Pointers in C

CS 350: Computer Organization & Assembly Language Programming

A. Why?

- C is an important ancestor of C++ and Java.
- Pointers are an efficient way to share large memory objects without copying them.

B. Outcomes

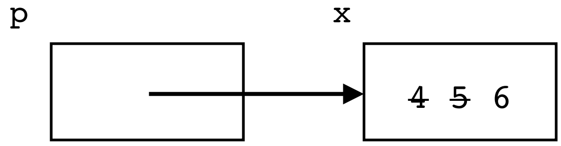
After this lecture, you should

- Know how to declare, initialize, assign, and use pointer variables in C.
- Know how to write the `&` and `*` syntax in C and when to use it.
- Know the difference between *ptr* = `&`*var* and `*`*ptr* = *var* in C.
- Know how basic pointer operations in C can be represented in LC-3 assembler.

C. Pointers in C

- A **pointer** is a memory address.
- A **pointer variable** is a variable whose value is an address.
- C lets us talk about and manipulate pointers as variables and in expressions.

Syntax	***Meaning***
`int x;`	Declare and allocate space for an integer
`int *p;`	Declare and allocate space for a pointer to an integer
`x = 4;`	Copy 4 into the memory location associated with `x`
`p = &x;`	Copy address of `x` into the memory location associated with `p`
`*p = 5;`	Follow `p` to memory location, copy 5 there (changes `x` to 5).
`x = *p+1;`	Set `x` to 5+1 (the value pointed to by `p`, plus 1)



- **Pointer Declarations**
  - A pointer in C is always a pointer to a particular data type:
    - The type name includes asterisk: `int *`, `double *`, `char *`, etc.
    - Often use `p`, `q`, `r` for pointers.
    - Must place the `*` before each pointer variable when declaring them. (Spaces before or after the `*` in `*p` are ignored.)
    - **Example**: Here, `p`, `q`, `s`, and `t` are pointers to `int`; `r` and `u` are `int`.

```
int *p, *q, r;
int* s, *t, u;
```

D. Pointer Operations

- There are two basic operators, `&` and `*`. The first is used to get an address, the second to go to an address ("follow a pointer").

The Address-of/Referencing Operator, &

- The **address-of/referencing operator** (`"&"`) yields the address of a variable
  - **Example**: `&x` is the address of the variable `x`
  - **Example**: `&b[0]` is the address of the 0'th element of array `b`.
- You can take the `&` of any expression that can be assigned to, so `&17` is illegal but `&` of an array element is legal
- **lvalue vs rvalue**: The meaning of an expression differs according to whether it's on the left- or right-hand side of an assignment.
  - Its **rvalue** is the value it has on the right-hand side
  - If it's assignable, its **lvalue** is the address it stands for.
    - So the meaning of `x = x+1;` is "Add `1` to the value at the location called `x` and copy the result to the location called `x`".
  - You can take the `&` of an expression iff it has an lvalue.

  - The `&` operator yields a constant (you can't assign to it): `&x =` *e* is illegal.
- The main reason for taking the `&` of something is to assign the result to a pointer variable.
  - **Example**: `int x = 12; int *p; p = &x;`
  - `p` "**points to**" `x` iff the test `(p == &x)` is true.

*The Dereferencing Operator, **

- The **dereferencing operator** (`"*"`) takes an address and yields the item at the address. You can take the `*` of any pointer-valued expression.
- As an lvalue, `*p` stands for the value at the location pointed to by `p`.
  - **Example**: `int x = 12; int *p; p = &x; printf("%d\n", *p);` prints out `12`
- As an rvalue, `*p` stands for the address pointed to by `p`.
  - If `p = &x`, then `*p =` *e* behaves like `x =` *e* .
  - Given `p = &x;` as above, `*p = 20; printf("%d\n", *p);` prints `20`.
- In general, `*&x` is `x`, so `x = 17` and `*&x = 17` have the same effect

Printing Pointers Using %p

- You can `printf` an address using the `%p` format code. E.g., to print the location `p` points to and the value stored there, use

```
printf("The value at %p is %d\n", p, *p);
```

- This also works with the `&` operator

```
printf(“x is at %p and has value %d\n”, &x, x);
```

E. Example

```
// ptr1.c: Declare, set, and inspect a pointer.

#include <stdio.h>
int main() {
  int x = 17;
  printf("Value and address of x:\n");
  printf("x : %d\n", x);
  printf("&x: %p\n", &x);
  int *p;
  p = &x;
  printf("\nValue of p, value of *p, address of p:\n");
  printf("p : %p\n", p);
  printf("*p: %d\n", *p);
  printf("&p: %p\n", &p);

  x = 23;
  printf("\nAfter x = 23:\n");
  printf("x : %d\n", x);
  printf("*p: %d\n", *p);
}
```

```
unix > gcc ptr1.c
unix > a.out
Value and address of x:
x : 17
&x: 0x7fff5fbff41c

Value of p, value of *p, address of p:
p : 0x7fff5fbff41c
*p: 17
&p: 0x7fff5fbff410

After x = 23:
x : 23
*p: 23
```

- **Declare and Initialize:** The syntax for initializing pointers is tricky:
    - **Example:** `int *p = &x;` is like `int *p; p = &x;` It's not like `int *p; *p = &x;` (which gets a type error because `*p` is of type `int` and `&x` is of type pointer-to-`int`).
    - Similarly, `int *q = p;` is like `int *q; q = p;` not `int *q; *q = p;` (which again gets a type error).
- **Pointer Comparisons**
    - You can compare two pointers for <, >, <=, >=, ==, and `!=` .
    - You can't compare a pointer and an integer.

```
int b[5], *p, *q;
p = &b[0];
q = &b[1];
printf("%p, %p\n", p, q);
printf("%d\n", (p < q));  // prints 1
```

- **Pointer Equality and Aliasing**
  - Two pointers are **aliased** if they are `==` (that is, they point to the same location)

    ```
    int x, *p, *q;
    p = &x;
    q = p; // Makes p and q aliased
    ```

  - In general, `p == q` implies `*p == *q`, but `*p == *q` doesn't always imply `p == q` because you could have `p` and `q` point to different locations that happen to have the same value stored in them.
- **Example**: After `int x = 17, y = 17, *p, *q;  p = &x; q = &y;` all the following expressions evaluate to true: `x == y, &x != &y, p == &x, q == &y, p != q, *p == *q`

F. Passing Pointer Arguments as Parameters

- **Call-by-Reference** (**CBR**, a.k.a. pass-by-reference) is a style of function call where changes to the parameter variable in the function simultaneously cause changes to the value of the argument variable passed by the caller.
  - In CBR, argument expressions and parameter variables are aliased. (Note in CBR, argument expressions must have lvalues.)
- One use of pointers in C is to emulate pass-by-reference.
  - In the caller, the we pass the address of the variable as the argument.
  - We write the subroutine to take pointer arguments; everywhere we want to refer to the caller's variable, we use `*` of our pointer parameter variable.

    ```
    // swap.c: Emulate call-by-reference using pointers
    //
    #include <stdio.h>
    void swap(int *v1, int *v2) {
    ```

```
   printf("Swapping values at %p and %p\n", v1, v2);
   int temp = *v1;
   *v1 = *v2;
   *v2 = temp;
}

main() {
   int x = 2, y = 3;
   printf("x and y are at %p and %p\n", &x, &y);
   printf("Before swap, x: %d, y: %d\n", x, y);
   swap(&x, &y);  // Now x == 3, y == 2
   printf("After swap, x: %d, y: %d"\n", x, y);
}

unix > gcc swap.c
unix > a.out
x and y are at 0x7fff5fbff41c and 0x7fff5fbff418
Before swap, x: 2, y: 3
Swapping values at 0x7fff5fbff41c and 0x7fff5fbff418
After swap, x: 3, y: 2
```

- Compare with this non-working swap program:

```
// badswap.c: Buggy call-by-reference
// Changes to arguments passed by value aren't
// reflected in caller's variables.

#include <stdio.h>
void badswap(int v1, int v2) {
   printf("Swapping values at %p and %p\n", &v1, &v2);
   int temp = v1;
   v1 = v2;
   v2 = temp;
}

main() {
   int x = 2, y = 3;
   printf("x and y are at %p and %p\n", &x, &y);
   printf("Before swap, x: %d, y: %d\n", x, y);
   badswap(x, y);  // Doesn't change x and y
   printf("After swap, x: %d, y: %d\n", x, y);
}

unix > gcc swap.c
unix > a.out
x and y are at 0x7fff5fbff41c and 0x7fff5fbff418
```

```
Before swap, x: 2, y: 3
Swapping values at 0x7fff5fbff3ec and 0x7fff5fbff3e8
After swap, x: 2, y: 3
```

G. Translating C code to Assembler

- Compilers generate object code by translating higher-level constructs to lower-level equivalents. For the LC-3, let *P* and *V* be registers that hold pointers and values respectively, then
    - `LEA` *P*, *var* means *P* `= &`*var*`;`
    - `LDR` *V*, *P*, `0` means *V* `= *`*P*`;`
    - `STR` *V*, *P*, `0` means `*`*P* `=` *V*`;`
- C declarations translated to LC-3
    - Note use of a variable as the value of a `.FILL`; the value of the FILL is the address of the variable.

C Declarations	***LC-3 Declarations***
`int x = 17;`	`x .FILL 17`
`int *p = &x;`	`p .FILL x`
`int b[3] = {0,0,0};`	`b .BLKW 3`

- C code translated to LC-3 (Note assumptions about kinds of variables in registers)

C Code	***LC-3 Code***
`; int *R1, *R2, R5;`	
`R1 = &x;`	`LEA R1, x`
`R2 = p;`	`LD  R2, p`
`p = R1;`	`ST  R1, p`
`R5 = *R1;`	`LDR R5, R1, 0`
`*R1 = R5;`	`STR R5, R1, 0`

- More-complicated C constructs get broken down into parts:

  V `= *p;` becomes P `= p;` V `= *`P`;`

  `*p =` V`;` becomes P `= p; *`P `=` V`;`

  `x = *p;` becomes V `= *p; x =` V`;`

  `*p = x;` becomes V `= x; *p =` V`;`

  `*p = *q;` becomes V `= *p; *q =` V`;`

Pointers in C

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Pointers are an efficient way to share large memory objects without copying them.

B. Outcomes

After this activity, you should

- Be able to hand-execute code that uses the `*` and `&` operators in C.
- Be able to translate C code that uses pointers into equivalent LC-3 code.

C. Questions

1. Draw a memory diagram for the state at the end of execution of

```
int w=3, z=4, *p, *q;
p = &w;
*p = z;
z = 5;
q = &z;
p = q;
*p = 6;
```

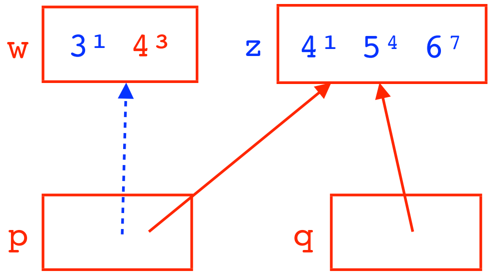
2. Draw a memory diagram for the state at the end of execution of

```
int x = 7, y = 9;
int *p = &x, *q = p;
*p = *q+1;
q = &y;
*q = *q+1;
```

3. Translate the C code from Problem 2 into corresponding LC-3 code.

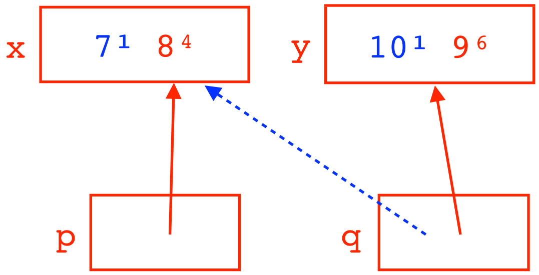
Solution

1. (Superseded values/pointers are shown in blue. Superscripts indicate the line of code that established that value.)



```
1  int w=3, z=4, *p, *q;
2  p = &w;
3  *p = z;
4  z = 5;
5  q = &z;
6  p = q;
7  *p = 6
```

2. (Same conventions as in problem 1.)



```
1  int x = 7, y = 10;
2  int *p = &x, *q;
3  q = p;
4  *p = *q+1;
5  q = &y;
6  *q = *p+1;
```

3. (LC-3 code for Problem 2 code)

```
; Register usage: R0 = p, R1 = q, R2 = temp
; See end of code for declarations of x, y, p, q

; q = p;
    LD   R0, p      ; establish R0 = p
    ADD  R1, R0, 0
    ST   R1, q      ; establish q = p (= R1)

; *p = *q+1;
    LDR  R2, R1, 0 ; R2 = *q
    ADD  R2, R2, 1 ; R2 = *q+1
    STR  R2, R0, 0 ; *p = *q+1

; q = &y;
    LEA  R1, y      ; R1 = &y
    ST   R1, q      ; q = &y

; *q = *p+1;
```

```
    LDR  R2, R0, 0 ; R2 = *p
    ADD  R2, R2, 1 ; R2 = *p+1
    STR  R2, R1, 0 ; *q = *p+1
    HALT

; int x = 7, y = 10;
x   .FILL 7
y   .FILL 10

; int *p = &x, *q
p   .FILL x
q   .BLKW 1
```