Code Generation & Parameter Passing

Lecture Outline

1. Allocating temporaries in the activation record
   - Let's optimize our code generator a bit
2. A deeper look into calling sequences
   - Caller/Callee responsibilities
3. Parameter passing mechanisms
   - Call-by-value
   - Call-by-reference
   - Call-by-value-result
   - Call-by-name
   - Call-by-need

Extra Material in the Appendix (not covered in lecture)

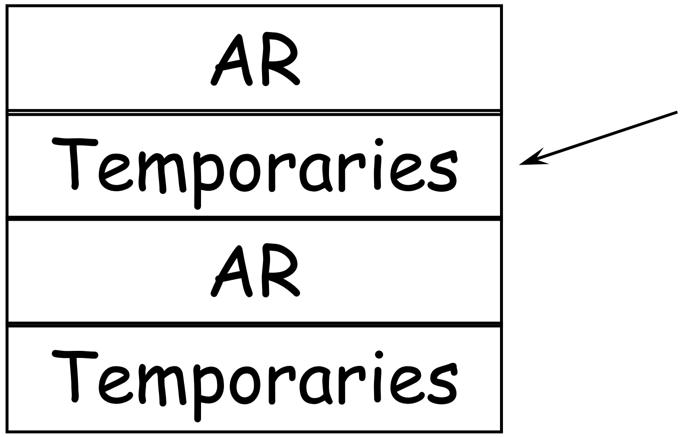
4. Code generation for OO languages
   - Object memory layout
   - Dynamic dispatch
5. Code generation of data structure references
   - Address calculations
   - Array references
6. Code generation for logical expressions
   - Short-circuiting

An Optimization: Temporaries in the Activation Record

Topic 1

Review

- The stack machine has activation records and intermediate results interleaved on the stack
- The code generator must assign a location in the AR for each temporary



These get put here when we evaluate compound expressions like $e_1 + e_2$ (need to store e_1 while evaluating e_2)

Review (Cont.)

- Advantage: Simple code generation
- Disadvantage: Slow code
  - Storing/loading temporaries requires a store/load and $sp adjustment

```
cgen(e1 + e2) = cgen(e1)             ; eval e1
                sw $a0 0($sp)        ; save its value
                addiu $sp $sp -4     ; adjust $sp (!)
                cgen(e2)             ; eval e2
                lw $t1 4($sp)        ; get e1
                add $a0 $t1 $a0      ; $a0 = e1 + e2
                addiu $sp $sp 4      ; adjust $sp (!)
```

An Optimization

- Idea: Predict how $sp will move at run time
  - Do this prediction at compile time
  - Move $sp to its limit, at the beginning

- The code generator must *statically* assign a location in the AR for each temporary

Improved Code

Old method

$cgen(e_1 + e_2)$ =

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

New idea

$cgen(e_1 + e_2)$ =

```
    cgen(e1)
    sw $a0 ?($fp)

    cgen(e2)
    lw $t1 ?($fp)
    add $a0 $t1 $a0
```

statically allocate

Example

```
add(w,x,y,z)
begin
    x + (y + (z + (w + 42)))
end
```

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

How Many Stack Slots?

- Let $NS(e)$ = # of slots needed to evaluate e
  - *Includes* slots for arguments to functions

- E.g: $NS(e_1 + e_2)$
  - Needs at least as many slots as $NS(e_1)$
  - Needs at least one slot to hold e_1, plus as many slots as $NS(e_2)$, i.e. $1 + NS(e_2)$

- Space used for temporaries in e_1 can be reused for temporaries in e_2

The Equations for the "Mini Bar" Language

$NS(e_1 + e_2) = \quad \max(NS(e_1), 1 + NS(e_2))$

$NS(e_1 - e_2) = \quad \max(NS(e_1), 1 + NS(e_2))$

$NS(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$

$\qquad \max(NS(e_1), 1 + NS(e_2), NS(e_3), NS(e_4))$

$NS(f(e_1,\ldots,e_n)) =$

$\qquad \max(NS(e_1), 1 + NS(e_2), 2 + NS(e_3), \ldots, (n-1) + NS(e_n), n)$

$NS(\text{int}) = \quad 0$

$NS(\text{id}) = \quad 0$

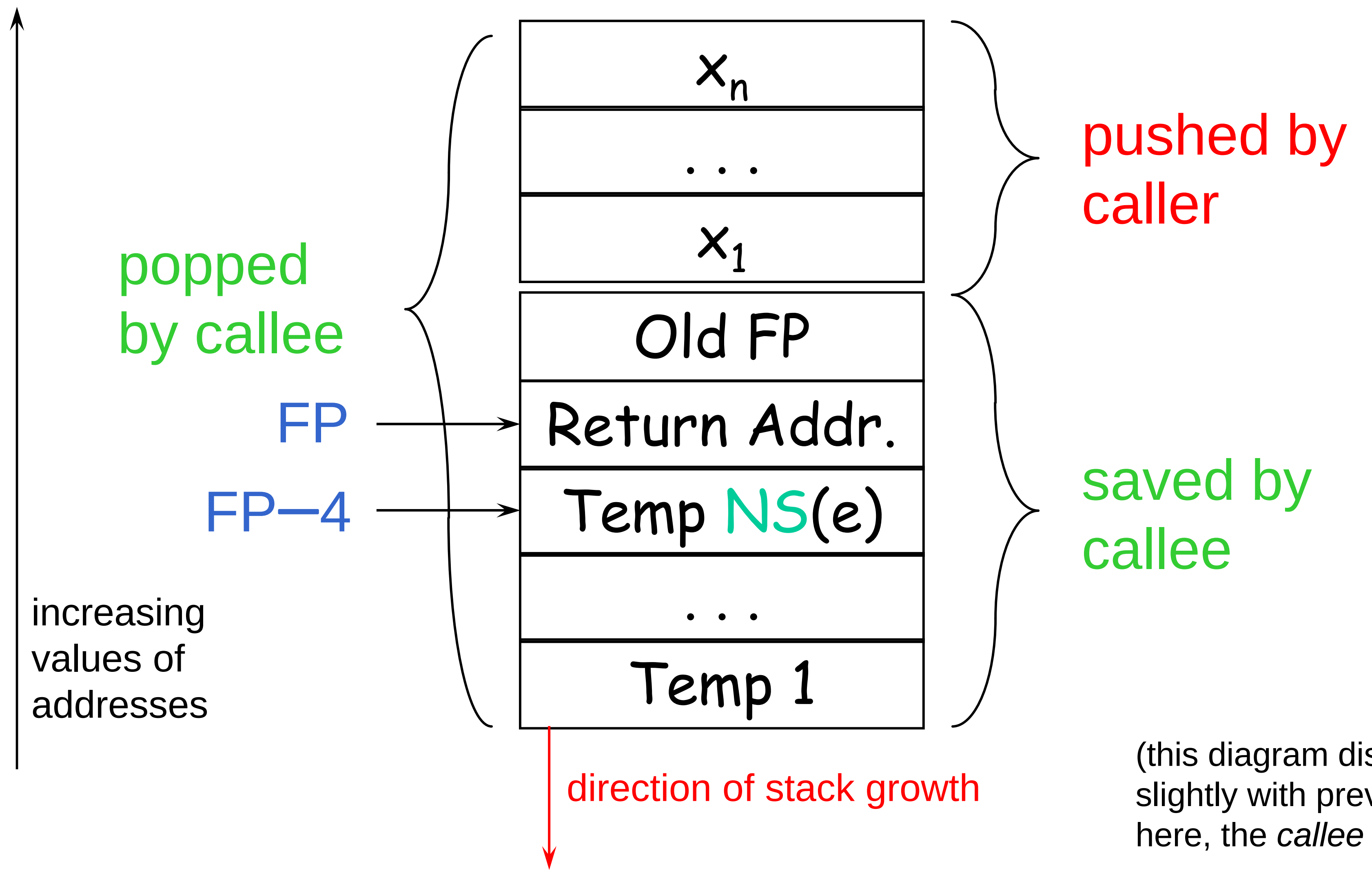
Rule for $f(e_1, \ldots, e_n)$: Each time we evaluate an argument, we put it on the stack

The Revised Activation Record

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

- Note that f's arguments are now considered to be part of its *caller's* AR

Picture: Activation Record



(this diagram disagrees slightly with previous lecture: here, the *callee* saves FP)

Revised Code Generation

- Code generation must know how many slots are in use at each point

- Add a new argument to code generation: the position of the *next available* slot

Improved Code

Old method

$cgen(e_1 + e_2) =$

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

New method

$cgen(e_1 + e_2, \underline{ns}) =$

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

compile-time prediction

static allocation

Notes

- The slots for temporary values are still used like a stack, but we predict usage at compile time
  - This saves us from doing that work at run time
  - Allocate all needed slots at start of a function

Exerc. Write some code which runs *slower* after performing the optimization just presented
  - Hint: Think about memory usage (& caches, etc.)

A Deeper Look into Calling Sequences

Topic 2

Handling Procedure Calls and Returns

Calling sequence: a code sequence that sets up a procedure call

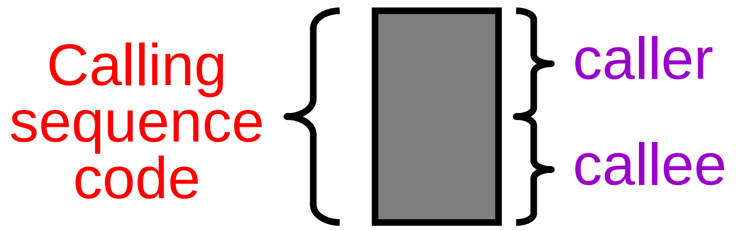
- allocates an activation record (model-dependent)
- loads actual parameters
- saves machine state (return address, etc.)
- transfers control to callee

Return sequence: a code sequence that handles the return from a procedure call

- deallocates the activation record
- sets up return value (if any)
- restores machine state (stack pointer, PC, etc.)

Calling Sequences: Division of Responsibilities

- The code in a calling sequence is often divided up between the caller and the callee



- If there are *m* calls to a procedure, the instructions in the caller's part of the calling sequence are repeated *m* times, while the callee's part is repeated exactly once
  - This suggests that, for smaller code size, we should try to put as much of the calling sequence as possible in the callee
  - However, it may be possible to carry out more call-specific optimization by putting more of the code into the caller instead of the callee

Calling Sequences: Layout Issues

General rule of thumb:

Fields that are fixed early, are placed near the middle of the activation record

- The caller has to evaluate the actual parameters, and retrieve the return value
  - these fields should be located near the caller’s activation record
- The callee has to fill in machine status fields so that the callee can restore state on return
  - the caller should have easy access to this part of the callee’s activation record

Calling/Return Sequences: Typical Actions

Typical calling sequence:

1. caller evaluates actuals; pushes them on the stack
2. caller saves machine status on the stack (in the callee's AR) and updates the stack pointer
3. caller transfers control to the callee
4. callee saves registers, initializes local data, and begins execution

Typical return sequence:

1. callee stores return value in the appropriate place
2. callee restores registers and old stack pointer
3. callee branches to the return address

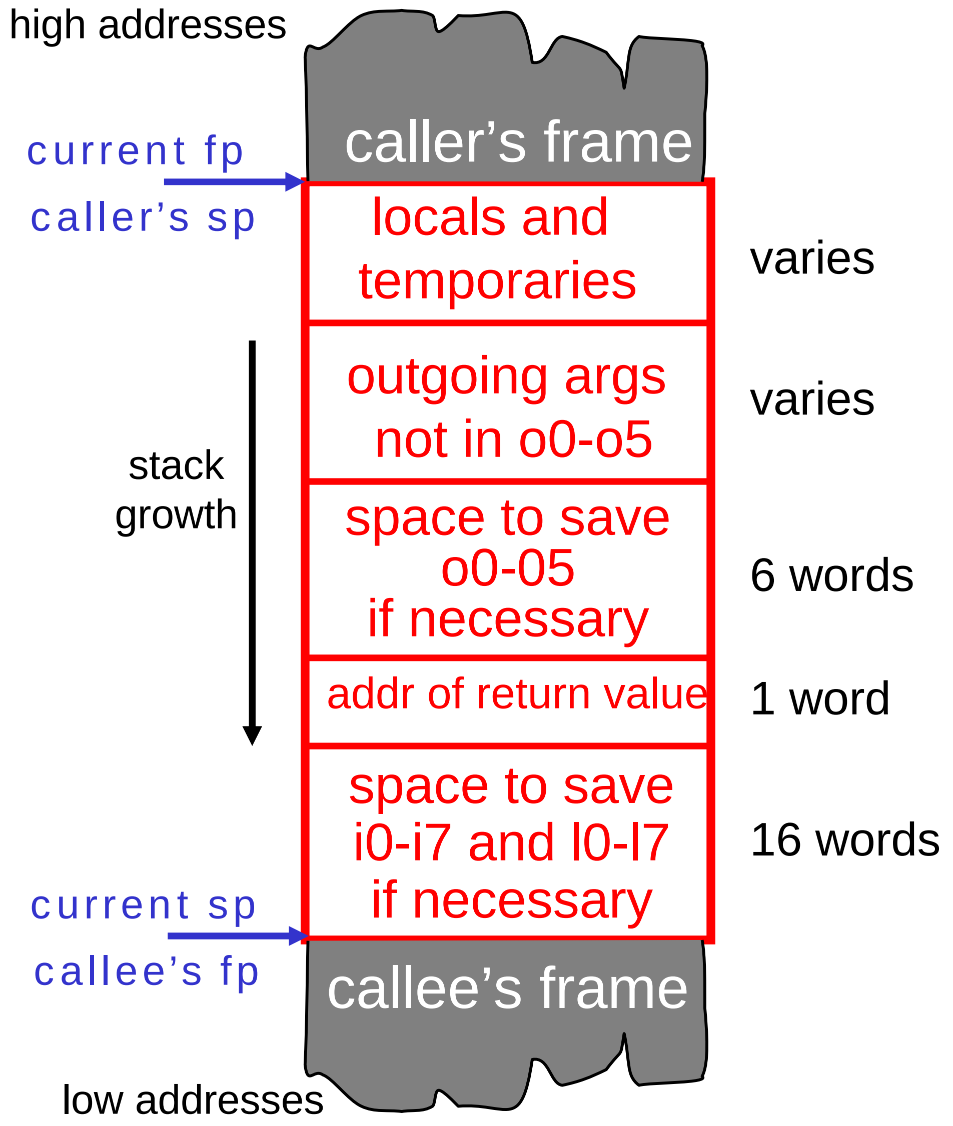
Example Activation Record: The SPARC

Registers

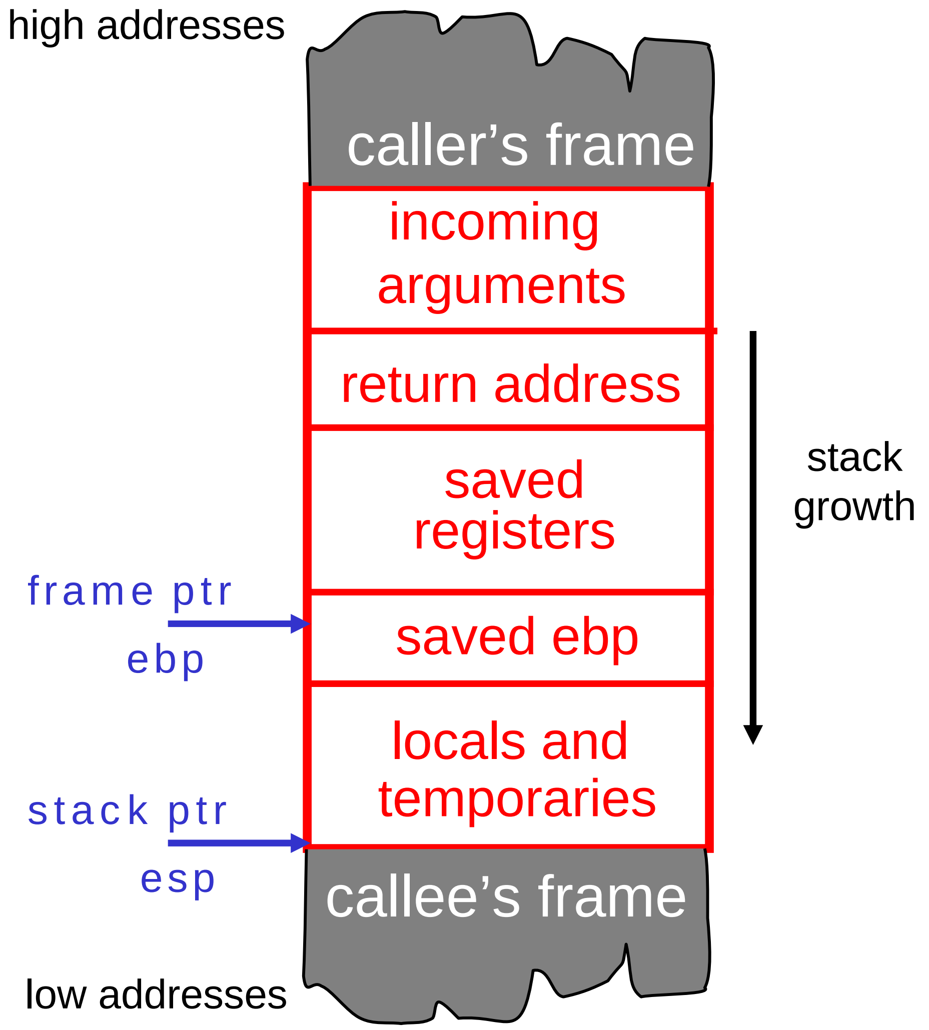
g0-g7	global registers
o0-o7	outgoing args
l0-l7	local registers
i0-i7	incoming args

function return address

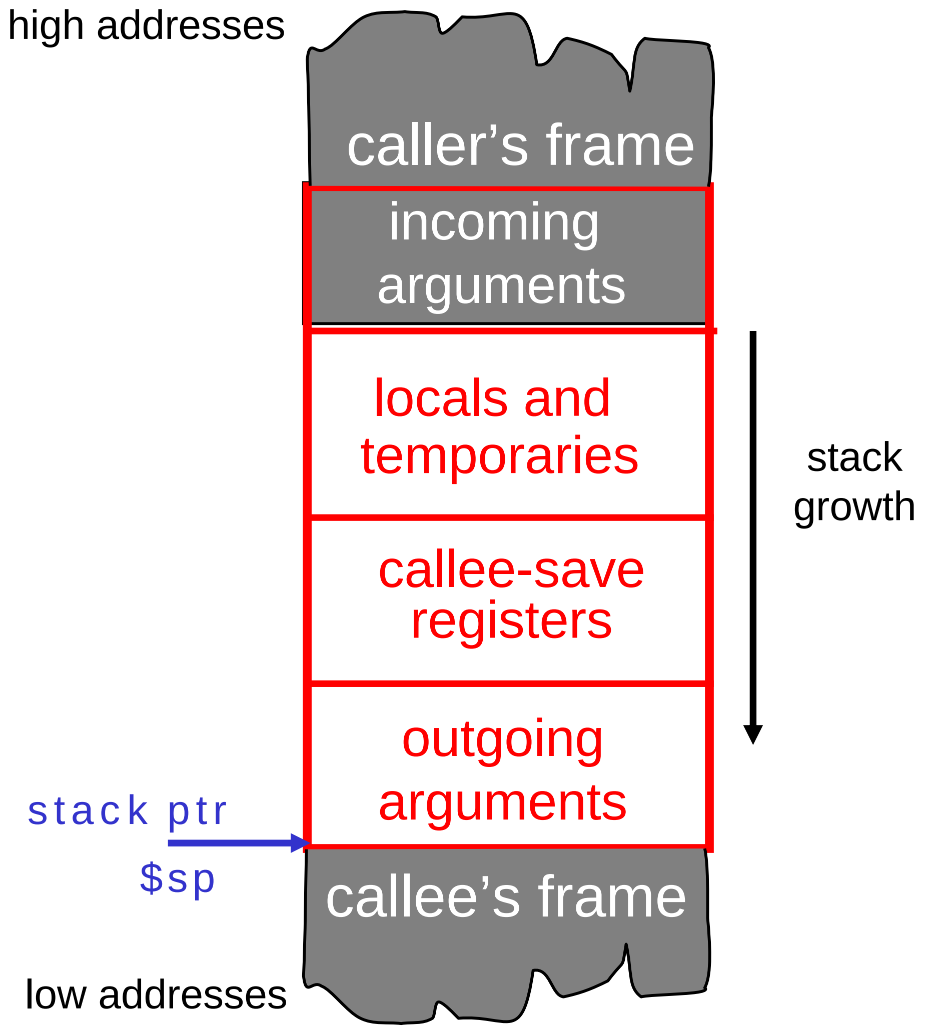
caller's o7/callee's i7



Example Activation Record: Intel x86



Example Activation Record: MIPS R3000



Parameter Passing Mechanisms

Topic 3

Parameter Passing Mechanisms

- There are many semantic issues in programming languages centering on *when* values are computed and the scopes of *names*
  - Evaluation is the heart of computation
  - Names are most primitive abstraction mechanism

- We will focus on parameter passing
  - *When* are arguments of function calls evaluated?
  - *What* are formal parameters bound to?

Parameter Passing Mechanisms (Cont.)

First, an issue not discussed much…

Order of argument evaluation

- "Usually" not important for the execution of a program
- However, in languages that permit side-effects in call arguments, different evaluation orders may give different results

  e.g. a call `f(++x,x)` in C
- A "standard" evaluation order is then specified

  C compilers typically evaluate their arguments right-to-left. Why?

Call-by-value

C uses call-by-value everywhere (except macros...)
Default mechanism in Pascal and in Ada

```
callByValue(int y)
{
    y = y + 1;
    print(y);
}

main()
{
    int x = 42;
    print(x);
    callByValue(x);
    print(x);
}
```

output:

```
x = 42
y = 43
x = 42
```

x's value does *not* change when y's value is changed

Call-by-reference

Available in C++ with the '&' type constructor
(and in Pascal with the **var** keyword)

```
callByRef(int &y)
{
    y = y + 1;
    print(y);
}

main()
{
    int x = 42;
    print(x);
    callByRef(x);
    print(x);
}
```

output:
```
  x = 42
  y = 43
  x = 43
```

x's value changes when y's value is changed

Call-by-reference can be faked with pointers

C++:

```
callByRef(int &y)
{
    y = y + 1;
    print(y);
}

main()
{
    int x = 42;
    print(x);
    callByRef(x);
    print(x);
}
```

C:

```
fakeCallByRef(int *y)
{
    *y = *y + 1;
    print(*y);
}

main()
{
    int x = 42;
    print(x);
    fakeCallByRef(&x);
    print(x);
}
```

must explicitly pass the address of a local variable

Pointers to fake call-by-reference (cont.)

- It's not *quite* the same
  - A pointer can be reassigned to point at something else; a C++ reference cannot
- The pointer itself was passed by value
- This is how arrays (they are implicitly pointers) and structures are passed in C

Call-by-value-result

Available in Ada for **in out** parameters
(code below in C syntax)

```
callByValueResult(int y, int z)
{
    y = y + 1;  z = z + 1;
    print(y);   print(z);
}

main()
{
    int x = 42;
    print(x);
    callByValueResult(x, x);
    print(x);
}
```

output:

```
x = 42
y = 43
z = 43
x = 43
```

Note that x's value is *different* from both using call-by-value and call-by-reference

What about Java?

- Primitive types (int, boolean, etc.) are always passed by value
- Objects are not quite -by-value nor -by-reference:
  - If you reassign an object reference, the caller's argument does not get reassigned (like -by-value)
  - But if the object referred-to is modified, that modification is visible to the caller (like -by-reference)
- It's really ordinary call-by-value with pointers, but the pointers are not syntactically obvious

Implementing Parameter Passing

Call-by-value (easy, no special compiler effort)

The arguments are evaluated at the time of the call and the value parameters are copied and either

- behave as *constant values* during the execution of the procedure (i.e., cannot be assigned to as in Ada), or
- are viewed as initialized *local* variables (in C or in Pascal)

Call-by-reference

The arguments must have allocated memory locations

The compiler passes the address of the variable, and the parameter becomes an *alias* for the argument

Local accesses to the parameter are turned into *indirect accesses*

Implementing Parameter Passing (Cont.)

Call-by-value-result

The arguments are evaluated at call time and the value parameters are copied (as in call-by-value) and used as a local variables

The final values of these variables are copied back to the location of the arguments when the procedure exits (note that the activation record cannot be freed by the callee!)

Issues left unspecified:

- the order in which the results are copied back
- whether the locations of the arguments are calculated only on entry and stored, or whether they are recalculated on exit

Call-by-name

- Whole different ballgame: it's like passing the *text* of the argument expression, unevaluated
  - The text of the argument is viewed as a function in its own right
  - Also passes the environment, so free variables are still bound according to rules of static scoping
- The argument is not evaluated until it is actually used, *inside* the callee
  - Might not get evaluated at all!

- An optimized version of call-by-name is used in some functional languages (e.g. Haskell, Miranda, Lazy-ML) under the names lazy evaluation (or call-by-need)

Call-by-name example (in "C++ Extra")

code + environment
(env has just 'x' here)

```
callByName(int closure y)
{
  print(y);
  print(y);   // => print(x = x+1)
}

main()
{
  int x = 42;
  print(x);
  callByName( [[ x = x+1 ]] );
  print(x);
}
```

eval

closure

output:

```
x = 42
y = 43
y = 44
x = 44
```

both evals have side effects

x's value changes when y is evaluated

Code Generation for OO Languages

Topic 4

(probably not covered in lecture)

Object Layout

- Object-Oriented (OO) code generation and memory layout

- OO Slogan: If C (child) is a subclass of P (parent), then an instance of class C can be used wherever an instance of class P is expected

- This means that P's methods should work with an instance of class C

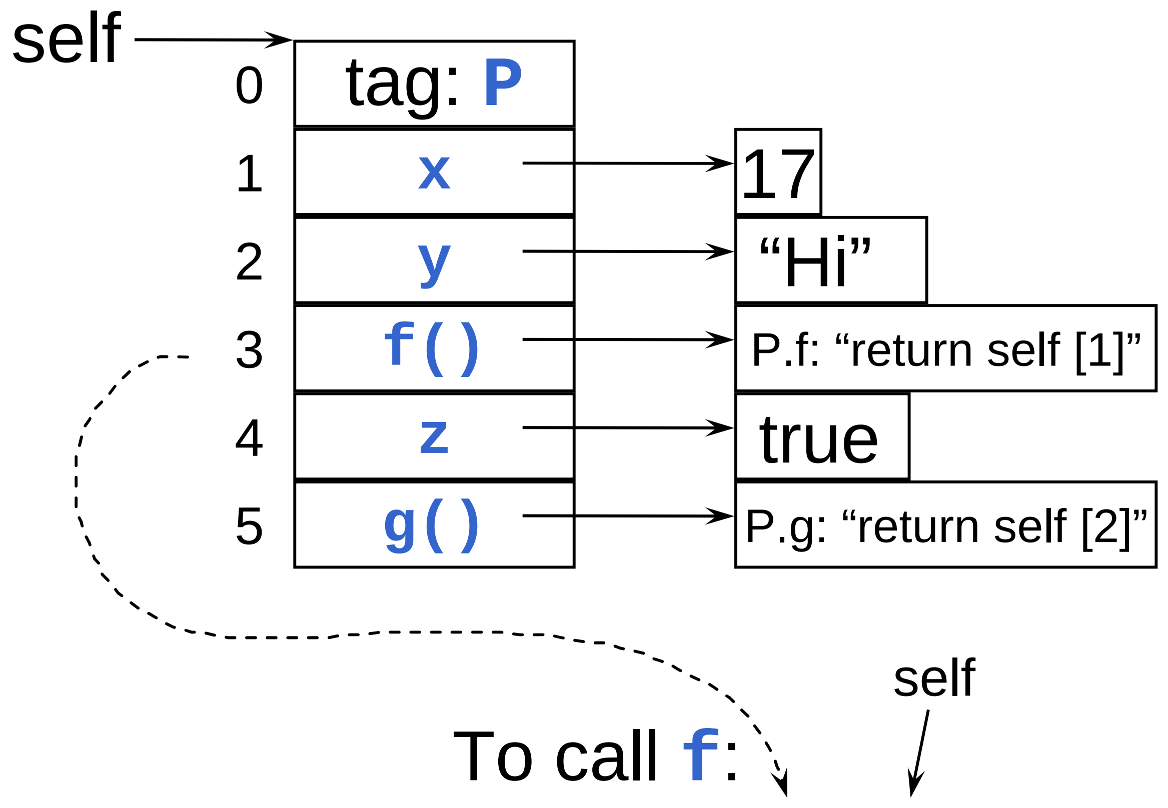
Two Issues

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

Object Representation

```
class P {
   x : Int <- 17;
   y : String <- "Hi";
   f() : Int { x };
   z : Bool <- true;
   g() : String { y };
};
```

- Why method pointers?
- Why the tag?



To call **f**:

```
lw $t1 12($s0)
jalr $t1
```

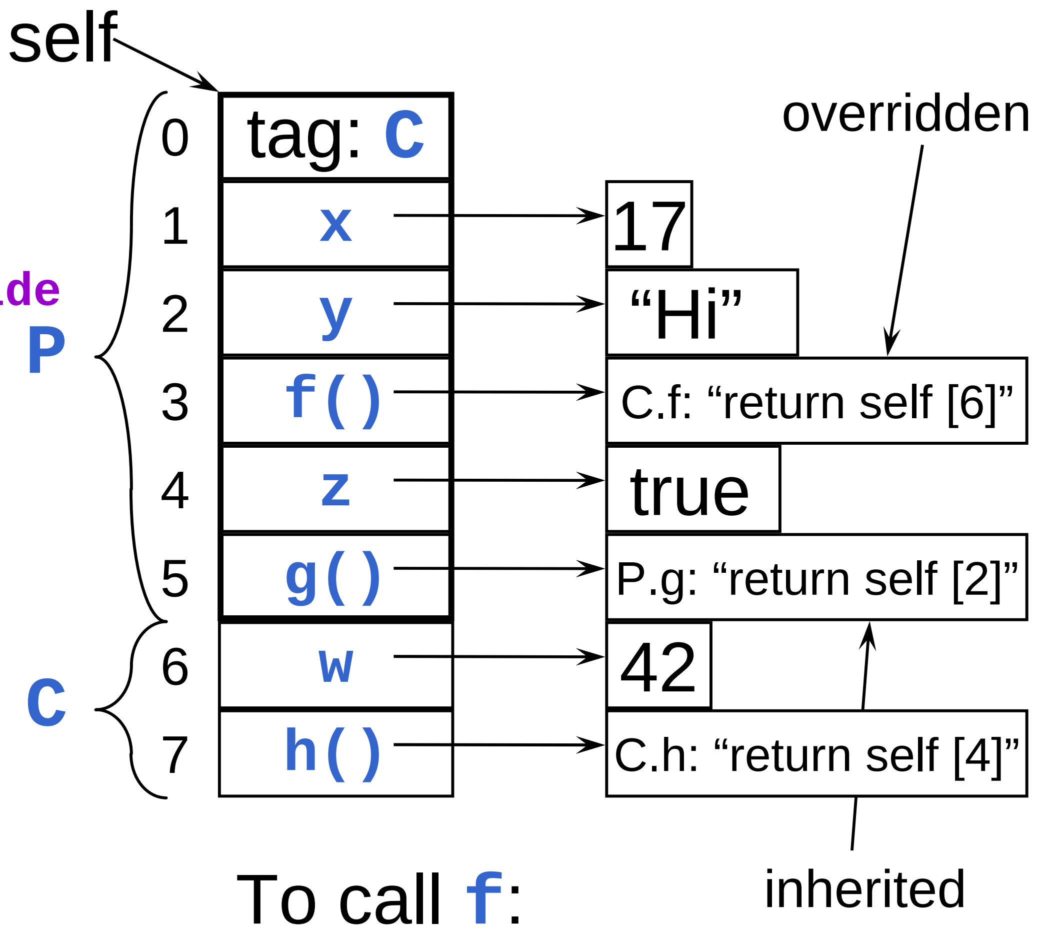
dynamic dispatch
“case”

Subclass Representation

```
class P { ..(same).. };

class C inherits P {
   w : Int <- 42;     // new
   f() : Int { w }; // override
   h() : Bool { z };// new
};
```

Idea: Append new fields

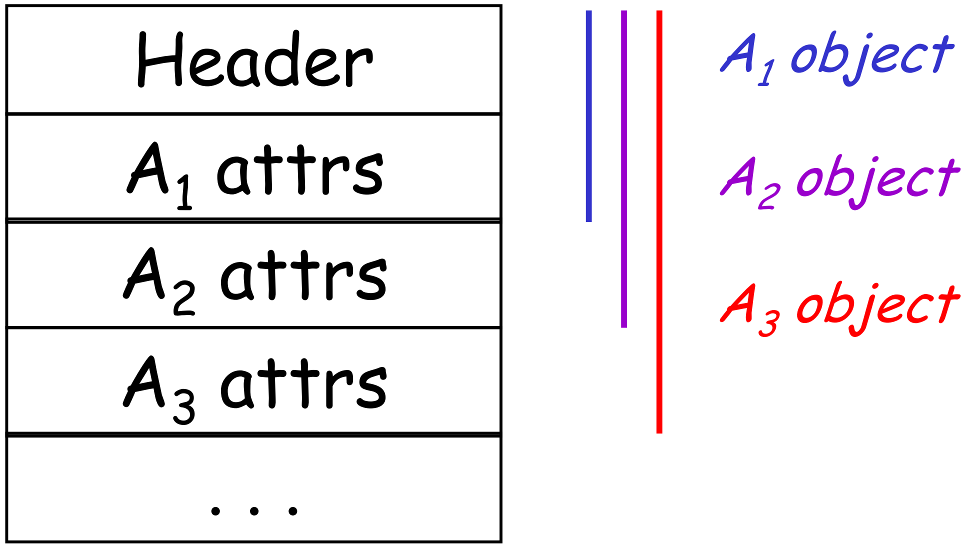


To call f:

```
lw $t1 12($s0)
jalr $t1
```

Subclasses (Cont.)

- The offset for an attribute 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 $A_n < \ldots < A_3 < A_2 < A_1$



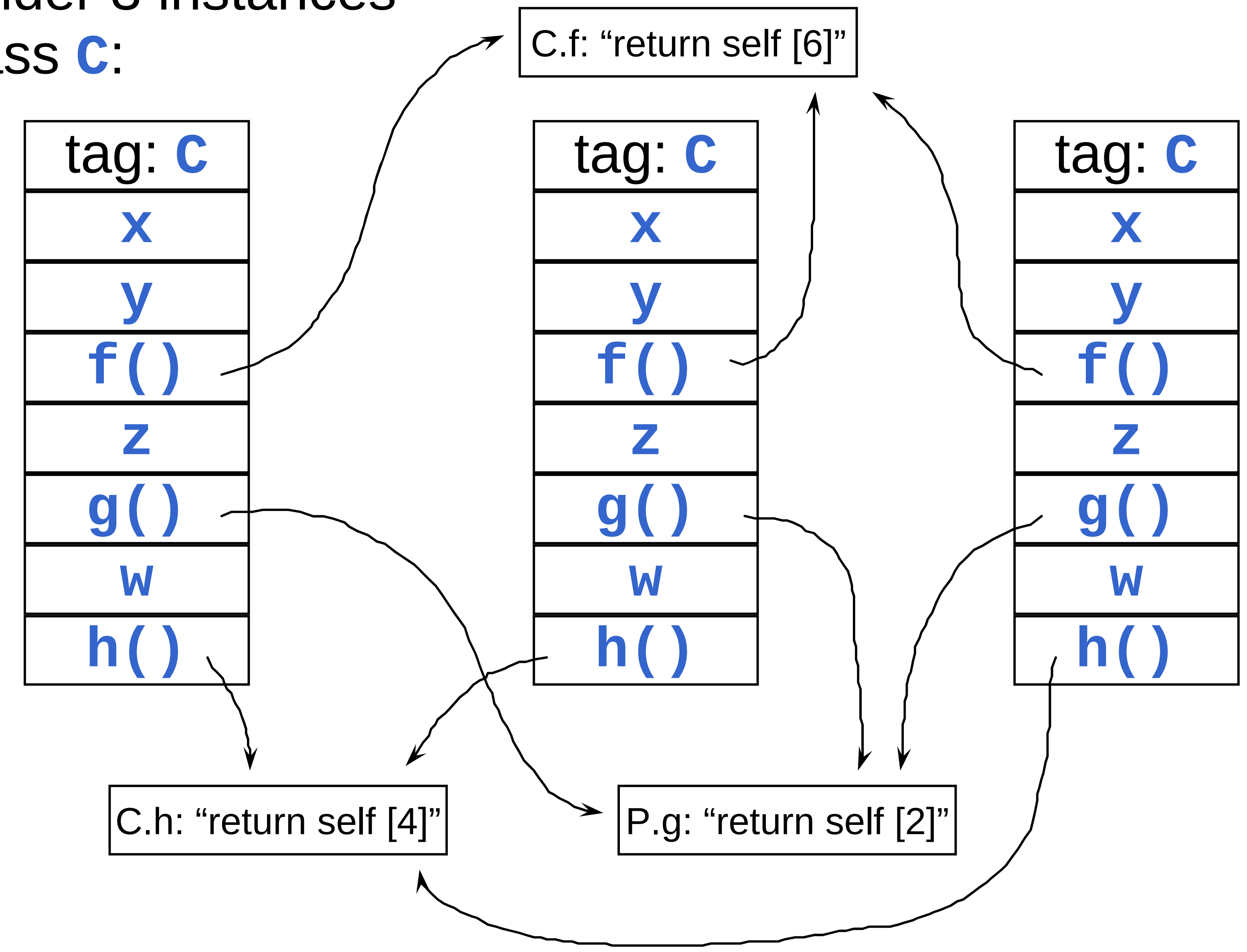
What about multiple inheritance?

What's the point?

- Simple
  - Just append subclass fields
- Efficient
  - Code can ignore dynamic type – just act as if it is the static type
- Supports overriding of methods
  - Just replace the appropriate dispatch pointers
- We implement type conformance (compile-time concept) with representation conformance (run-time concept)

An Optimization: Dispatch Tables

Consider 3 instances of class C:



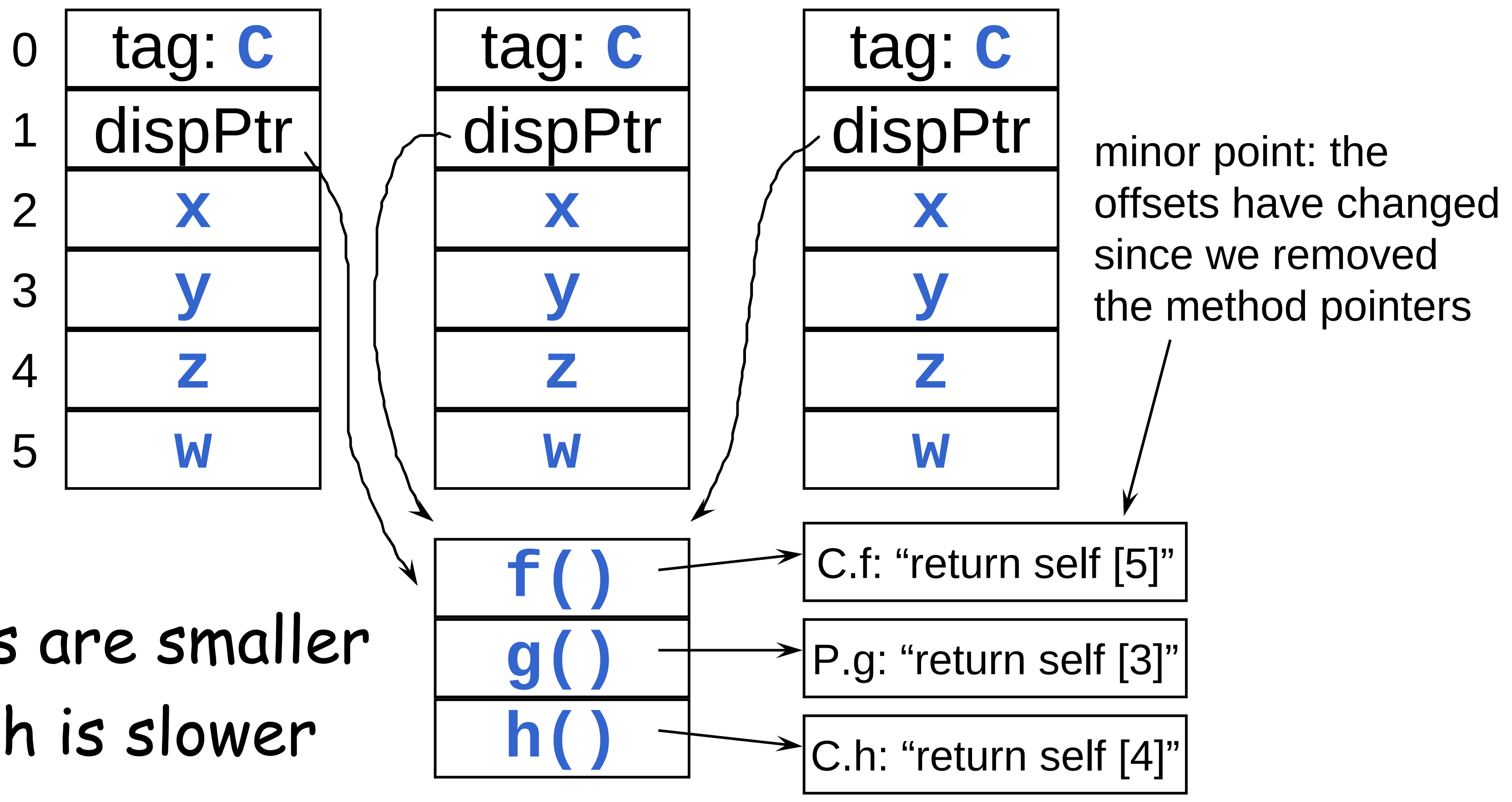
Observation

- Every instance of a given class has the same values for all of its method pointers

- Space optimization: Put all method pointers for a given class into a common table, called the "dispatch table"
  - Each instance has a pointer to the dispatch table

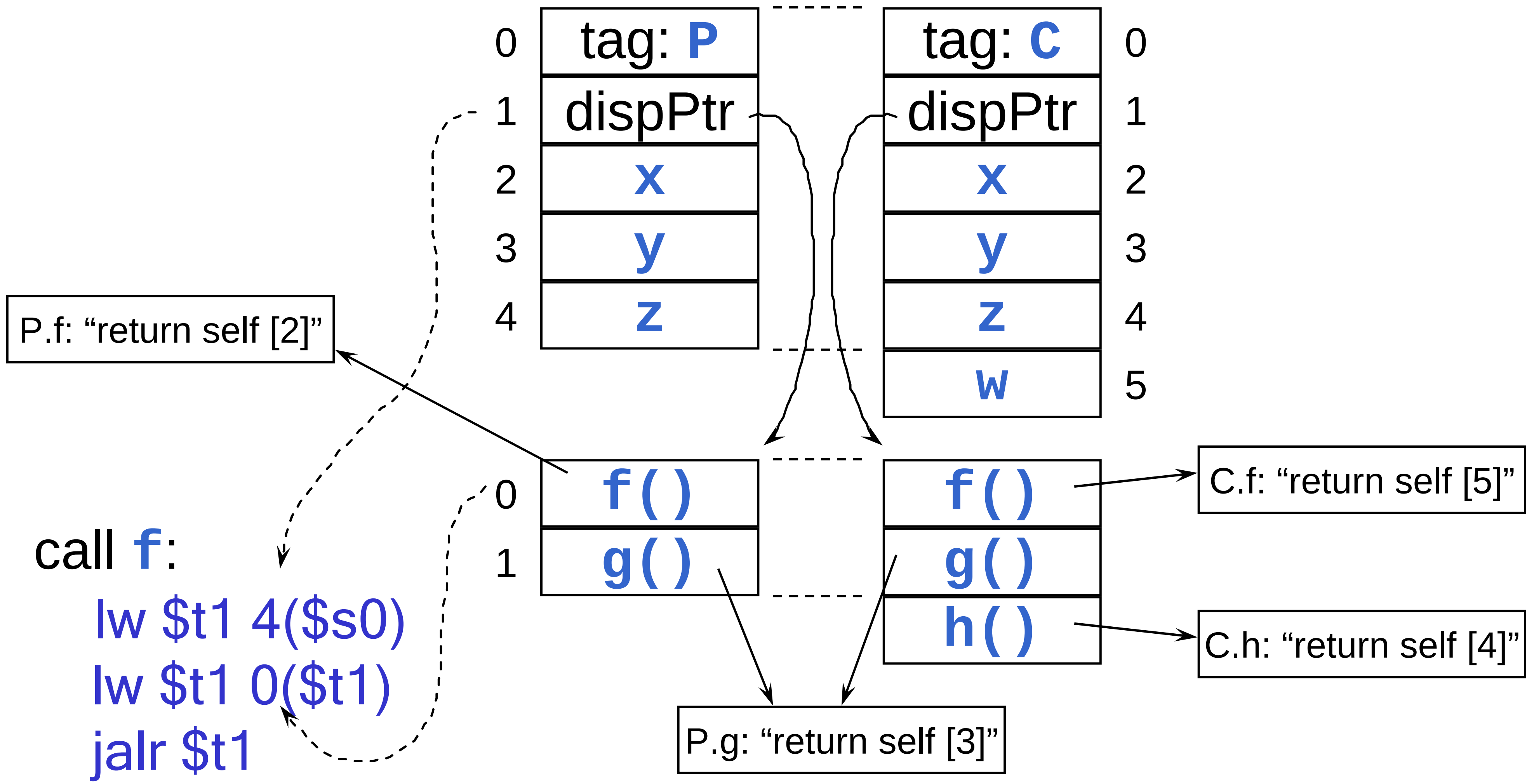
Picture with Dispatch Table

- Consider again 3 instances of C:



- Objects are smaller
- Dispatch is slower

Subclassing, again

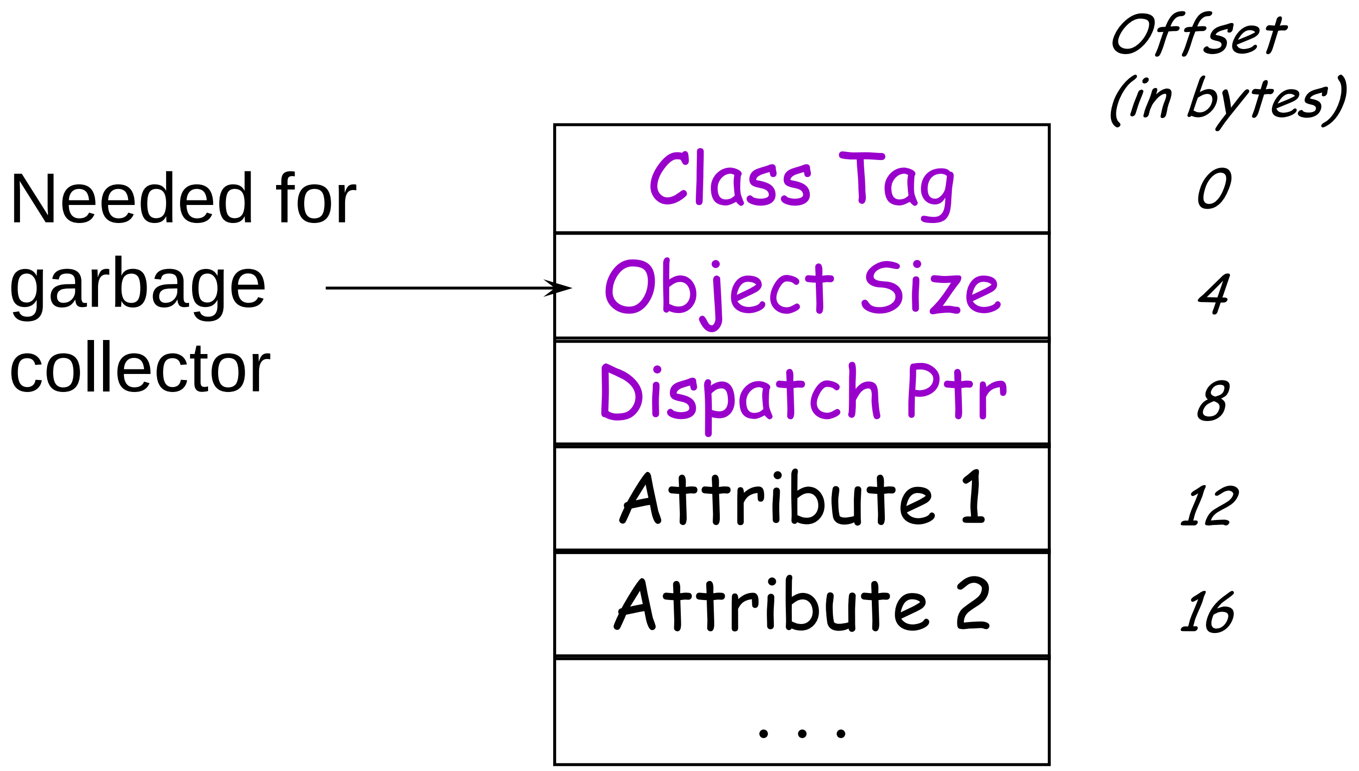


call **f**:

```
lw $t1 4($s0)
lw $t1 0($t1)
jalr $t1
```

Real Object Layout

- Actually, the first *3* words of objects contain header information:



Summary of Dispatch Tables

Pulled method pointers out, into separate table

- Makes objects smaller
- Makes (dynamic) dispatch slower

Q: Why don't we do this for attributes?

Exerc. Write some code that is <u>slower</u> with dispatch tables (instead of embedded method pointers)

Exerc. Write some code that is <u>faster</u> with dispatch tables