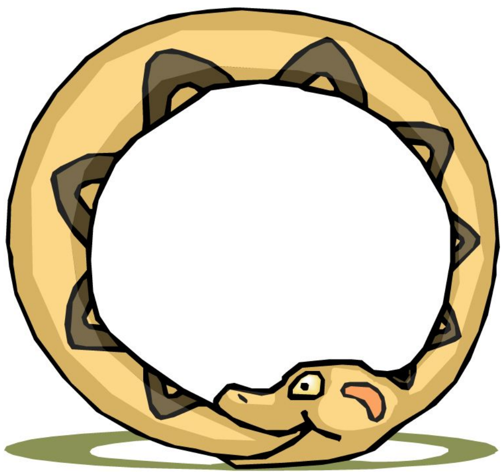
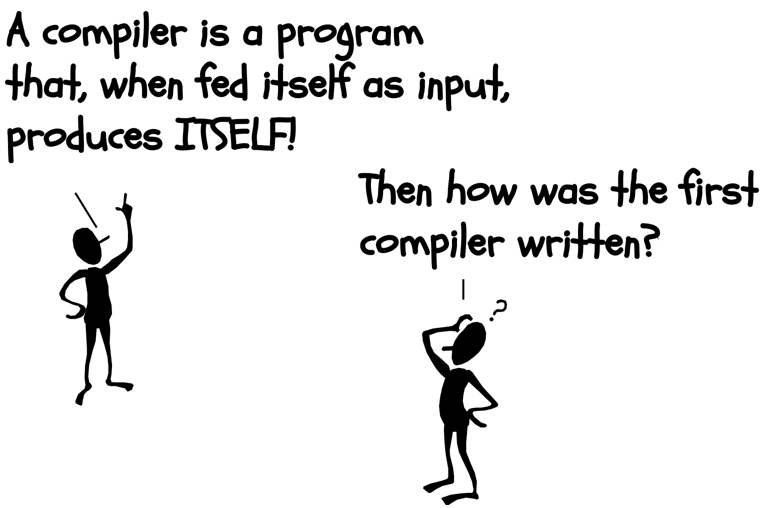
Compilers and Interpreters





- Pointers, the addresses we can see
- Programs that write other programs
- Managing the details

1) Okay... still playing catch up. We'll move the first midterm to 10/8
2) Third Problem Set goes out tonight. 2nd is due 10/1.
3) Short lecture and mini lab on Friday.

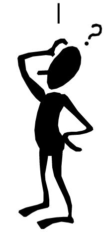
Warm up

```
        .word   0x03fffffc, main
x:      .word   42

main:   ldr     R0,x
        mov     R1,#x
halt:   b       halt
```

miniARM

What is in registers R0 and R1 after these two instructions are executed?



An Aside: Let's C

C is the ancestor to most languages commonly used today.

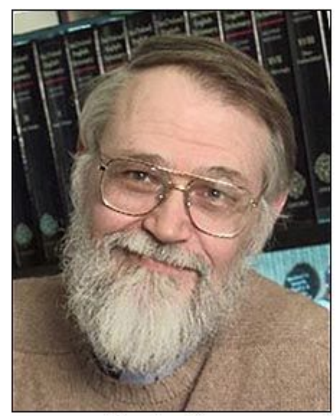
{Algol, Fortran, Pascal} → C → C++ → Java

C was developed to write the operating system UNIX.

C is still widely used for "systems" programming

C's developers were frustrated that the high-level languages available at the time, lacked many of the capabilities of assembly.

An advantage of high-level languages is that they are portable (i.e. not ISA specific).
C, thus, was an attempt to create a portable blend of a "high-level language" and "assembler"

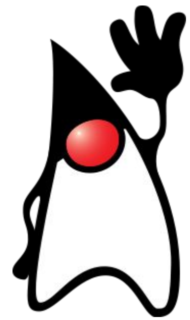


Brian Kernighan

Dennis Ritchie

C begat Java

C++ was envisioned to add Object-Oriented (OO) concepts from *Simula* and CLU on top of C



Java was envisioned to be more purely OO, and to hide the details of memory management as well as Class/Method/Member implementation

For our purposes C is almost identical to JAVA except:

- C has "functions", whereas JAVA has "methods".
- C has explicit variables that contain the **addresses** of other variables or data structures in memory.
- JAVA hides addresses under the covers.

Your first C pointer!

Let's start with a feature that Java does not have-- called "pointers"

```
int i = 4;    // simple integer variable
int a[10];    // array of integers (a is a pointer)
int *p;       // pointer to integer (s)
```

`*(expression)` means the "**contents of address computed by expression**".

`a[i] ≡ *(a+i)`

`a is a constant of type "int *"`

`a[i] = a[i+1]  ≡  *(a+i) = *(a+i+1)`

Array variables are our first hint that "pointers" exist. The name of an array tells us where a collections of indexable variables could be found.

We now know that *all* variables are shorthands for addresses in memory.

Normal variables are just the 0th element of a length "1" array..

Other Pointer Related Syntax

```
int i;            // simple integer variable
int a[10];        // array of integers
int *p;           // pointer to integer(s)

p = &i;           // & means address of (not AND)
p = a;            // no need for & on a
p = &a[5];        // address of 6th element of a
*p = 1;           // change value of that location
*(p+1) = 1;       // change value of next location
p[1] = 1;         // exactly the same as above
p++;              // step pointer to the next element
(*p)++;           // increments contents of location
*p++;             // get contents, and then modifies p
```

The ampersand operator, "&", means "give me the address of this variable reference". Whereas the star operator, "*", means "give me the contents of the memory location implied by the expression". These are VERY different things. Not to mention, "&" and "*" can sometimes be confusing because of their other uses as "anding" and "multiplying" operators.

Legal uses of Pointers

```
int i;        // simple integer variable
int a[10];    // array of integers
int *p;       // pointer to integer(s)

So what happens when: p = &i;
What is value of p[0]?
What is value of p[1]?
```

p[0] is always an alias for the variable i in this context. p[1] could reference a[0], but don't count on it.

C Pointers vs. Object Size

```
int i;        // simple integer variable
int a[10];    // array of integers
int *p;       // pointer to integer(s)

i = *p++;
```

Does “p++” really add 1 to the pointer?
NO! It adds 4. Why 4?

```
char *q;
...
```

The “char” type is slightly different than the type of the same name in Java. C chars are 8-bit signed bytes. Java chars are 16-bits and hold only Unicode variables (they have no sign). Java has a type called “byte” that is most similar to a C “char”.

```
q++; // really does add 1
```

Clear1,2,3, All are valid C!

```
void clear1(int array[], int size) {
    for (int i = 0; i < size; i++)
        array[i] = 0;
}
```

Written using "Array" semantics

```
void clear2(int array[], int size) {
    for (int *p = array; p < array + size; p++)
        *p = 0;
}
```

Written using C "Pointer" semantics.

```
void clear3(int *array, int size) {
    int *end = array + size;
    while (array < end)
        *array++ = 0;
}
```

Array is just a pointer.

Pointer Summary

- In the "C" world and in the "machine" world:
  - a pointer is just the ***address*** of an object in memory
  - size of pointer is fixed and architecture dependent, regardless of size of object that it points to
  - to get to objects of the same type, we offset by increments of the object's size in bytes
  - Ex: to get the the i^{th} object add i * sizeof(object)

- More details:
  - `int R[5] ≡ R` (i.e. R is an int* to 20 bytes of storage)
  - `R[i] ≡ *(R+i)` (array offsets are just pointer arithmetic)
  - `int *p = &R[3] ≡ p = (R+3)` (p points to 3^{rd} element of R)

Indirect Addressing

- What we want:
  - The contents of a memory location held in a register
- Examples:

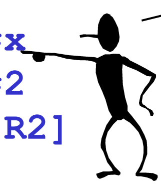
"C"

```
int x = 10;

main() {
    int *y = &x;
    *y = 2;
}
```

"ARM Assembly"

```
x:        .word   10
main:     mov     R2,#x
          mov     R3,#2
          str     R3,[R2]
          bx      LR
```



Loads the "address" of x into R2, not its contents

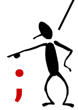
- Caveats
  - You must make sure that the register contains a ***valid address*** (double, word, or short ***aligned*** as required)

Compilers as Template Matchers

The basic task of a compiler is to scan a file looking for particular sequences of operators and keywords called **templates**.

The first major sort of template is an **expression**. We've already played around converting C expressions to assembly language. A compiler does basically the same thing.

```c
int x, y;
y = (x-3)*(y+123456);
```



Here the compiler noticed that the desired constant was too big to fit as an immediate constant, so it creates a new variable, c, to keep track of this constant. (Usually, compiler generated constant names are cryptic, so you can't generate them by chance).

```
x:   .word 0
y:   .word 0
c:   .word 123456

     ...
     ldr    R0,x
     sub    R0,R0,#3
     ldr    R1,y
     ldr    R2,c
     add    R1,R1,R2
     mul    R0,R0,R1
     str    R0,y
```

Once a template is matched, a compiler emits a specific code sequence.

C Arrays

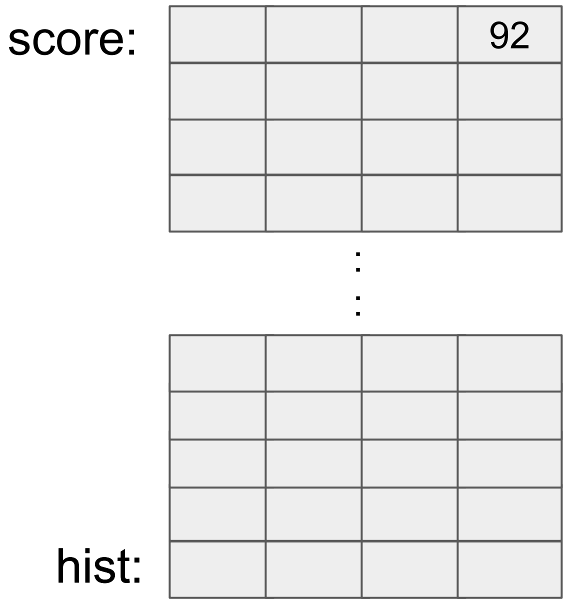
The C source code

```
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

An implied dereference with an implied offset. Similar to *(p+1)

In place of the alternative syntax:

*(*pointerVariable).member*

Here the dereference is more explicit by requires more typing.

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

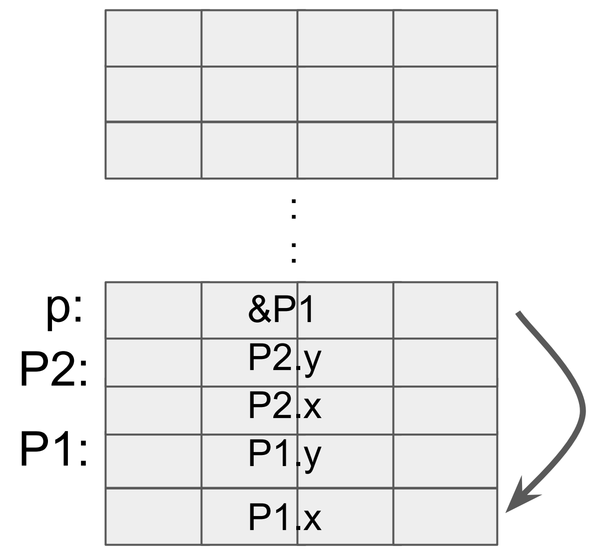
Structs in Action

```
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

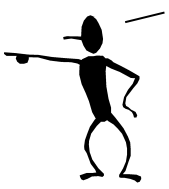
- Most high-level languages provide loop constructs that establish and update an iterator, which controls the loop's behavior

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

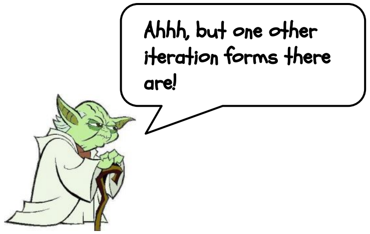
Assembly:

```
     (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