

Compilers and Interpreters



A compiler is a program that, when fed itself as input, produces ITSELF!



Then how was the first compiler written?



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

An Aside: Let's C

C is the ancestor to most languages commonly used today.

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

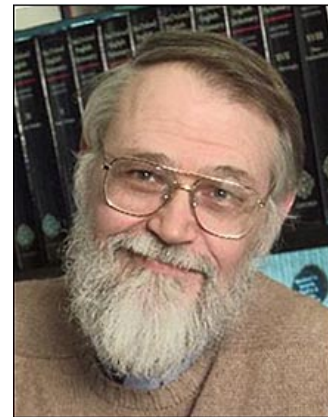
C was developed to write the operating system UNIX.

C is still widely used for “systems” programming

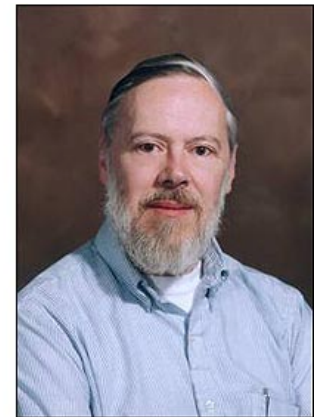
C's developers were frustrated that the high-level languages available at the time, lacked all the capabilities of assembly code.

But, the 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 them under the covers.

Your first C pointer!

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

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

`* (expression)` is content of address computed by expression .

`a[k] ≡ *(a+k)`

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

`a[k] = a[k+1] ≡ *(a+k) = *(a+k+1)`

Array variable names were our first hint that "pointers" existed. The name of an array was somehow telling us where a collection of indexable variables could be found. We now know that *all* variables are shorthands for addresses in memory. And, array variables are just the address of the 0th element.



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
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
if(*p) (*p)++; // contents of location pointed by p
if(*p) *p++;   // changes location pointed to by 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;
```


```
...
```


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




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".

Clear123, All are valid C!

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

```
void clear2(int array[], int size) {  Written using C "Pointer" semantics.  
    for (int *p = &array[0]; p < &array[size]; p++)  
        *p = 0;  
}
```

```
void clear3(int *array, int size) {  Written using more efficient C "Pointer" semantics.  
    int *end = array + size;  
    while (array < end)  
        *array++ = 0;  
}
```


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 the next object of the same type, we increment by the object’s size in bytes
 - to get the the i^{th} object add $i * \text{sizeof}(\text{object})$
- More details:
 - $\text{int } R[5] \equiv R$ (i.e. an int^* to 20 bytes of storage)
 - $R[i] \equiv *(R+i)$ (array offsets are just pointer arithmetic)
 - $\text{int } *p = \&R[3] \equiv p = (R+3)$ (p points to 3rd element of R)

Indirect Addressing

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

“C”

```
int x = 10;

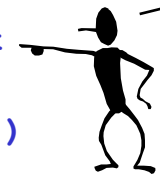
main() {
    int *y = &x;
    *y = 2;
}
```

“MIPS Assembly”

```
main:    ori    $2,$0,x
        addi  $3,$0,2
        sw   $3,0($2)
        jr   $31
```

```
x: .word 10
```

Loads the “address”
of x into \$2, 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 operations 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.

Input: \longrightarrow Output:

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

```
x:      .word 0
y:      .word 0
c:      .word 123456
...
lw      $t0, x
addi    $t0, $t0, -3
lw      $t1, y
lw      $t2, c
add     $t1, $t1, $t2
mul     $t0, $t0, $t1
sw      $t0, y
```

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

C Data Structures: Arrays

The C source code

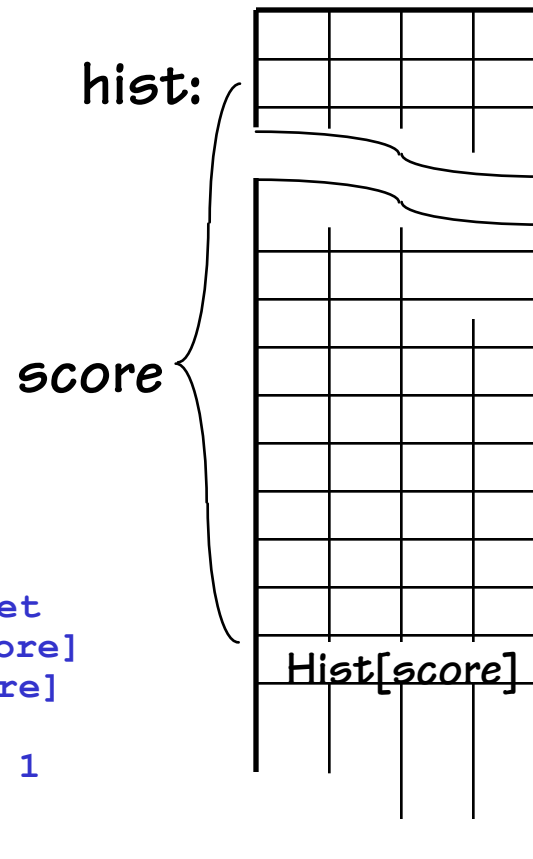
```
int hist[100];
int score = 2;
...
hist[score] += 1;
```

might translate to:

```
        .align 2
hist:   .space 400
score:  .word 2
...

lw      $24,score      # $24 = score
sll     $24,$24,2      # make word offset
addui   $24,$24,hist   # $24 = &hist[score]
lw      $15,($24)      # $15 = hist[score]
addui   $15,$15,1      # $15 = $15 + 1
sw      $15,($24)      # hist[score] += 1
```

Memory:



Address:

CONSTANT base address +
VARIABLE offset computed from index

Displacement Addressing

- What we want:
 - The contents of a memory location relative to a register

- Examples:

“C”

```
int a[5];

main() {
    int i = 3;
    a[i] = 2;
}
```

“MIPS Assembly”

```
main:    addi $2,$0,3    # i = 3
        addi $3,$0,2
        sll  $1,$2,2
        sw   $3,a($1)
        jr   $31
```

```
a:      .space 5
```

Space for a 5 integers
(20-bytes)



- Caveats

- Must multiply (shift) the “index” to be properly aligned,
(i.e. to match the size of the type that the pointer references)

C Data Structures: 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;
```

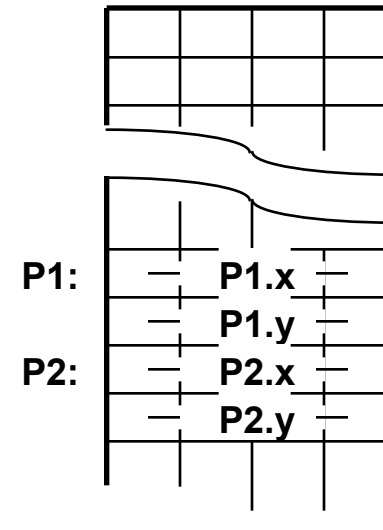
C Data Structures: Structs

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

might translate to:

```
P1:      .space 8  
P2:      .space 8  
p:       .space 4  
...  
  
addiu $15,$0,157  
sw    $15,P1          # P1.x = 157  
addui $24,$0,P1  
sw    $24,p           # p = P1  
lw    $24,p  
addiu $15,$0,123  
sw    $15,4($24)     # p->y = 123
```

Memory:



Address:

VARIABLE base address +
CONSTANT component offset

Offset Addressing

- What we want:
 - The contents of a memory location relative to a register
- Examples:

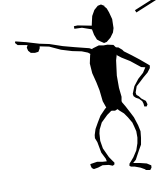
“C”

```
struct p {  
    int x, y;  
}  
  
main() {  
    p.x = 3;  
    p.y = 2;  
}
```

“MIPS Assembly”

```
main:    ori    $1,$0,p  
        addi   $2,$0,3  
        sw     $2,0($1)  
        addi   $2,$0,2  
        sw     $2,4($1)  
        jr     $31  
  
p:      .space 8
```

Allocates space for
2 uninitialized
integers (8-bytes)



- Caveats
 - Constants offset to the various fields of the structure
 - Structures larger than 32K use a different approach

C/Assembly Translation: Conditionals

C code:

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

C code:

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

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



MIPS assembly:

```
(compute expr in $rx)  
beq $rx, $0, Lendif  
(compile STUFF)
```

Lendif:

MIPS assembly:

```
(compute expr in $rx)  
beq $rx, $0, Lelse  
(compile STUFF1)  
beq $0, $0, Lendif
```

Lelse:

```
(compile STUFF2)
```

Lendif:

There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:

```
if (y > 32) {  
    x = x + 1;  
}
```

compiles to:

```
lw    $24, y  
ori   $15, $0, 32  
slt   $1, $15, $24  
beq   $1, $0, Lendif  
lw    $24, x  
addi  $24, $24, 1  
sw    $24, x
```

Lendif:

C/Assembly Translation: Loops

C code:

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

MIPS assembly:

Lwhile:

(compute expr in \$rX)

beq \$rX,\$0,Lendw

(compile STUFF)

beq \$0,\$0,Lwhile

Lendw:

Alternate MIPS
assembly:

beq \$0,\$0,Ltest

Lwhile:

(compile STUFF)

Ltest:

(compute expr in \$rx)

bne \$rX,\$0,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/Assembly Translation: For Loops

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

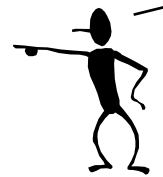
```
for (initialization; conditional; afterthought) {  
    STUFF;  
}
```

Assembly:

```
(compile initialization)  
Lfor:  
    (compute conditional in $rX)  
    beq $rX,$0,Lendfor  
    (compile STUFF)  
    (compile afterthought)  
    beq $0,$0,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.



C/Assembly Translation: For Loops

- For loops with “**for object in iterator**” were added to Java and its other modern descendants like Python. They were not included in C.

C code:

```
int sum = 0;


int data[10] =
    {1,2,3,4,5,6,7,8,9,10};

int i;
for (i=0; i<10; i++) {
    sum += data[i]
}
```

MIPS assembly:

```
sum:
    .word 0x0
data:
    .word 0x1, 0x2, 0x3, 0x4, 0x5
    .word 0x6, 0x7, 0x8, 0x9, 0xa

    add $30,$0,$0
Lfor:
    lw $24,sum($0)
    sll $15,$30,2
    lw $15,data($15)
    addu $24,$24,$15
    sw $24,sum
    add $30,$30,1
    slt $24,$30,10
    bne $24,$0,Lfor
Lendfor:
```



for-loop semantics allow the compiler to eliminate the initial test if it cannot be true as a result of the initialization.

Next Time

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

