C Arrays

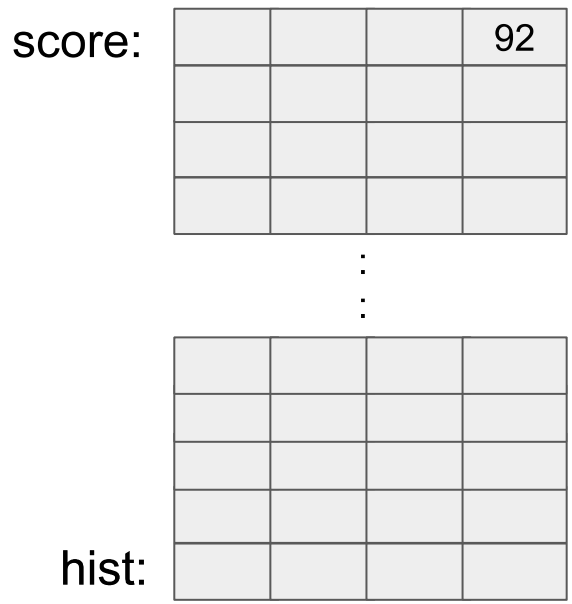
The C source code

```
int hist[100];
int score = 92;
...
hist[score] += 1;
```

might translate to:

```
hist:    .space 100
score:   .word 92

         mov      R3,#hist
         ldr      R2,score
         ldr      R1,[R3,R2,LSL #2]
         add      R1,R1,#1
         str      R1,[R3,R2,LSL #2]
```



Address:
CONSTANT base address + scaled VARIABLE offset

C "Structs"

- C "structs" are lightweight "container objects" - objects with members, but no methods.
- There is special "Java-like" syntax for accessing particular members: *variable.member* (actually, Java's dot operator "." is borrowed from C)
- You can also have pointers to structs.

C provides an new operator to access them:

pointerVariable->member

This simplifies the alternative syntax:

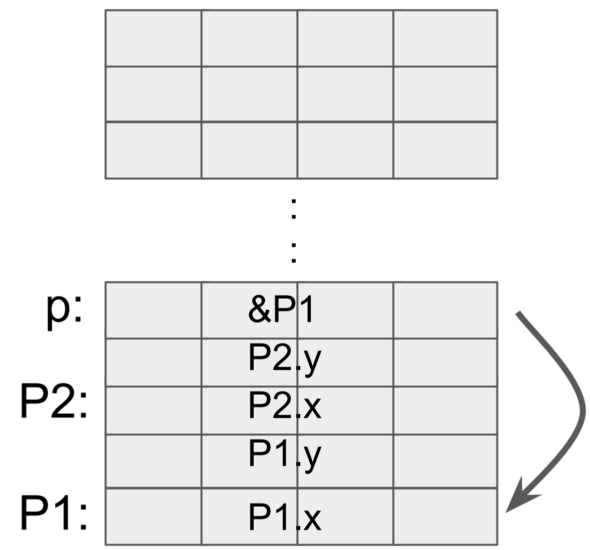
*(*pointerVariable).member*

```
struct Point {
    int x, y;
} P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 123;
```

Structs in Action

```
struct Point {
    int x, y;
} P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 123;
```

Address:
VARIABLE base address + CONSTANT offset



might translate to:

```
P1:  .space 8
P2:  .space 8
p:   .space 4
…
     mov     R1,#P1
     mov     R0,#157
     str     R0,[R1,#0]    ; P1.x = 157
     str     R1,#p         ; p = &P1
     ldr     R2,#p
     mov     R0,#123
     str     R0,[R2,#4]    ; p->y = 123
```

C "IF" TO ASSEMBLY TRANSLATION

C code:

```
if (expr) {
      STUFF
}
```

C code:

```
if (expr) {
      STUFF1
} else {
      STUFF2
}
```

ARM assembly:

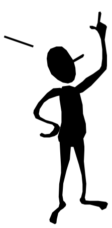
(compute *expr*)

```
    beq Lendif
```

(compile *STUFF*)

```
Lendif:
```

Note: the branches used in assembly "SKIP" code blocks rather than cause them to be executed. This often results in a complement test!



ARM assembly:

(compute *expr*)

```
    beq Lelse
```

(compile *STUFF1*)

```
    b   Lendif
Lelse:
```

(compile STUFF2)

```
Lendif:
```

C "While" Loops

C code:

```
while (expr) {
    STUFF
}
```

Assembly:

```
Lwhile:
        (compute expr)
        beq       Lendw
        (compile STUFF)
        b         Lwhile
Lendw:
```

Alternate Assembly:

```
        b         Ltest
Lwhile:
        (compile STUFF)
Ltest:
        (compute expr)
        bne       Lwhile
Lendw:
```

Compilers spend a lot of time optimizing in and around loops.
- moving all possible computations outside of loops
- unrolling loops to reduce branching overhead
- simplifying expressions that depend on "loop variables"

C "FOR" LOOPS

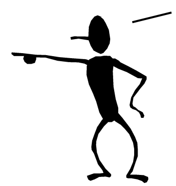
- Most high-level languages provide loop constructs that establish and update an iterator, which controls the loop's behavior

```
for (initialization; conditional; afterthought) {
   STUFF;
}
```

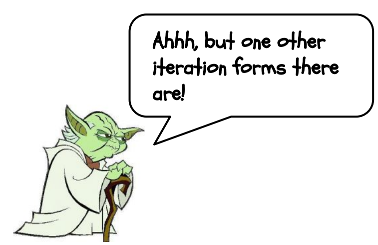
Assembly:

```
     (compile initialization)
Lfor:
     (compute conditional)
     beq Lendfor
     (compile STUFF)
     (compile afterthought)
     B    Lfor
Lendfor:
```

For loops are the most commonly used form of iteration found programming languages.



Their advantage is readability. They bring together the three essential components of iteration, setting an initial value, establishing a termination condition, and giving an update rule.

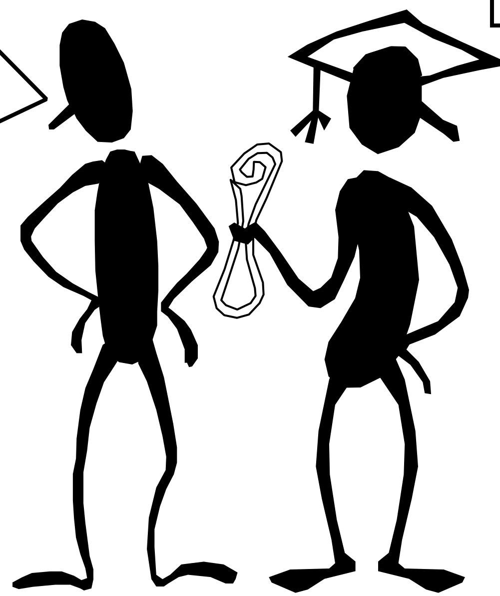


Next time

- The details behind assemblers
- 2-pass and 1-pass assembly
- Linkers and dynamic libraries

Assemblers and Linkers

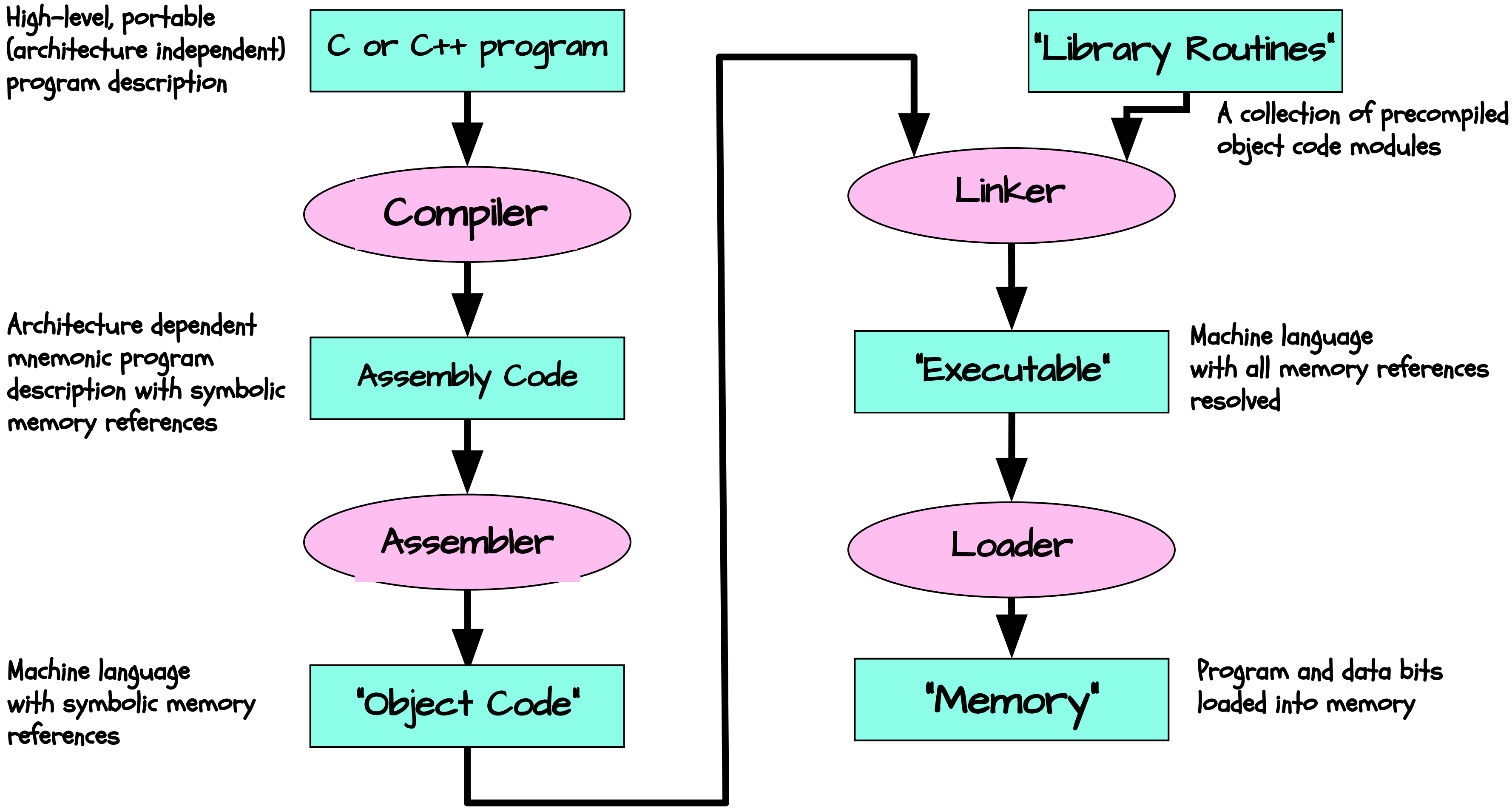
Long, long, time ago, I can still remember
How mnemonics used to make me smile...
Cause I knew with just those opcode names
that I could play some assembly games
and I'd be hacking kernels in just awhile.
But Comp 411 made me shiver,
With every new lecture that was delivered,
There was bad news at the doorstep,
I just didn't get the problem sets.
I can't remember if I cried,
When inspecting my stack frame's insides,
All I know is that it crushed my pride,
On the day the joy of software died.
And I was singing...



When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
"Write in C."

Routes from Programs to Bits

- Traditional Compilation



How an Assembler Works

Three major components of assembly

1) Allocating and initializing data storage
2) Conversion of mnemonics to binary instructions
3) Resolving addresses

```
        .word   0x03fffffc, main
array:  .space  11
total:  .word   0

main:   mov     r1,#array
        mov     r2,#0
        mov     r3,#1
        ldr     r0,total
        b       test
loop:   add     r0,r0,r3
        str     r3,[r1,r2,lsl #2]
        add     r3,r3,r3
        add     r2,r2,#1
test:   cmp     r2,#11
        blt     loop
        str     r0,total
*halt:  b       halt
```

- So is this (main)
- Need to figure out this immediate value (#array)
- This one is a PC-relative offset (ldr r0,total)
- This is a forward reference (b test)
- This offset is completely different than the one a few instructions ago (str r0,total)

Resolving Addresses– 1st Pass

"Old-style" 2-pass assembler approach

Address	Machine code	Assembly code		
0	0x03FFFFFC		.word	0x03fffffc, main
4	0x00000000			
8		array:	.space	11
52	0x00000000	total:	.word	0
56	0xE3A01000	main:	mov	r1,#array
60	0xE3A02000		mov	r2,#0
64	0xE3A03001		mov	r3,#1
68	0xE51F0000		ldr	r0,total
72	0xEA000000		b	test
76	0xE0800003	loop:	add	r0,r0,r3
80	0xE7813102		str	r3,[r1,r2,lsl #2]
84	0xE0833003		add	r3,r3,r3
88	0xE2822001		add	r2,r2,#1
92	0xE352000B	test:	cmp	r2,#11
96	0xBA000000		blt	loop
100	0xE50F0000		str	r0,total
104	0xEA000000	*halt:	b	halt

- In the first pass, data and instructions are encoded and assigned offsets, while a symbol table is constructed.
- Unresolved address references are set to 0

Symbol	Address
array	8
total	52
main	56
loop	76
test	92
halt	104

Resolving Addresses in 2nd pass

"Old-style" 2-pass assembler approach

Address	Machine code	Assembly code
0	0x03FFFFFC	.word 0x03fffffc, main
4	0x00000038	
8		array: .space 11
52	0x00000000	total: .word 0
56	0xE3A01008	main: mov r1,#array
60	0xE3A02000	mov r2,#0
64	0xE3A03001	mov r3,#1
68	0xE51F0018	ldr r0,total
72	0xEA000003	b test
76	0xE0800003	loop: add r0,r0,r3
80	0xE7813102	str r3,[r1,r2,lsl #2]
84	0xE0833003	add r3,r3,r3
88	0xE2822001	add r2,r2,#1
92	0xE352000B	test: cmp r2,#11
96	0xBAFFFFF9	blt loop
100	0xE50F0038	str r0,total
104	0xEAFFFFFE	*halt: b halt

- In the first pass, data and instructions are encoded and assigned offsets, while a symbol table is constructed.
- Unresolved address references are set to 0

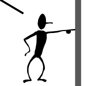
Symbol	Address
array	8
total	52
main	56
loop	76
test	92
halt	104

Modern 1-pass Assembler

Modern assemblers keep more information in their symbol table which allows them to resolve addresses in a single pass.

- Known addresses (backward references) are immediately resolved.
- Unknown addresses (forward references) are "back-filled" once they are resolved.

State of the symbol table after the instruction str r3, [r1,r2,lsl #2] is assembled



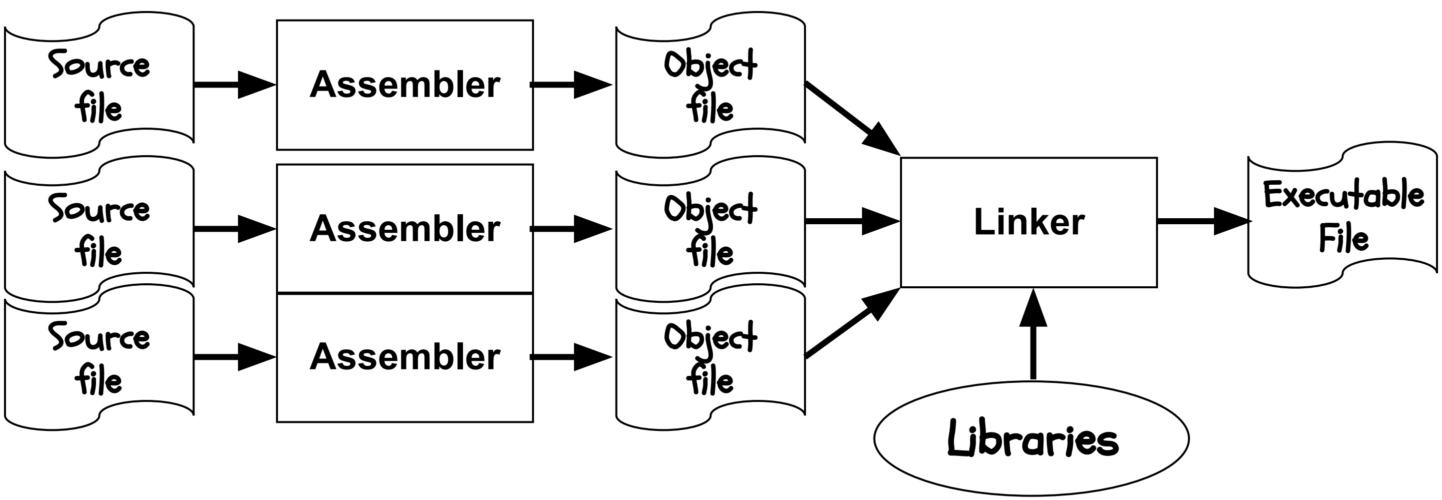
Symbol	Address	Resolved?	Reference list
array	8	y	56
total	52	y	68
main	56	y	4
loop	76	y	?
test	?	n	72

Role of a Linker

Some aspects of address resolution cannot be handled by the assembler alone.

1. References to data or routines in other object modules
2. The layout of all segments in memory
3. Support for **REUSABLE** code modules
4. Support for **RELOCATABLE** code modules

This final step of resolution is the job of a **LINKER**



Static and Dynamic Libraries

- **LIBRARIES** are commonly used routines stored as a concatenation of "Object files". A global symbol table is maintained for the entire library with **entry points** for each routine.
- When a routine in a LIBRARY is referenced by an assembly module, the routine's address is resolved by the **LINKER**, and the appropriate code is added to the executable. This sort of linking is called STATIC linking.
- Many programs use common libraries. It is wasteful of both memory and disk space to include the same code in multiple executables. The modern alternative to STATIC linking is to allow the **LOADER** and **THE PROGRAM ITSELF** to resolve the addresses of libraries routines. This form of lining is called DYNAMIC linking (e.x. .dll).

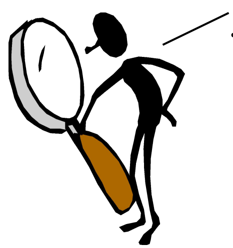
Dynamically Linked Libraries

- C call to library function:

```
printf("sqr[%d] = %d\n", x, y);
```

- Assembly code

```
mov     R0,#1
mov     R1,ctrlstring
ldr     R2,x
ldr     R3,y
mov     IP,__stdio__
mov     LR,PC
ldr     PC,[IP,#16]
```



Dynamically Linked Libraries

- Lazy address resolution:

```
sysload: stmfd sp!,[r0-r10,lr]
         .
         .
         ; check if stdio module
         ; is loaded, if not load it
         .
         .
         ; backpatch jump table
         mov  r1,__stdio__
         mov  r0,dfopen
         str  r0,[r1]
         mov  r0,dfclose
         str  r0,[r1,#4]
         mov  r0,dfputc
         str  r0,[r1,#8]
         mov  r0,dfgetc
         str  r0,[r1,#12]
         mov  r0,dfprintf
         str  r0,[r1,#16]
```

Because, the entry points to dynamic library routines are stored in a TABLE. And the contents of this table are loaded on an "as needed" basis!

Before any call is made to a procedure in "stdio.dll"

```
 .globl __stdio__:
__stdio__:
fopen:    .word sysload
fclose:   .word sysload
fgetc:    .word sysload
fputc:    .word sysload
fprintf:  .word sysload
```

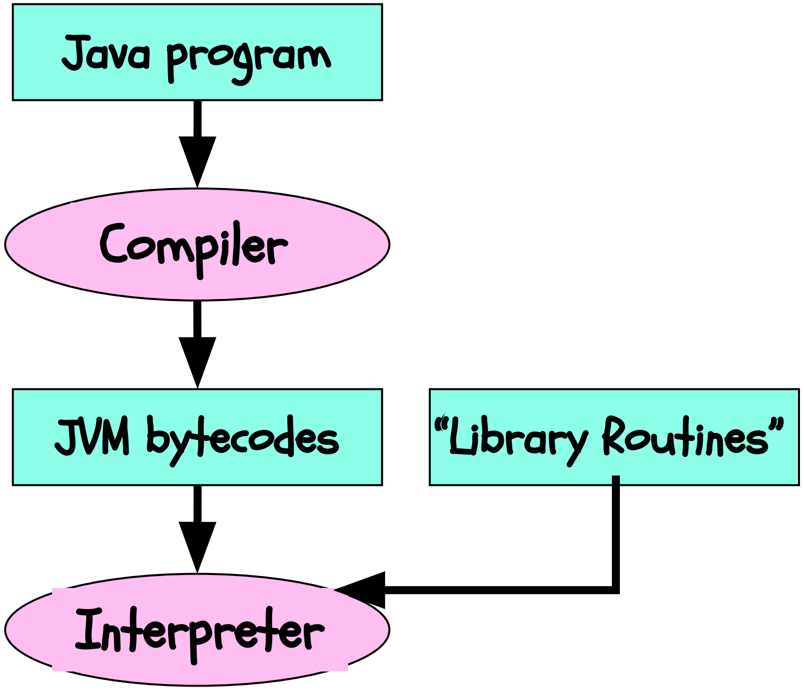
After the first call is made to any procedure in "stdio.dll"

```
 .globl __stdio__:
__stdio__:
fopen:    dfopen
fclose:   dclose
fgetc:    dfgetc
fputc:    dfputc
fprintf:  dprintf
```

Modern Languages

Intermediate "object code language"

High-level, portable (architecture independent) program description



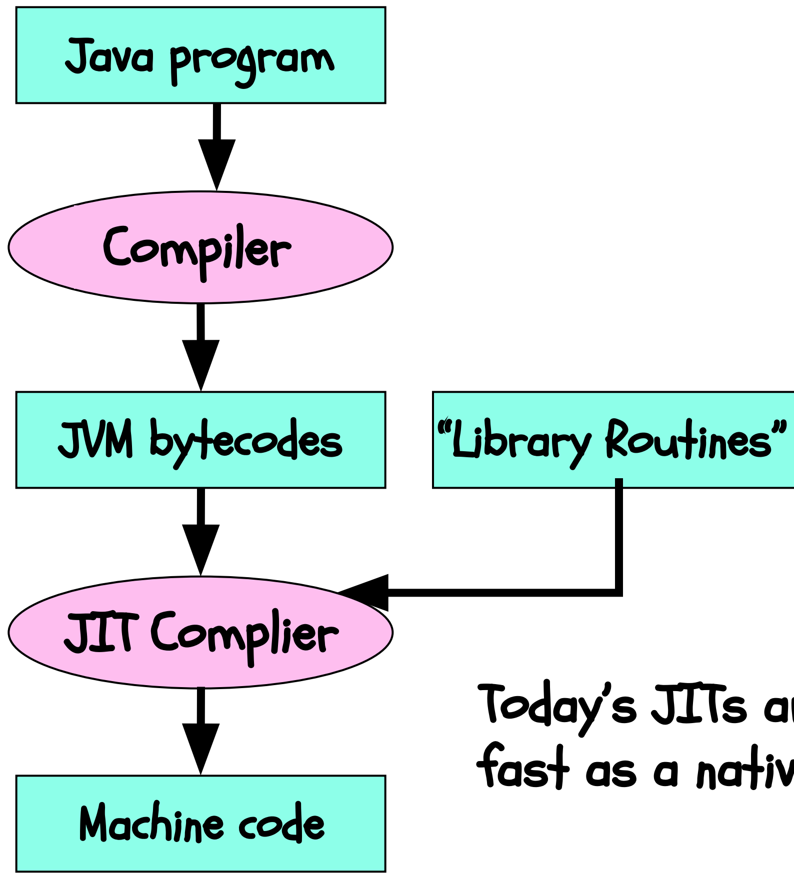
PORTABLE mnemonic program description with symbolic memory references

An application that EMULATES a virtual machine. Can be written for any Instruction Set Architecture. In the end, machine language instructions must be executed for each JVM bytecode

Modern Languages

Intermediate "object code language"

High-level, portable (architecture independent) program description



PORTABLE mnemonic program description with symbolic memory references

While interpreting on the first pass the JIT keeps a copy of the machine language instructions used.
Future references access machine language code, avoiding further interpretation

Today's JITs are nearly as fast as a native compiled code.

Assembly? Really?

- In the early days compilers were dumb
  - literal line-by-line generation of assembly code of "C" source
  - This was efficient in terms of S/W development time
    - C is portable, ISA independent, write once- run anywhere
    - C is easier to read and understand
    - Details of stack allocation and memory management are hidden
  - However, a savvy programmer could nearly always generate code that would execute faster
- Enter the modern era of Compilers
  - Focused on optimized code-generation
  - Captured the common tricks that low-level programmers used
  - Meticulous bookkeeping (i.e. will I ever use this variable again?)
  - It is hard for even the best hacker to improve on code generated by good optimizing compilers

Next Time

- Compiler code optimization
- We look deeper into the Rabbit hole