Code Generation II

Lecture 21

Lecture Outline

- Code generation for OO languages
  - Object layout
  - Dynamic dispatch
- Parameter-passing mechanisms
- Allocating temporaries in the AR
  - Clean-up from lecture 19

Code Generation for OO Languages

Topic I

Object Layout

- OO implementation = Stuff from last lecture + More stuff
- OO Slogan: If B is a subclass of A, than an object of class B can be used wherever an object of class A is expected
- This means that code in class A works unmodified for an object of class B

Two Issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?

Object Layout Example

```
class A {
    int a = 0;
    int d = 1;
    int f() { a = a + d; return a; }
};

class B extends A {
    int b = 2;
    int f() { return a; }
    int g() { a = a – b; return a; }
};

class C extends A {
    int c = 3;
    int h() { a = a * c; return a; }
};
```

Object Layout (Cont.)

- Fields a and d are inherited by classes B and C
- All methods in all classes refer to a
- For A methods to work correctly in A, B, and C objects, field a must be in the same "place" in each object

Object Layout (Cont.)

An object is like a struct in C. The reference

foo.field

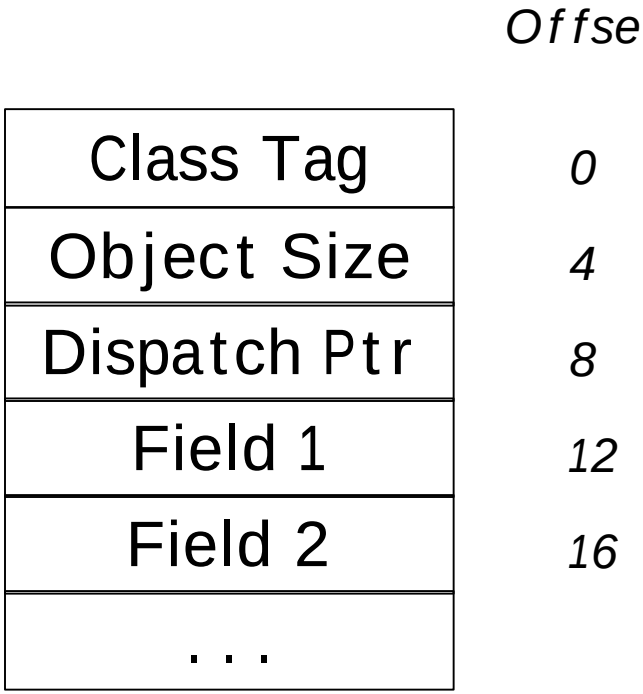
is an index into a foo struct at an offset corresponding to field

Objects in Java/C++ are implemented similarly

- Objects are laid out in contiguous memory
- Each field is stored at a fixed offset in object

A Sample Object Layout

- The first 3 words of an object contain header information:



Sample Object Layout (Cont.)

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object in words
- Dispatch ptr is a pointer to a table of methods
  - More later
- Fields in subsequent slots

- Lay out in contiguous memory

Subclasses

Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional field of B

Leaves the layout of A unchanged

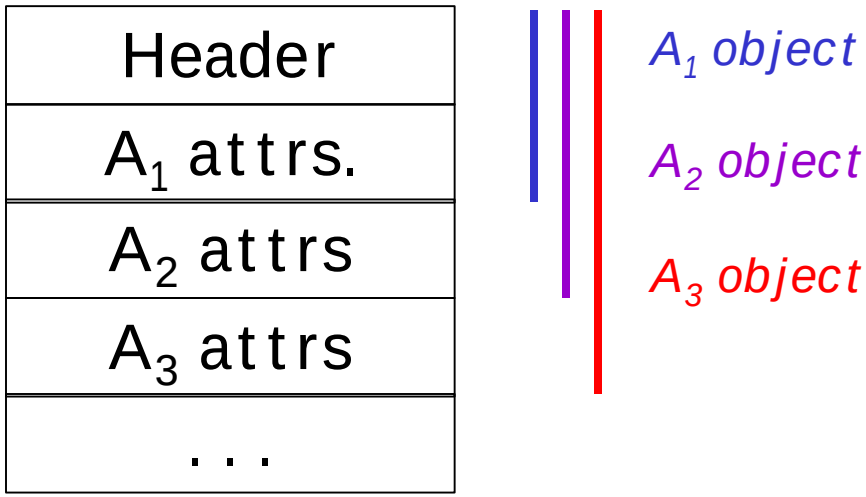
(B is an extension of A)

Layout Picture

Offset Class	0	4	8	12	16	20
A	Atag	5	*	a	d	
B	Btag	6	*	a	d	b
C	Ctag	6	*	a	d	c

Subclasses (Cont.)

- The offset for a field is the same in a class and all of its subclasses
  - Any method for an A_1 can be used on a subclass A_2
- Consider layout for $A_n < \ldots < A_3 < A_2 < A_1$



What about multiple inheritance?

Dynamic Dispatch

- Consider the following dispatches (using the same example)

Object Layout Example (Repeat)

```
class A {
    int a = 0;
    int d = 1;
    int f() { a = a + d; return a; }
};

class B extends A {
    int b = 2;
    int f() { return a; }
    int g() { a = a – b; return a; }
};
```

```
class C extends A {
    int c = 3;
    int h() { a = a * c; return a; }
};
```

Dynamic Dispatch Example

- e.g()
  - g refers to method in B if e is a B
- e.f()
  - f refers to method in A if f is an A or C (inherited in the case of C)
  - f refers to method in B for a B object

- The implementation of methods and dynamic dispatch strongly resembles the implementation of fields

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)

- A *dispatch table* indexes these methods
  - dispatch table = an array of method entry points
  - A method f lives at a fixed offset in the dispatch table for a class and all of its subclasses

Dispatch Table Example

Offset Class	0	4
A	fA	
B	fB	g
C	fA	h

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A to the right
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset O_f in the dispatch table at compile time

Using Dispatch Tables (Cont.)

- To implement a dynamic dispatch e.f() we
  - Evaluate e. The result is a pointer to an object x
  - Call $D[O_f]$
    - D is the dispatch table for x
    - In the call, this is bound to x

Parameter Passing Mechanisms

Topic II

Parameter Passing Mechanisms

- There are many semantic issues in programming languages centering on when values are computed and the scope of names
  - we've already seen static vs. dynamic scoping

- Now we'll focus on parameter passing
  - When are arguments of function calls evaluated?
  - To what objects are the formal parameters bound?

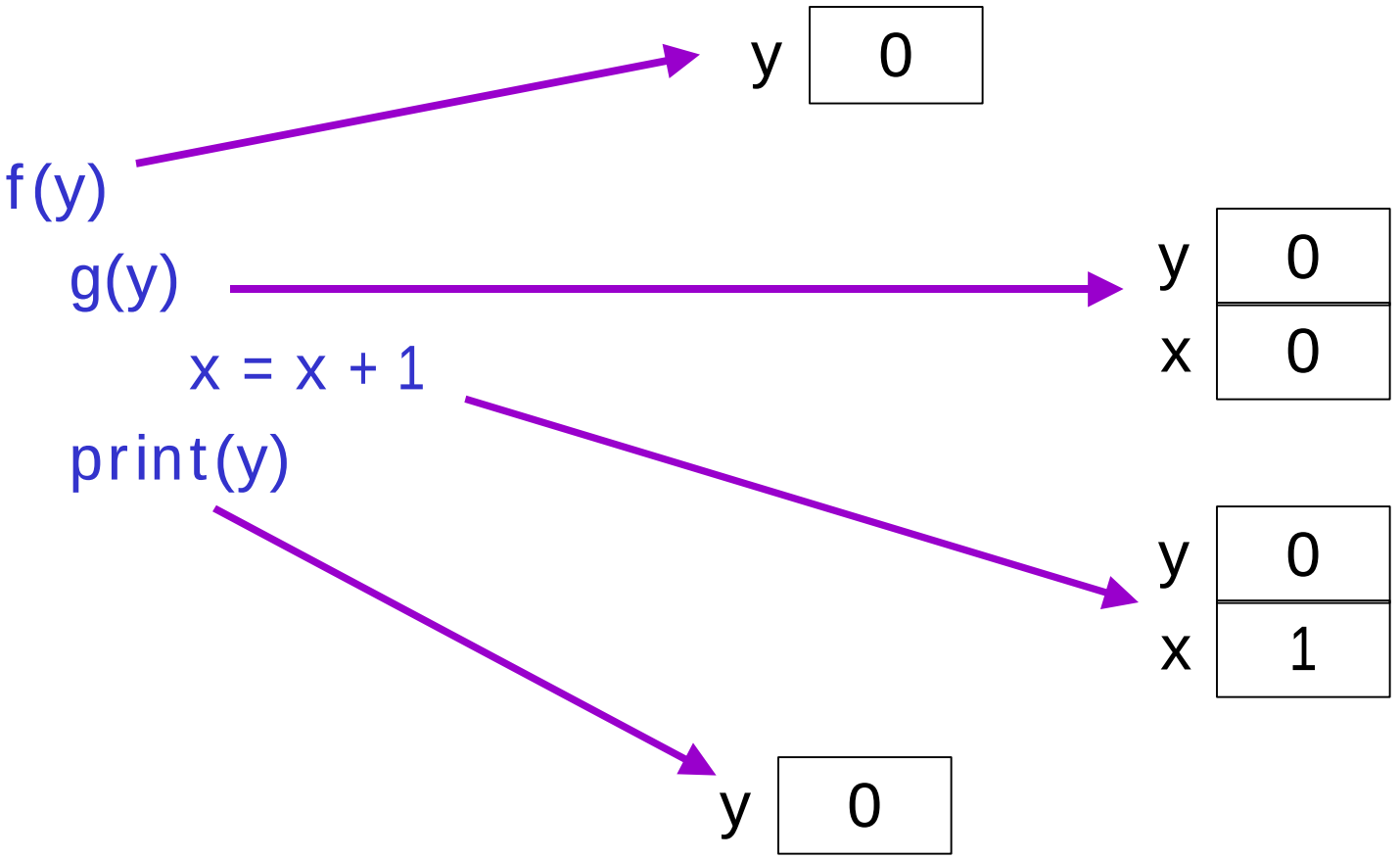
Call-By-Value

- To evaluate f(e)
  - Evaluate e to a value v
  - Bind v to the formal parameter in the function body
- Example

```
void g(x) { x = x + 1; }
void f(y) { g(y); print(y); }
```

- Under call-by-value, f(0) prints 0.

The Stack Under Call-By-Value



Call-By-Value Discussion

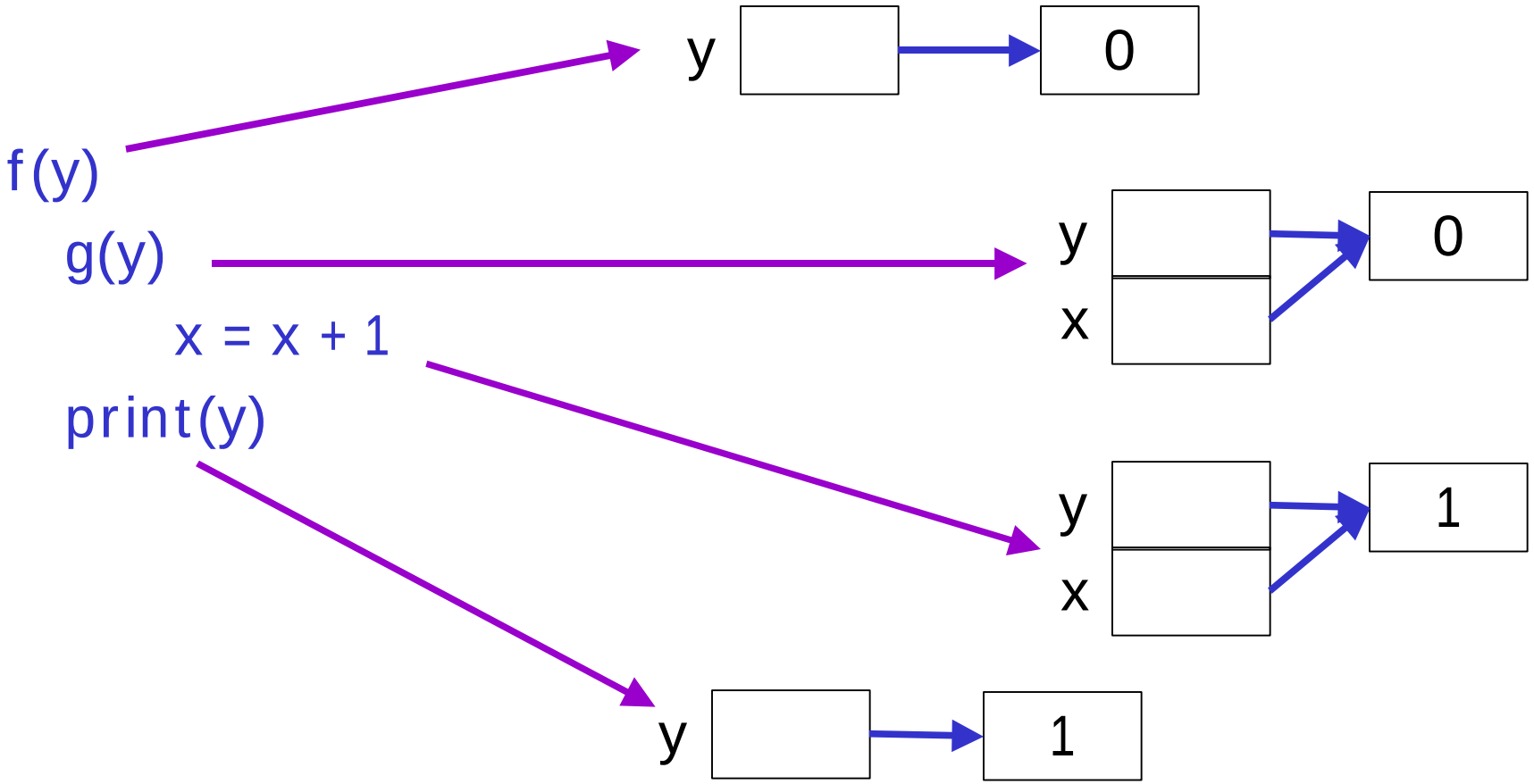
- Under call-by-value, g(y) does not affect the value of y
- y's value, not y itself, is passed to g
  - The formal parameter is stored in a different location from the actual parameter
- Call-by-value is widely used
  - C, C++, Java are prominent examples

Call-By-Reference

- To evaluate f(e)
  - e is evaluated
  - A pointer to e is passed as the argument
  - f's code accesses the argument through the pointer

- If e is already a stored value (i.e., a variable) a pointer to that location is used
- Otherwise, e is evaluated and stored in a fresh, temporary location first

The Stack Under Call-By-Reference



Call-By-Reference Discussion

- Under Call-By-Reference, only the address is passed
  - References to the value dereference the pointer
- In the example, because x and y refer to the same value, changes to x also change y
- Many languages pass large data structures (e.g., arrays) by reference

Call-By-Reference Discussion (Cont.)

- IMPORTANT: There is a difference between passing a value by reference and passing a pointer by value!

- In "old" C (i.e., pre-ANSI C), one could not pass structures as arguments;
  - hence one passed pointers to the structures
  - the pointer was passed by value

Call-By-Name

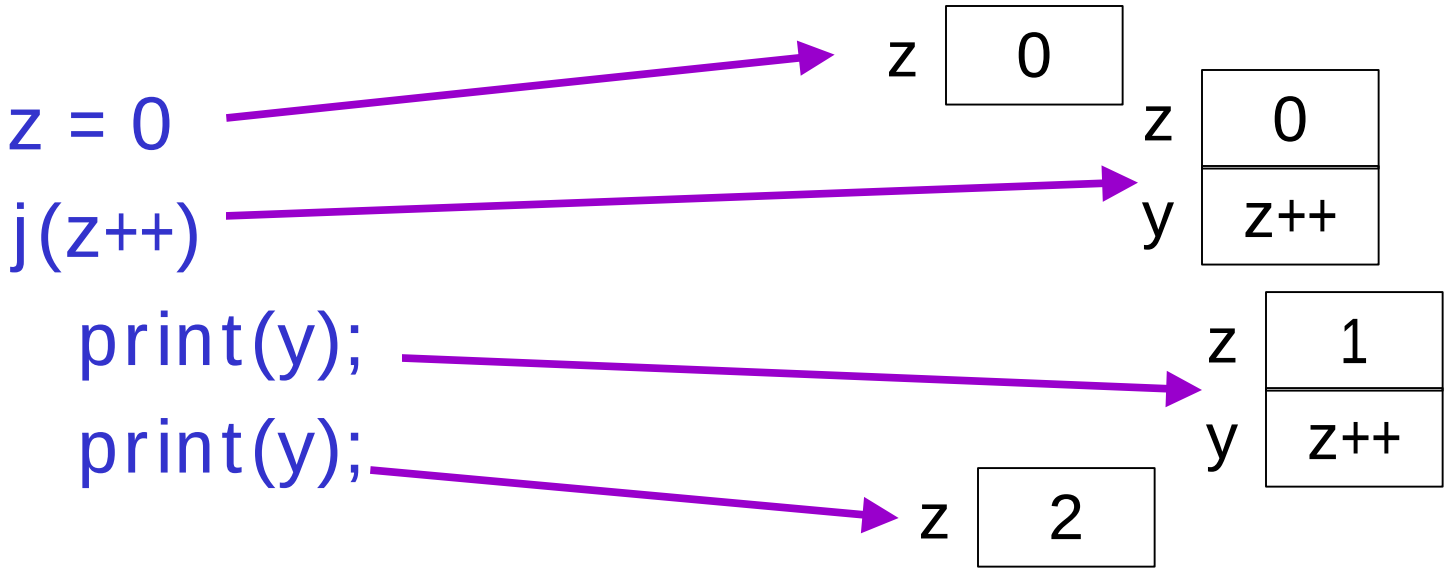
- Consider a definition f(x) { return e; }

- To evaluate f(e')
  - e' is passed *unevaluated* to f
  - e' is evaluated each time e refers to x
  - The expression e' is always evaluated in its original environment

The Stack Under Call-By-Name

Example

```
void j(y) { print(y); print(y); }
void main { z = 0; j(z++); }
```



Call-By-Name Discussion

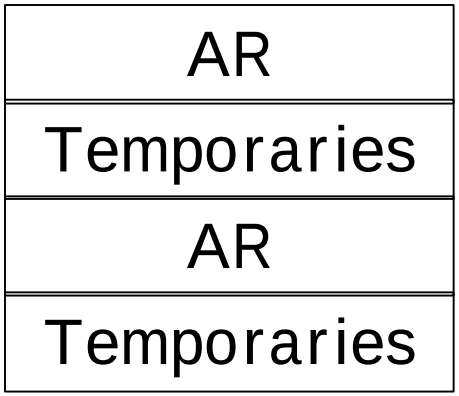
- Note a function argument may be evaluated 0 or more times
  - Evaluated 0 times if the argument is never used
- Call-By-Name is used in lazy functional languages
- There are also other, less important, parameter passing mechanisms

Allocating Temporaries in the AR

Topic III

Review

- The stack machine has activation records and intermediate results interleaved on the stack



Review (Cont.)

- Advantage: Very simple code generation
- Disadvantage: Very slow code
  - Storing/loading temporaries requires a store/load and $sp adjustment

A Better Way

- Idea: Keep temporaries in the AR
- The code generator must assign a location in the AR for each temporary

Example

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
                fib(x - 1) + fib(x – 2)
```

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values?

How Many Temporaries?

- Let NT(e) = # of temps needed to evaluate e
- NT(e_1 + e_2)
  - Needs at least as many temporaries as NT(e_1)
  - Needs at least as many temporaries as NT(e_2) + 1
- Space used for temporaries in e_1 can be reused for temporaries in e_2

The Equations

$$NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$$
$$NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$$
$$NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$$
$$NT(id(e_1,\ldots,e_n) = \max(NT(e_1),\ldots,NT(e_n))$$
$$NT(\text{int}) = 0$$
$$NT(\text{id}) = 0$$

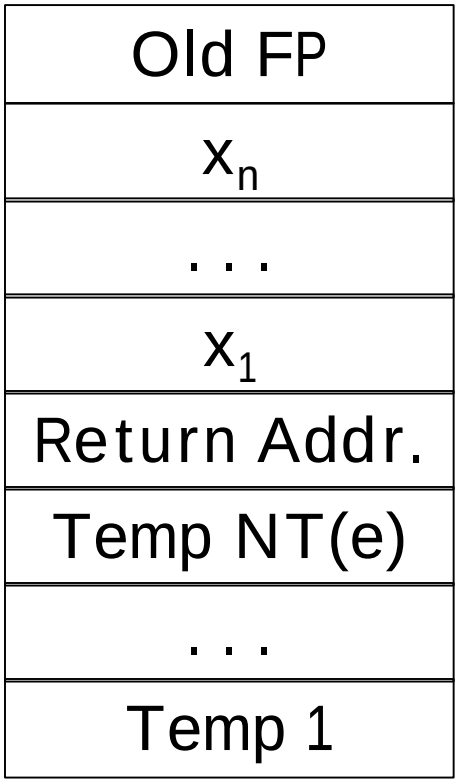
Is this bottom-up or top-down?

What is NT(…code for fib…)?

The Revised AR

- For a function definition $f(x_1,\ldots,x_n) = e$ the AR has $2 + n + NT(e)$ elements
  - Return address
  - Frame pointer
  - n arguments
  - NT(e) locations for intermediate results

Picture



Revised Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation: the position of the next available temporary

Code Generation for + (original)

cgen(e_1 + e_2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Code Generation for + (revised)

cgen(e_1 + e_2, nt) =

```
    cgen(e1, nt)
    sw $a0 nt($fp)
    cgen(e2, nt + 4)
    lw $t1 nt($fp)
    add $a0 $t1 $a0
```

Notes

- The temporary area is used like a small, fixed-size stack

- Exercise: Write out cgen for other constructs