

# CSc 453 — Compilers and Systems Software

## 23 : OO Languages

Christian Collberg  
Department of Computer Science  
University of Arizona  
collberg@gmail.com

Copyright © 2009 Christian Collberg

November 3, 2009

## 1 Object-Oriented Languages

- Object-oriented languages extend imperative languages with:
  1. A classification scheme that allows us to specify is-a as well as has-a relationships. Has-a is supported by Pascal, where we can declare that one data item **has** another item (a record variable *has-a* record field). Object-Pascal, Oberon, etc, extends this capability with **inheritance** which allows us to state that one data item **is** (an extension of) another item.
  2. Late binding, which allows us to select between different implementations of the same abstract data type at run-time.

## 2 Object-Oriented Languages...

- 3. Polymorphism, which is the ability of a variable to store values of different types. OO languages support a special kind of polymorphism, called inclusion polymorphism, that restricts the values that can be stored in a variable of type  $T$  to values of type  $T$  or subtypes of  $T$ .
  4. Data encapsulation. Data (instance variables) and operations (methods) are defined together.
  5. Templates and objects. A template (**class** or **prototype**) describes how to create new objects (instances of abstract data types).

## 3 Compiling OO Languages

- Runtime type checking (a variable of type **ref**  $T$  may only reference objects of type  $T$  or  $T$ 's subtypes).
- Because of the polymorphic nature of OO languages, we can't always know (at compile-time) the type of the object that a given variable will refer to at run-time. When we invoke a method we can't actually know which piece of code we should execute. Finding the right piece of code is called **method lookup**. It can be done by name (Objective-C) or number (C++).
- Most OO languages rely on dynamic allocation. Garbage collection is a necessary part of the runtime system of a compiler for an OO language (C++ non-withstanding). This requires **runtime type description**.

# Example

## 5

```

TYPE Shape = CLASS
  x, y : REAL;
  METHOD draw(); BEGIN ...; END;
  METHOD move(X,Y:REAL); BEGIN x := x+X; END;
END;
TYPE Square = Shape CLASS
  side : REAL;
  METHOD draw(); BEGIN ...; END;
END;
TYPE Circle = Shape CLASS
  radius : REAL;
  METHOD draw(); BEGIN ...; END;
  METHOD area():REAL; BEGIN ... END;
END;

```

## 6 Example in Java

```

// Example in Java

class Shape {
  double x, y;
  void draw(); { ... }
  void move(double X, double Y); {x = x+X; }}
class Square extends Shape {
  double side;
  void draw(); { ...}}
class Circle extends Shape {
  double radius;
  void draw(); { ... }
  double area(); { ... }}

```

## 7 Example in Modula-3 (A)

```

(* Example in Modula-3 *)
TYPE Shape = OBJECT
  x, y : REAL
  METHODS
  draw() := DefaultDraw; move(X, Y : REAL):=Move;
END;
Square = Shape OBJECT
  side : REAL
  METHODS
  draw() := SquareDraw

```

```

END;
Circle = Shape OBJECT
  radius : REAL
  METHODS
  draw() := CircleDraw; area() := ComputeArea
END;

```

## 8 Example in Modula-3 (B)

```

(* Example in Modula-3 (continued) *)
PROCEDURE Move (Self : Shape; X, Y : REAL) =
BEGIN ... END Move;

PROCEDURE DefaultDraw (Self : Shape) =
BEGIN ... END DefaultDraw;

PROCEDURE SquareDraw (Self : Square) =
BEGIN ... END SquareDraw;

PROCEDURE CircleDraw (Self : Circle) =
BEGIN ... END CircleDraw;

PROCEDURE ComputeArea (Self : Circle) : REAL =
BEGIN ... END ComputeArea;

```

## 9 Example in Oberon-2

```

TYPE  Shape = RECORD x, y : REAL END;
      Square = RECORD (Shape) side : REAL END;
      Circle = RECORD (Shape) radius : REAL END;
PROCEDURE (Self : Shape) Move (X, Y : REAL) =
BEGIN ... END Move;
PROCEDURE (Self : Shape) DefaultDraw () =
BEGIN ... END DefaultDraw;
PROCEDURE (Self : Square) SquareDraw () =
BEGIN ... END SquareDraw;
PROCEDURE (Self : Circle) CircleDraw () =
BEGIN ... END CircleDraw;
PROCEDURE (Self : Circle) ComputeArea () : REAL =
BEGIN ... END ComputeArea;

```

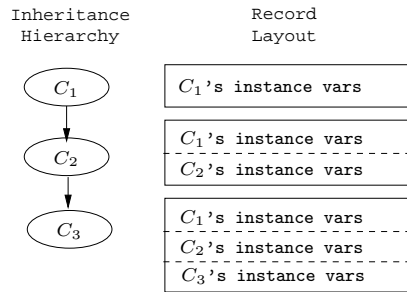
## 10

# Record Layout

## 11 Record Layout

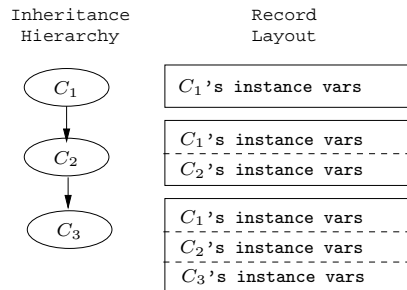
- Single inheritance is implemented by *concatenation*, i.e. the instance variables of class *C* are

1. the variables of  $C$ 's supertype, *followed by*
2. the variables that  $C$  declares itself.



## 12 Record Layout

- The offsets of the variables that  $C$  inherits from its supertype will be the same as in the supertype itself.
- In this example,  $C_3$  inherits from  $C_2$  which inherits from  $C_1$ .
- $C_3$  will have the fields from  $C_1$  followed by the fields from  $C_2$  followed by  $C_3$ 's own fields. The order is significant.



## 13 Record Layout...

```

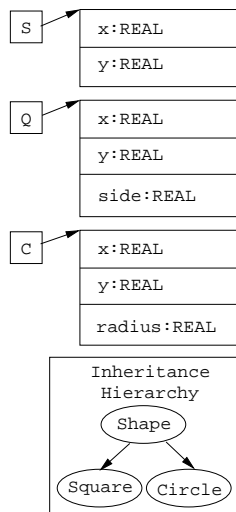
TYPE Shape =
  CLASS x,y: REAL; END;

TYPE Square = Shape
  CLASS side:REAL; END;

TYPE Circle = Shape
  CLASS radius:REAL; END;

VAR S:Shape;
VAR Q:Square;
VAR C:Circle;

```



## 14 Record Layout...

- An OO language compiler would translate the declarations in the previous slide into something similar to this:

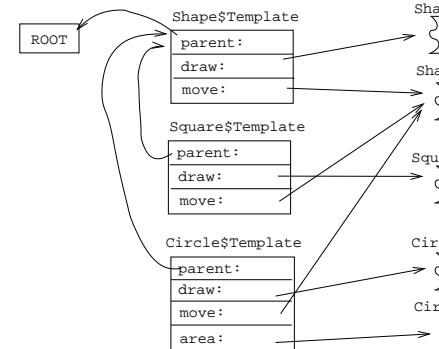
```
TYPE Shape=POINTER TO RECORD
  x, y: REAL;
END;
TYPE Square=POINTER TO RECORD
  x, y: REAL;
  side:REAL;
END;
TYPE Circle=POINTER TO RECORD
  x, y: REAL;
  radius:REAL;
END;
VAR S:Shape; Q:Square; C:Circle;
```

15

# Templates

## 16 Class Templates

To support late binding, runtime typechecking, etc, each class is represented by a *template* at runtime. Each



template has pointers to the class's methods and supertype.

## 17 Class Templates...

- Square's x,y fields are inherited from Shape. Their offsets are the same as in Shape.

```
TYPE $TemplateT=POINTER TO RECORD
  parent : $TemplateT;
  move : ADDRESS;
  draw : ADDRESS;
END;
TYPE Square=POINTER TO RECORD
  $template : $TemplateT;
  x, y : REAL;
```

```

    side : REAL;
  END;
CONST Square$Template:$TemplateT =
  [ parent= ADDR(Shape$Template);
    move = ADDR(Shape$move);
    draw = ADDR(Square$draw); ];

```

## 18

Each method is a procedure with an extra argument (**SELF**), a pointer to the object through which the method was invoked.

```

TYPE Shape = CLASS
  x, y : REAL;
  METHOD draw (); BEGIN ...;
  METHOD move (X, Y : REAL);
  BEGIN x := x+X; ... END;
END;

```

↓

```

PROCEDURE Shape$move (SELF : Shape; X,Y:REAL);
BEGIN
  SELF^.x := SELF^.x + X;
  SELF^.y := SELF^.y + X;
END;

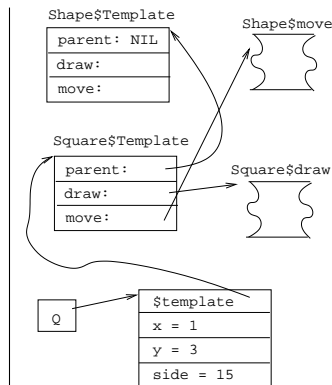
```

## 19

# Method Lookup

## 20 Method Invocation

- Sending the message draw to Q:
  1. Get Q's template, *T*.
  2. Get draw's address at offset 4 in *T*.
  3. Jump to draw's address, with Q as the first argument.



## 21

```

VAR Q : Square;
BEGIN
  Q := NEW (Square);
  Q.x := 1; Q.y := 3; Q.side := 15;

```

```

    Q.draw(); Q.move(20, 30);
END;
    ↓
BEGIN
    Q := malloc(SIZE(Square));
    Q^.$template := Square$Template;
    Q^.x := 1; Q^.y := 3; Q^.side := 15;
    Q^.$template^.draw(Q);
    Q^.$template^.move(Q, 20, 30);
END;

```

## 22

# Runtime Type Checking

## 23 Inclusion Polymorphism

Consider the last two lines of the example in the following slide:

- In  $L_1$ ,  $S$  points to a `Shape` object, but it could just as well have pointed to an object of any one of `Shape`'s subtypes, `Square` and `Circle`.
- If, for example,  $S$  had been a `Circle`, the assignment  $C := S$  would have been perfectly OK. In  $L_2$ , however,  $S$  is a `Shape` and the assignment  $C := S$  is illegal (a `Shape` isn't a `Circle`).

## 24 Inclusion Polymorphism

```

VAR S : Shape; Q : Square; C : Circle;
BEGIN
    Q := NEW (Square);
    C := NEW (Circle);

    S := Q; (* OK *)
    S := C; (* OK *)

    Q := C; (* Compile-time Error *)

    L1: S := NEW (Shape);
    L2: C := S; (* Run-time Error *)
END;

```

## 25 Typechecking Rules

```

TYPE  T = CLASS ... END;
      U = T CLASS ... END;
      S = T CLASS ... END;
VAR   t,r : T; u : U; s : S;

```

- A variable of type  $T$  may refer to an object of  $T$  or one of  $T$ 's subtypes.

Assignment	Compile-time	Run-Time
t := r;	Legal	Legal
t := u;	Legal	Legal
u := t;	Legal	Check
s := u;	Illegal	

## 26 Run-time Type Checking

---

Modula-3 Type-test Primitives:

---

ISTYPE(object, T) Is object's type a subtype of T?

NARROW(object, T) If object's type is *not* a subtype of T, then issue a run-time type error. Otherwise return object, typecast to T.

TYPECASE Expr OF Perform different actions depending on the runtime type of Expr.

- The assignment `s := t` is compiled into `s := NARROW(t, TYPE(s))`.

## 27 Run-time Type Checking...

- The Modula-3 runtime-system has three functions that are used to implement typetests, casts, and the TYPECASE statement
- **NARROW** takes a template and an object as parameter. It checks that the type of the object is a subtype of the type of the template. If it is not, a run-time error message is generated. Otherwise, **NARROW** returns the object itself.

1. ISTYPE(S,T : Template) : BOOLEAN;
2. NARROW(Object, Template) : Object;
3. TYPECODE(Object) : CARDINAL;

## 28

# Algorithm

## 29 Run-time Checks

- Casts are turned into calls to **NARROW**, when necessary:

```

VAR S : Shape; VAR C : Circle;
BEGIN
  S := NEW (Shape); C := S;
END;

      ↓
VAR S : Shape; VAR C : Circle;
BEGIN
  S := malloc (SIZE(Shape));
  C := NARROW(S, Circle$Template);
END;

```



## 30 Implementing ISTYPE

- We follow the object's template pointer, and immediately (through the templates' parent pointers) gain access to its place in the inheritance hierarchy.

```
PROCEDURE ISTYPE (S, T : TemplatePtr) : BOOLEAN;
BEGIN
  LOOP
    IF S = T THEN RETURN TRUE; ENDIF;
    S := S^.parent;
    IF S = ROOT THEN RETURN FALSE; ENDIF;
  ENDLOOP
END ISTYPE;
```

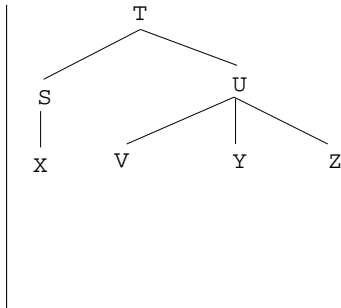
## 31 Implementing NARROW

- **NARROW** uses **ISTYPE** to check if **S** is a subtype of **T**. If so, **S** is returned. If not, an exception is thrown.

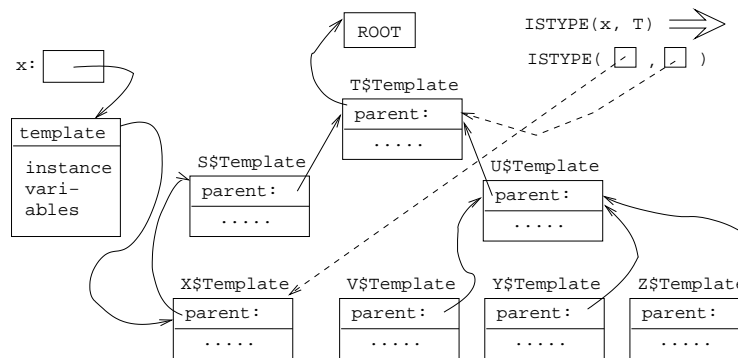
```
PROCEDURE NARROW(T:TemplatePtr; S:Object):Object;
BEGIN
  IF ISTYPE(S^.template, T) THEN
    RETURN S (* OK *)
  ELSE WRITE "Type error"; HALT;
  ENDIF;
END NARROW;
```

## 32 Run-time Checks — Example

```
TYPE T = CLASS [...];
      S = T CLASS [...];
      U = T CLASS [...];
      V = U CLASS [...];
      X = S CLASS [...];
      Y = U CLASS [...];
      Z = U CLASS [...];
VAR x : X;
```



### 33 Run-time Checks — Example...



34

## Compile-Time Organization

### 35 Organizing the Symbol Table

- In  $C.M$ 's method body we can refer to
  1.  $M$ 's locals and formals, and  $M$ 's **SELF**.
  2.  $C$ 's methods and instance variables.
  3. Methods and instance variables of  $C$ 's superclasses.

```
TYPE T = CLASS [  
  v : INTEGER; c : CHAR;  
  METHOD P(x:INTEGER); BEGIN ...v...c... END;  
  METHOD Q(x:CHAR); BEGIN ...v...c... END;  
];  
TYPE U = T CLASS [  
  c : REAL; k : INTEGER;  
  METHOD P(x:INTEGER); BEGIN ...v...c...k... END;  
  METHOD Q(r:REAL); BEGIN ...v...c...k... END;  
];
```

36

## Homework

### 37 Exam Problem

- In the following object-oriented program
  - "TYPE U = T CLASS" means that U inherits from T.
  - NEW T means that a new object of type T is created.
  - All methods are *virtual*, i.e. a method in a subclass overrides a method with the same name in a superclass.

```

PROGRAM X;
  TYPE T = CLASS [
    v : INTEGER; c : CHAR;
    METHOD P (x:INTEGER); BEGIN ... END P;
    METHOD Q (x:CHAR); BEGIN ... END Q;
  ];

```

### 38 Exam Problem I...

```

TYPE U = T CLASS [
  x : REAL; k : INTEGER;
  METHOD R(x:INTEGER); BEGIN ... END R;
  METHOD Q(r:REAL); BEGIN ... END Q;
];
VAR t : T; u : U;
BEGIN
  t := NEW T; u := NEW U; ◇
END

```

1. Draw a figure that describes the state of the program at point ◇. It should have one element for each item stored in memory (i.e. global/heap variables, templates, method object code, etc.) and should explicitly describe what each pointer points to.

### 39

## Summary

### 40 Readings and References

- Read the Tiger book:
  - Object-oriented Languages** pp. 283–298
- For information on constructing layouts for multiple inheritance, see
  - William Pugh and Grant Weddell: “Two-directional record layout for multiple inheritance.”
- The time for a type test is proportional to the depth of the inheritance hierarchy. Many algorithms do type tests in constant time:
  1. Norman Cohen, “Type-Extension Type Tests can be Performed in Constant Time.”
  2. Paul F.Dietz, “Maintaining Order in a Linked List”.

### 41 Summary

- For single inheritance languages, an instance of a class  $C$  consists of (in order):
  1. A pointer to  $C$ 's template.
  2. The instance variables of  $C$ 's ancestors.
  3.  $C$ 's instance variables.

- For single inheritance languages, subtype checks can be done in  $\mathcal{O}(1)$  time.
- Method invocation is transformed to an indirect call through the template.
- If we can determine the exact type of an object variable at compile time, then method invocations through that variable can be turned into “normal” procedure calls.

## 42 Summary...

- A template for class  $C$  consists of (in order):
  1. A pointer to the template of  $C$ 's parent.
  2. The method addresses of  $C$ 's ancestors.
  3. Addresses of  $C$ 's methods.
  4. Other information needed by the runtime system, such as
    - The size of a  $C$  instance.
    - $C$ 's pre- and postorder numbers, if the  $\mathcal{O}(1)$  subtype test algorithm is used.
    - $C$ 's type code.
    - A type description of  $C$ 's instance variables. Needed by the garbage collector.

## 43 Confused Student Email

*What happens when both a class and its subclass have an instance variable with the same name?*

- The subclass gets both variables. You can get at both of them, directly or by casting. Here's an example in Java:

```
class C1 {int a;}
class C2 extends C1 {double a;}
class C {
    static public void main(String[] arg) {
        C1 x = new C1(); C2 y = new C2();
        x.a = 5; y.a = 5.5;
        ((C1)y).a = 5;
    }
}
```