the compiler must store some kind of type information in each object of class Employee and use it to call the right version of the virtual function print()
- The compiler converts the name of a virtual function into an index into a table of pointers to functions
- That table is usually called the **virtual function table** or simply the **vtbl**
- Each class with virtual functions has its own **vtbl** identifying its virtual functions
- Each object of a class with a virtual function has a pointer to its class **vtbl**

virtual function table



vptr.cpp

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;
```

shapes.cpp

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;
}
```

```
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;
}
}
```

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();
```

RTTI - Run-time Type Identification

- RTTI enables a program to determine the type of object **during runtime**.
- The need arises because you want to perform **derived class operation** on a **derived class object**, but you have only a **pointer** (or reference) **to base**
- RTTI is provided through two operators:
  - The dynamic_cast operator, which safely converts from a pointer (or reference) to a base type, to a pointer (or reference) to a derived type.
    - It answers the question of whether we can safely assign the address of an object to a pointer of a particular type
    - If the object bound to the pointer is **not** an object of the target type, it fails and the value is 0
  - The typeid operator, which returns the actual type of the object.
- Querying the type of an object at run-time frequently means a design problem

dynamic_cast

```
class Base { };
class Derived : public Base { };
int main()
{
    Base *pBBase = new Base;
    Base *pBDerived = new Derived;
    Derived *pd;

    pd = dynamic_cast<Derived*>(pBDerived); // OK
    pd = dynamic_cast<Derived*>(pBBase);    // pd = 0
    return 0;
}
```

dynamic_cast

- We can use dynamic_cast to get type information

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

Virtual Tables

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

Base* b = new Derived();
b->bar();
```

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

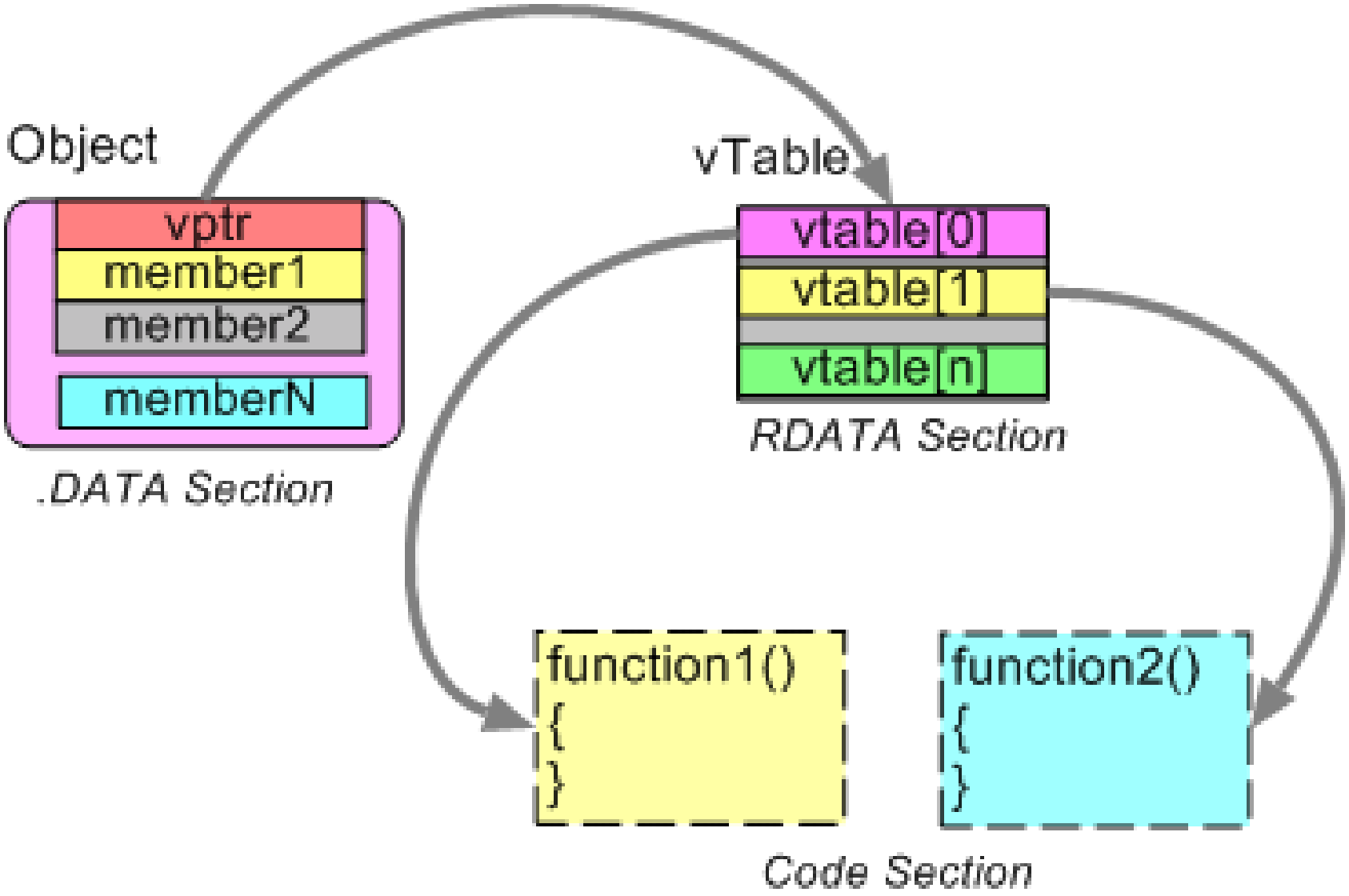
Entries in the vtable can point to either functions declared in the class itself, or virtual functions inherited from a base class.

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

Virtual Tables



Virtual Tables