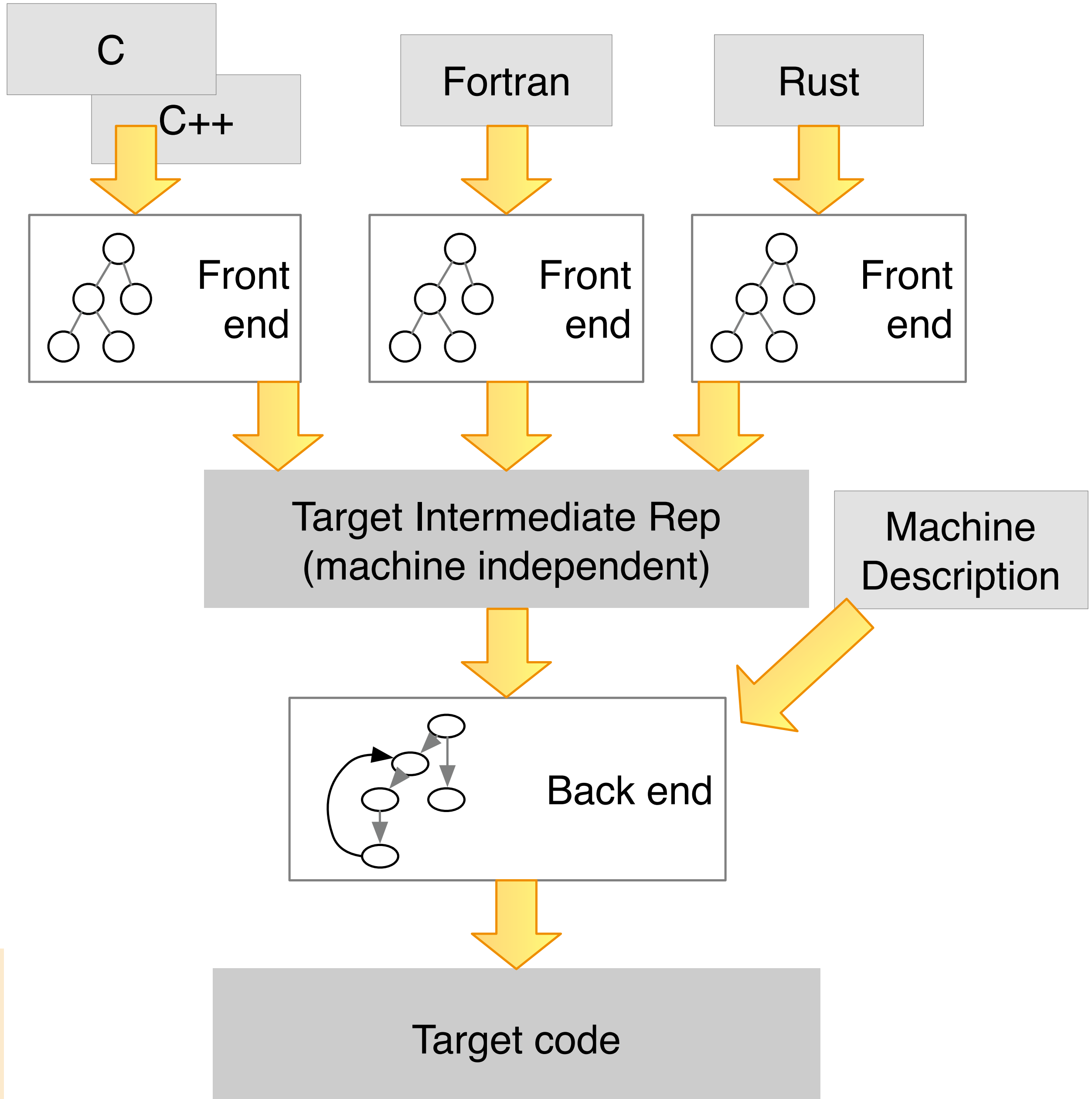
From Abstract Machine to Concrete Machine

Generating Assembly Code

(although we will generate C)

Intermediate & target code



Example back ends:
GCC, LLVM

"Sandboxed" languages look a little different ... VM instead of back end

Resources to Manage

Memory

Divided into text (code), stack, and heap

- Code: where the program code lives
- Stack: activation records; global data may also live here, or in its own area
- Heap: dynamically managed memory

Registers

General-purpose and dedicated

Untyped (except for floating-point)

– But llvm registers (an infinite supply) are sorta kinda typed

Registers and Memory

Memory (RAM) is large and slow

- Much, much slower than modern processors

Registers are few but fast

- They are *part of the processor*
- As few as 4 general purpose registers (x86)
- As many as 256 (Sparc)
- 16 to 32 is common
- additional special purpose registers (program counter, flags register, etc.)

RISC machines are load/store architectures

- No other operations on memory

Since we will generate C code, we will not do register allocation, but you should understand registers vs. memory.

Managing Registers and Memory

Each architecture (processor + OS) has a set of *calling conventions*

How do I allocate a stack frame?

How do I pass parameters and results?

Which registers are preserved?

- Preserved: Callee must save, or leave alone
- Volatile: Callee may clobber; caller can save

Conventions described in "application binary interface" document

Calling Convention - Skeleton

Before the call:

- Put parameters into registers, or on the stack
- Save any volatile registers with important stuff

On entry:

- Allocate a stack frame
- Save return address and old stack pointer

On exit

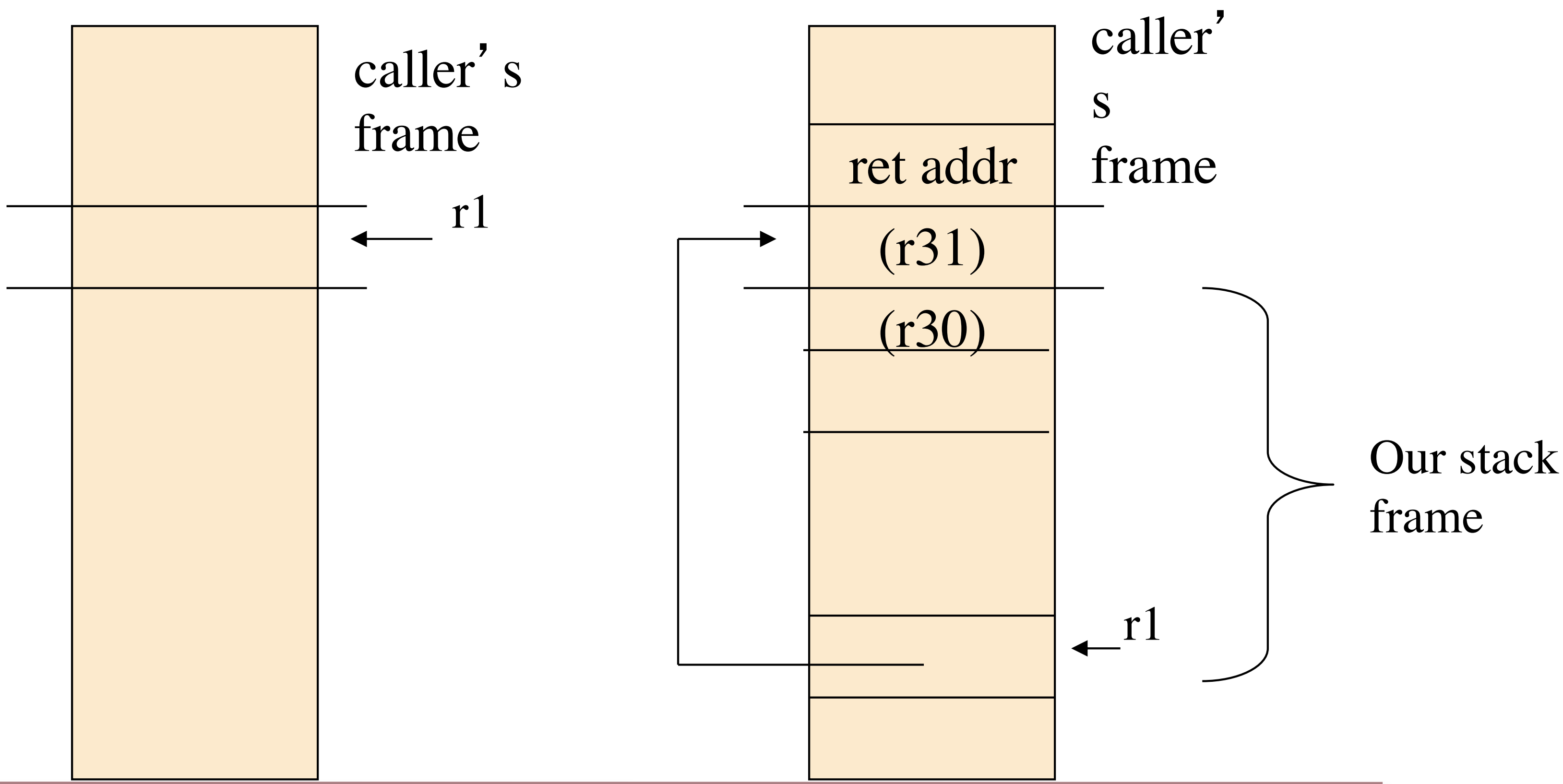
- Place return value in register (or on stack)
- Restore stack frame
- Jump to return address

In assembly language, we might write macros (like inline functions, or C preprocessor macros) to handle the call/return conventions.

Creating an Activation Record

PowerPC example. There are probably mistakes in this.

```
mflr r0              ; R0 <- return address
stmw r30,-8(r1)      ; Save R30 and R31in my area
stw  r0,8(r1)        ; Save return address into caller frame
stwu r1,-64(r1)      ; Allocate 64 bytes and save old sp
mr r30,r1          ; R30 <- R1  (huh?  frame pointer?)
```

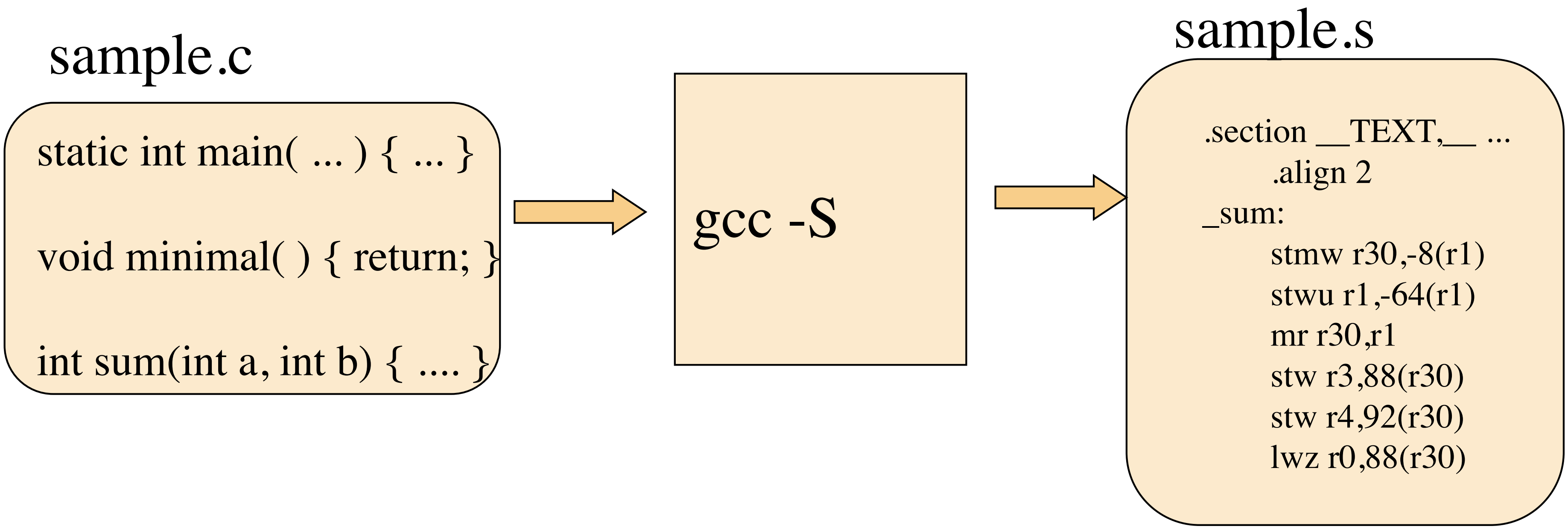


Calling Conventions

Basic calling conventions are mostly language independent

Though you' ll need a couple of implicit parameters, like "self-object"

Gcc will spill the beans with -S option



Calling Conventions: LLVM

Similar trick as gcc -S:

- Use Clang or other llvm-targeted compiler to generate llvm intermediate code
- Identify boilerplate and idioms

> Generating C, we won't need this. We will translate a method call into a C function call.
>
> If you generate assembly code, for this project or others, gcc –S is your friend. Write tiny functions with variations.

Tactic for Identifying Boilerplate

Two procedures

- One does almost nothing
  - and so should contain almost only boilerplate
- One does a little something
  - and so should have some code besides boilerplate

Compare side-by-side

- What is boilerplate on entry?
- What is boilerplate on exit?

PowerPC (old Mac OS X)

```
int sum(int a, int b) {
  int c, d, e;
  c = a;
  d = b;
  e = a + b;
  return  e;
}

int nothing() {
  return;
}
```

```
section __TEXT,__text, ....,   .align 2
_sum:
    stmw r30,-8(r1)
    stwu r1,-64(r1)
    mr r30,r1
    stw r3,88(r30)
    stw r4,92(r30)
    ... code in sum but not in nothing ...
    lwz r0,40(r30)
    mr r3,r0
    lwz r1,0(r1)
    lmw r30,-8(r1)
    blr
    .align 2
    .globl _nothing
.section
    __TEXT,__text,regular,pure_instructions
    .align 2
_nothing:
```

Sparc (old ix)

```
int sum(int a, int b) {
        int c;
        c = a + b;
        return c;
}
```

```
        .section       ".text"
        .align 4
        .global sum
        .type   sum, #function
        .proc   04
sum:
        save    %sp, -120, %sp
        st      %i0, [%fp+68]
        st      %i1, [%fp+72]
        ld      [%fp+68], %i5
        ld      [%fp+72], %g1
        add     %i5, %g1, %g1
        st      %g1, [%fp-20]
        ld      [%fp-20], %g1
        mov     %g1, %i0
        ret
        restore
        .size   sum, .-sum
```

Shift register window (peculiar to sparc)

Note incredibly stupid code from gcc. Don't be too depressed by yours!

Branch delay slot (peculiar to sparc)

clang (llvm) on MacOS X, x86:

```c
int sum(int a, int b) {
  int c;
  c = a + b;
  return c;
}

int nothing() {
  return 42;
}
```

```
_sum:                        ## @sum
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp2:
        .cfi_def_cfa_offset 16
Ltmp3:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp4:
        .cfi_def_cfa_register %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -4(%rbp), %esi
        addl    -8(%rbp), %esi
        movl    %esi, -12(%rbp)
        movl    -12(%rbp), %eax
        popq    %rbp
        retq
        .cfi_endproc
```

Viewing llvm intermediate code

clang -S -flto -emit-llvm sample.c

more sample.s

When you see what clang or gcc emits, you won't feel so bad the code you generate.

```
; Function Attrs: nounwind ssp uwtable
define i32 @sum(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* %c, align 4
  %6 = load i32* %c, align 4
  ret i32 %6
}
```

Allocate space in stack

Copy args into stack (why?)

Load arg values into registers

Add (result in register)

Save into 'c' in stack

Load 'c' into a register

Return it

Addressing Data

Local variables

Offset from SP (stack pointer) or FP (frame pointer)

- Including variables declared in nested blocks. Do you see how?

Object fields

Offset from object pointer

- Keep "this" object in register. Implicit parameter?
- Others: object reference is pointer

Non-local

- "Display" or "lexical chain"
- (Not in Quack --- we don't have nested block scopes!)

In C:

Objects are structs ... fields are members of struct
Classes are structs ... methods are function pointers
Local variables are C variables (with generated names)

For each method ...

Calculate total size of frame

Save areas + links + local variables

- Easy in Quack: All local variables are 1 word (4 bytes); larger objects are pointers to heap

Associate address with each local

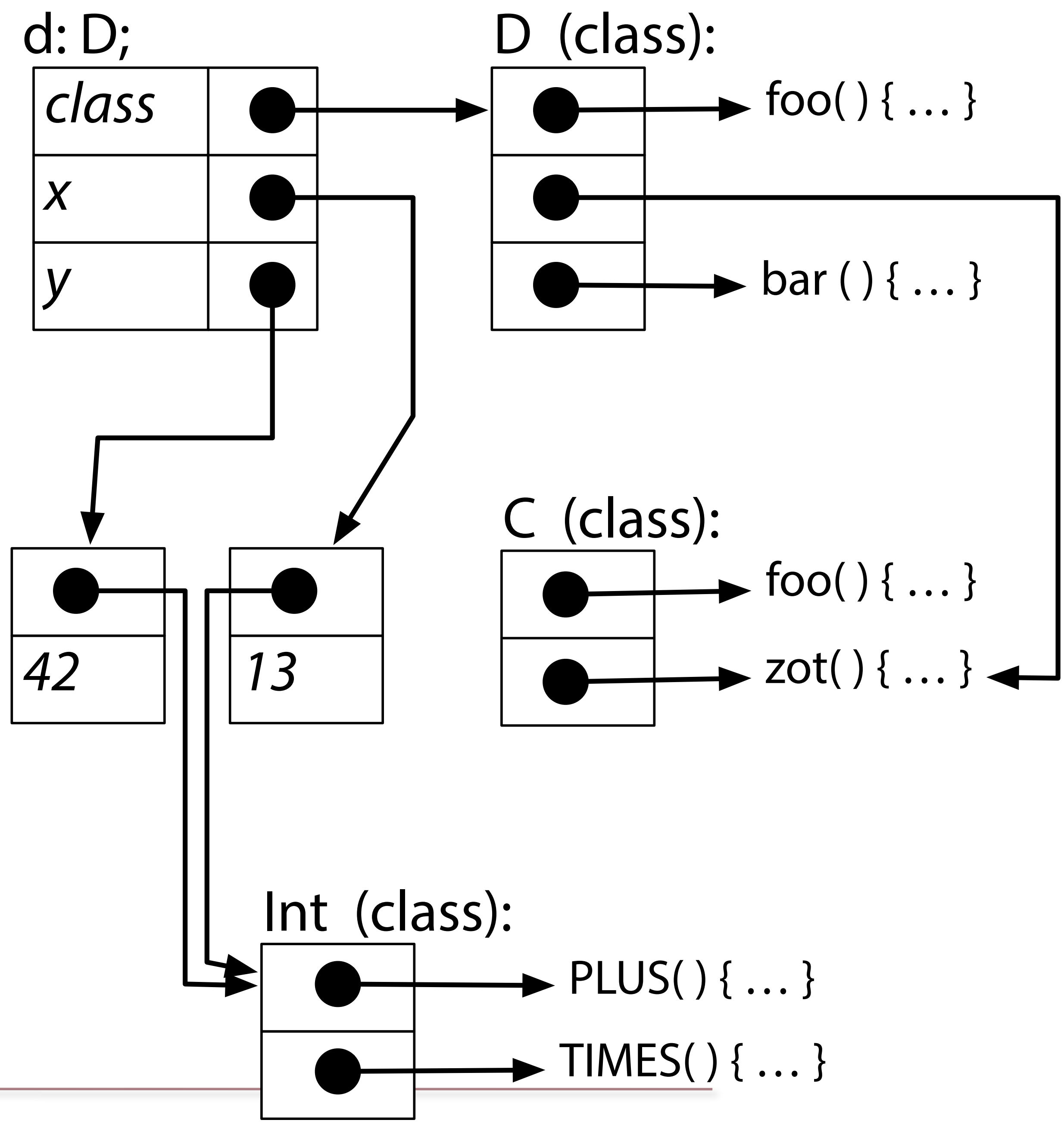
Offset within frame

- From stack pointer
- Or from frame pointer (fixed location in frame)

In C: We just create local variable in generated code.
gcc or llvm will calculate frame sizes for us

Addressing fields and methods

```
class C( ) {
    this.x = 42;
    def foo( ): Int {
          this.zot( );
    }
    def zot( ) { … }
}

class D() extends C {
  this.x = 43;
  this.y = 13;
  def foo( ): Int {
       this.zot( );
       this.bar( );
   }
  def bar( ) { … }
}
```

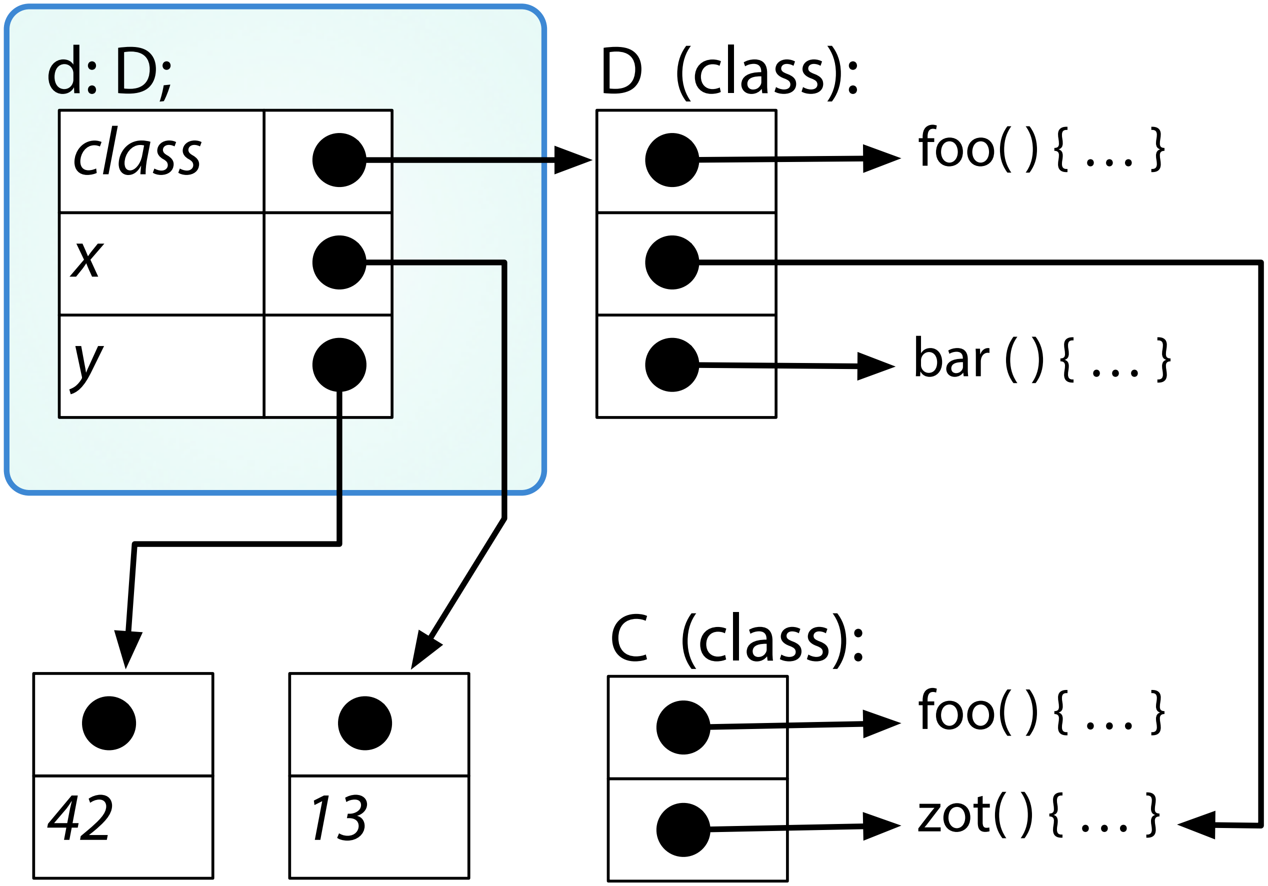


What does this really look like?

```
struct class_Pt_struct;
typedef struct class_Pt_struct* class_Pt;

typedef struct obj_Pt_struct {
  class_Pt  clazz;
  obj_Int x;
  obj_Int y;
} * obj_Pt;

struct  class_Pt_struct  the_class_Pt_struct;

struct class_Pt_struct {
  obj_Pt (*constructor) (obj_Int, obj_Int );
  obj_String (*STRING) (obj_Obj);
  obj_Pt (*PRINT) (obj_Pt);
  obj_Boolean (*EQUALS) (obj_Obj, obj_Obj);
  obj_Pt (*PLUS) (obj_Pt, obj_Pt);
};
```

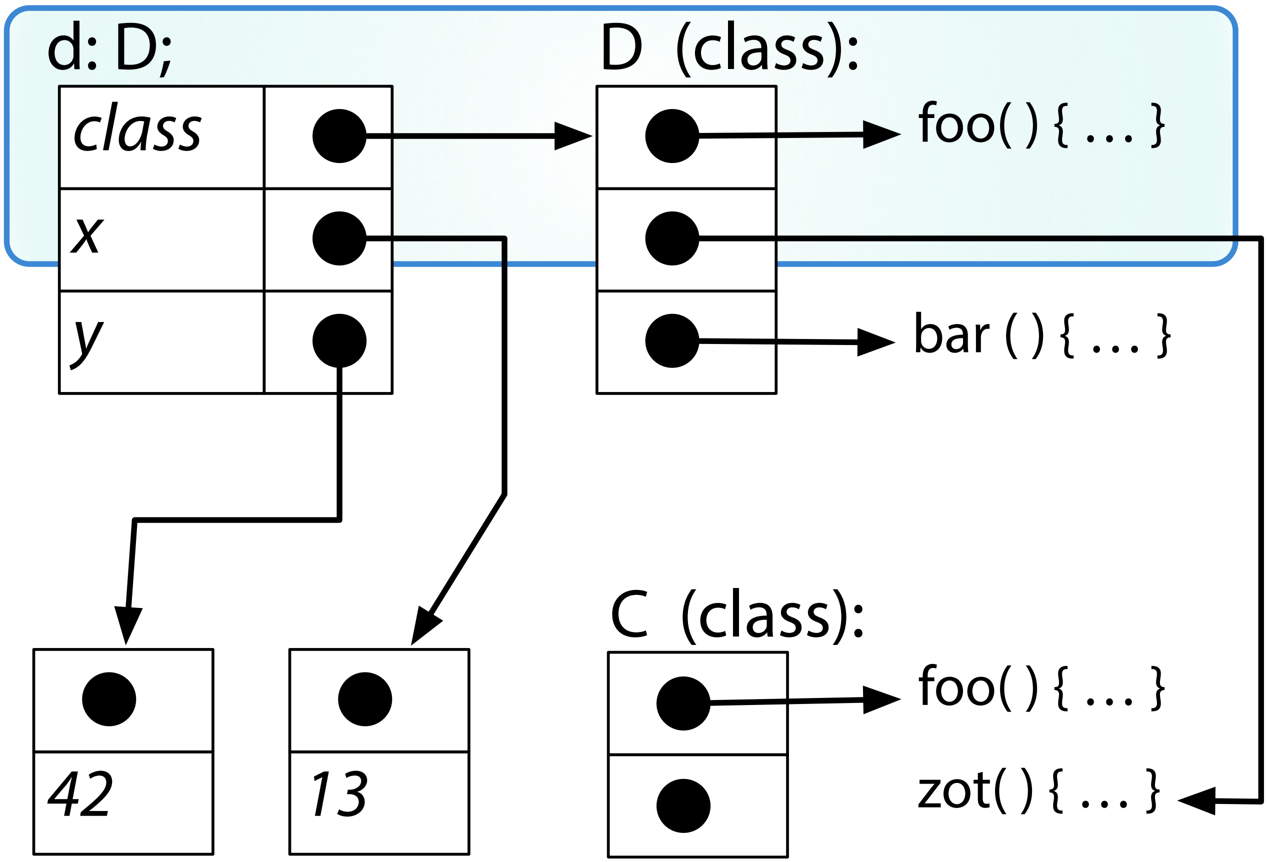
```
extern class_Pt the_class_Pt;

obj_Pt Pt_method_PLUS(obj_Pt this, obj_Pt other) {
  obj_Int this_x = this->x;
  obj_Int other_x = other->x;
  obj_Int this_y = this->y;
  obj_Int other_y = other->y;
  obj_Int x_sum = this_x->clazz->PLUS(this_x, other_x);
  obj_Int y_sum = this_y->clazz->PLUS(this_y, other_y);
  return the_class_Pt->constructor(x_sum, y_sum);
}

  /* The Pt Class (a singleton) */
  struct  class_Pt_struct  the_class_Pt_struct =
  {
    new_Pt,     /* Constructor */
    Obj_method_STRING,
    Pt_method_PRINT,
    Obj_method_EQUALS,
    Pt_method_PLUS
  };
```

Addressing fields (instance variables) in an object



Each field is at some fixed offset from the beginning of the object.
If D is a subclass of C, its inherited fields are at the same offset.

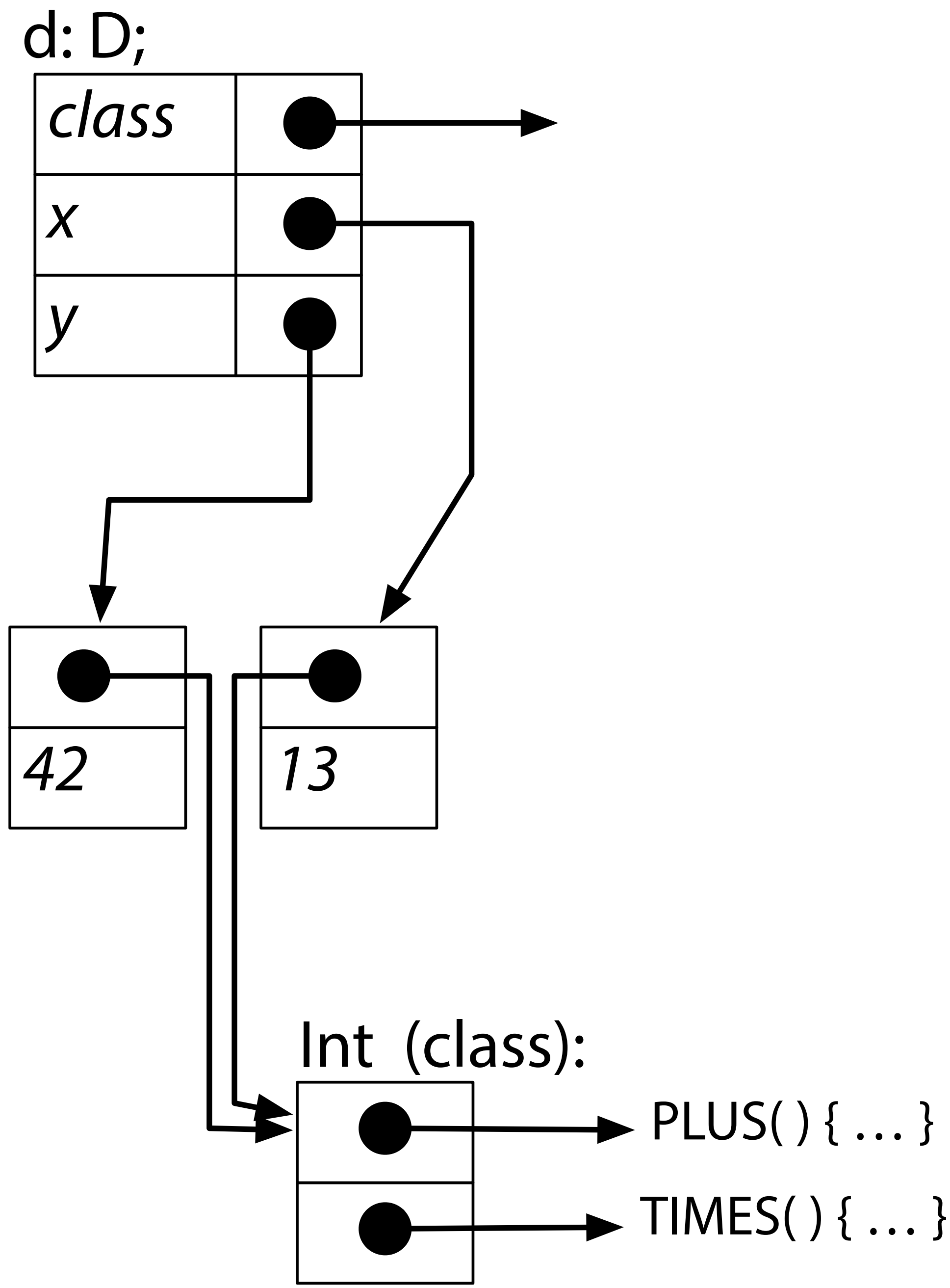
Method dispatch



Each method is at a known offset in the class object. We follow the object pointer, then index into the method table. Subclass method tables align with superclass method tables, so method offsets work for any subclass of static class.

In C: members of structs stand in for offsets in object. Subclasses have same member names at start of struct.

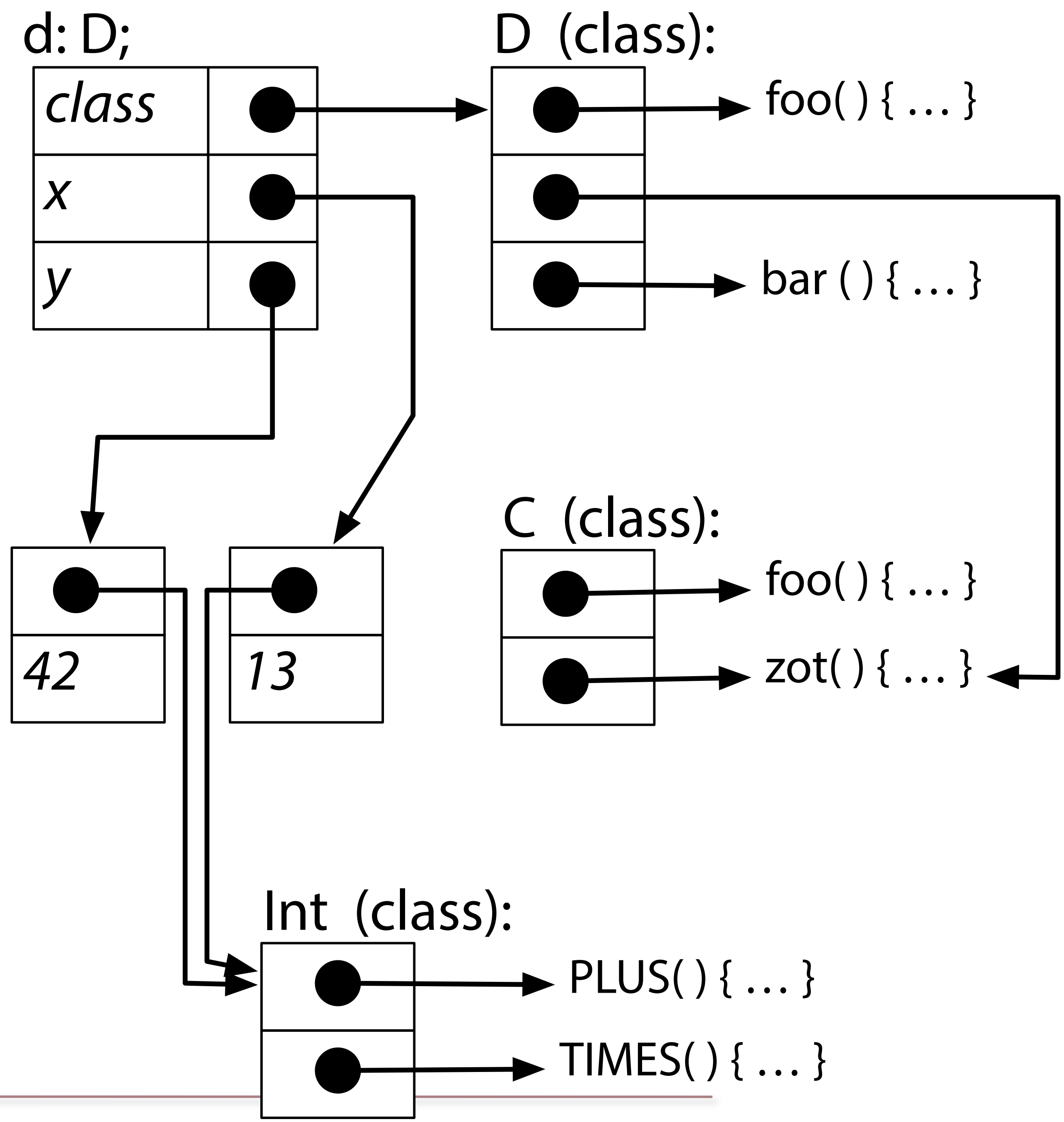
Built-in classes



All values in Quack are objects. Even Int values are "boxed". Only built-in object types may have non-object fields, which may be accessed by methods written in C. Primitive classes mimic Quack classes in layout, to minimize special cases.

The whole picture, again …

```
class C( ) {
    this.x = 42;
    def foo( ): Int {
          this.zot( );
    }
    def zot( ) { … }
}

class D() extends C {
  this.x = 43;
  this.y = 13;
  def foo( ): Int {
       this.zot( );
       this.bar( );
   }
  def bar( ) { … }
}
```



What About Expressions?

- Expression evaluation: Walk the tree
  - Leaves up, leaving result in register
  - Method calls: Evaluate each argument, place in temporary, get method address, make the call
- Control structures - boilerplate
- Conditionals
  - Short circuit evaluation