Pointers and Arrays in C

CS 350: Computer Organization & Assembler Language Programming

9/22: Solved

A. Why?

- In C, pointer arithmetic lets you change a pointer to point to a nearby location.
- In C, array references can be written using pointer arithmetic and dereferencing operations (and vice versa).

B. Outcomes

After this lecture, you should

- Know how pointer arithmetic works in C.
- Know the connection between arrays and pointers in C.

C. Array Element Addresses, Plus or Minus an Integer

- Array elements are stored sequentially in memory; if `b` is an integer array and we print out the addresses of `b[0]`, `b[1]`, …, we find that they are an integer-width apart (4 bytes on my laptop). To get the address of `b[0]`, we write `&b[0]`, which is short for `&(b[0])`).
- Since `b[0]` is an integer, `&b[0] + 1` is an integer-width (4 bytes) larger than `&b[0]`. If `b` was an array of some other type, like `double` or `char`, the C compiler would substitute the width of that type instead (8 bytes for `double`, 1 byte for `char`).

```
// ptr_arith.c
//
  int b[6], i;
  // Print addresses of b[0..5]
  for (i = 0; i < 6; i++) {
      printf("&b[%d] = %p\n", i, &b[i]);
  }
```

- **Output**:

 © James Sasaki, 2016

```
&b[0] = 0x7fff57894ab4
&b[1] = 0x7fff57894ab8
&b[2] = 0x7fff57894abc
&b[3] = 0x7fff57894ac0
&b[4] = 0x7fff57894ac4
&b[5] = 0x7fff57894ac8
```

- In general, for arbitrary `k`, since `b[k]` is the `k`'th element away from `b[0]`, we have `&b[0] + k == &b[k]`. Addition is commutative, so `k + &b[0] == &b[k]` too.

```
// Print (address of b[0]) + i (i = 0..5)
// They match addresses of b[0..5]
//
for (i = 0; i < 6; i++) {
    printf("&b[0] + %d = %p\n", i,  &b[0]+i);
}
```

- **Output**:

```
&b[0] + 0 = 0x7fff57894ab4
&b[0] + 1 = 0x7fff57894ab8
&b[0] + 2 = 0x7fff57894abc
&b[0] + 3 = 0x7fff57894ac0
&b[0] + 4 = 0x7fff57894ac4
&b[0] + 5 = 0x7fff57894ac8
```

- You can also subtract from an address: `&b[k] - j == &b[k-j]`. For example, `&b[5]-1 == &b[4]`, `&b[5]-2 == &b[3]`, and so on. Subtraction is addition of the negative, so `&b[k] - j`, `&b[k] + (-j)` and `-j + &b[k]` are all `==`, for example.

```
// Print (address of b[5]) - i (i = 5, 4, ..., 0)
// They also match addresses of b[0..5]
//
for (i = 5; i >= 0; i--) {
    printf("&b[5] - %d = %p\n", i, &b[5]-i);
}
```

- **Output**

```
&b[5] - 5 = 0x7fff57894ab4
&b[5] - 4 = 0x7fff57894ab8
&b[5] - 3 = 0x7fff57894abc
&b[5] - 2 = 0x7fff57894ac0
&b[5] - 1 = 0x7fff57894ac4
&b[5] - 0 = 0x7fff57894ac8
```

 © James Sasaki, 2016

D. General Pointers, Plus or Minus an Integer

- We can also do arithmetic on general pointer expressions. E.g., if `p` is a pointer, then `p+1` and `p-1` are legal and evaluate to the address in memory of the value after `p` (for `p+1`) or before `p` (for `p-1`).
  - Again, addition is commutative, so `1+p == p+1`. Similarly, `-1+p == p-1`
- **Subtraction**: *pointer* **–** *pointer* is a `long int`, the number of values between the addresses.

```
// Subtraction: pointer - pointer gives the
// number of values between the two addresses.
// (The number is a long int.)
//
printf("&b[4] - &b[1] = %ld\n", &b[4]-&b[1]);
printf("&b[1] - &b[4] = %ld\n", &b[1]-&b[4]);
```

- **Output**:

```
&b[4] - &b[1] = 3
&b[1] - &b[4] = -3
```

E. Pointer Dereferencing & Array Indexing

- In C, pointers and arrays are interrelated concepts.
- The array name `b` is like a pointer: `b == &b[0]`.
  - It's a pointer constant (you can't assign to it): `b` = some address is illegal.
- `b == &b[0]` implies `*b == *&b[0]`, which `== b[0]`
  - So `*b` and `b[0]` are interchangeable.
  - In particular, `*b` = *e*`;` and `b[0]` = *e*`;` behave the same.
- More generally, `b + i == &b[0] + i`, which `== &b[i]`
  - So `*(b+i)` and `*&b[i]` and `b[i]` are interchangeable.
  - That means that an assignment like `b[i] = 2` is equivalent to the assignment `*(b + i) = 2`.
- This also works for general pointers: `*p` and `p[0]` are interchangeable; `*(p+i)` and `p[i]` are interchangeable.
  - In fact, the C compiler treats `b[i]` as shorthand for `*(b+i)`.

 © James Sasaki, 2016

- The table below shows various cases for how you can rewrite array code as pointer arithmetic code and vice versa. (But really, these are all basically derived from the interchangeability of `b[i]` and `b+i`.)

Array notation	***Pointer notation***
`b[0]`	`*b` or `*(b + 0)`
`b[i]`	`*(b + i)`
`b[-i]`	`*(b - i)`
`&b[0]`	`b`
`&b[i]`	`b + i`
`&b[-i]`	`b - i`
`p = &p[1];`	`p = p+1;` or `p++;`
`p = &p[-1];`	`p = p-1;` or `p--;`

 © James Sasaki, 2016

- **Example**: In the code below, the comments give an alternate notation for the code to the left. Also note accesses to `b[5]` and `b[6]`, which are semantically illegal.

```
// ptr_arith2.c: More pointer arithmetic
//
#include <stdio.h>
int main(void) {
    int b[5] = {0,0,0,0,0};
    *b = 2;          // b[0] = 2;
    *(b+3) = (*b)*2; // b[3] = b[0]*2
    printf("b[0] = %d, b[3] = %d\n", b[0], b[3]);

    // gcc -Wall will detect (e.g.) b[5] as illegal
    // but not b[i] where i == 5; accessing b[5] or
    // b[6] may cause a runtime error.
    //
    /*
    int i = 5;
    *(b+i) = 17;     // b[5] = 17 (semantic error)
    *(b+6) = 18;     // b[6] = 18 (semantic error)
    b[5] = 17;
    printf("b[%d] = %d, b[6] = %d\n", i, b[i], b[6]);
     */

    int *p;
    p = b+2;         // p = &b[2];
    p[0] = p[1];     // *p = *(p+1) (sets b[2] = b[3])
    printf("p pts to b[%ld], b[2] = %d\n", p-b, b[2]);
    p++;             // p = p+1 (makes p point to b[3])
    printf("p pts to b[%ld]\n", p-b);
    p = &p[1];       // p++
    printf("p pts to b[%ld]\n", p-b);
    p[0]--;          // (*p)--
    printf("b[4] = %d\n", b[4]);
}
```

- **Output**:

```
b[0] = 2, b[3] = 4
p pts to b[2], b[2] = 4
p pts to b[3]
p pts to b[4]
b[4] = -1
```

 © James Sasaki, 2016

- Since array names are pointer constants, code that changes pointers doesn't always translate to array code easily.
  - E.g., if `p == b+i` then `*(--p);` is like `b[--i]`, not `(--b)[i]`, which is illegal).
- Note C implementations don't check for array references being out-of-bounds (or for pointers pointing to things they shouldn't).
  - If `b` is an array of length ≤ 5, then assignments like `b[5] = 17` or `*(b+5) = 17` are syntactically legal but semantically illegal.
  - If you're lucky, the compiler will warn you about bad accesses.
  - It's your responsibility to make sure your array references and pointer dereferences are semantically valid.
- **Very scary**: Since addition is commutative, `*(b+i)` equals `*(i+b)`, so `b[i]` and `i[b]` are both syntactically correct. (eek!) Check out the following program:
- **Example**: `horrifying.c`

```
int x[5];
2[x] = 17;

printf("&x[2] = %p\n", &x[2]);
printf("x+2   = %p\n", x+2);
printf("2+x   = %p\n", 2+x);
printf("&2[x] = %p\n", &2[x]);
printf("\n");

printf("x[2]  = %d\n", x[2]);
printf("2[x]  = %d\n", 2[x]);
```

- **Output**:

```
&x[2] = 0x7fff5fbff3c8
x+2   = 0x7fff5fbff3c8
2+x   = 0x7fff5fbff3c8
&2[x] = 0x7fff5fbff3c8

x[2]  = 17
2[x]  = 17
```

 © James Sasaki, 2016

Pointers and Arrays in C

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- In C, pointer arithmetic lets you change a pointer to point to a nearby location.
- In C, array references can be written using pointer arithmetic and dereferencing operations (and vice versa).

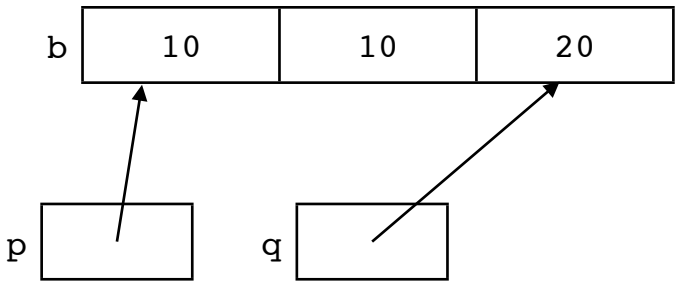
B. Outcomes

After this activity, you should be able to

- Perform pointer arithmetic in C.
- Translate between array references and their equivalent pointer expressions.

C. Problems

1. Write some C declarations and code to establish the memory diagram below. (There are multiple right answers.)



2. Using the memory diagram for Problem 1, say whether each of the following expressions causes a compile-time error (and why), or may or must cause an error when evaluated, or evaluates to true or to false.

   (a) `p == &b[0]` (b) `q == p+2` (c) `*p == *q-10` (d) `*p == *(q-10)`
   (e) `p[0] == p[1]` (f) `q == &p[2]` (g) `*p == *(p+1)` (h) `p != p+1`

 © James Sasaki, 2016

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

```
int *p, *q, w[2] = {3,4};
p = &w[0];
q = p + 1;
*p = *q + w[1];
w[1] = 5;
p = w+1;
*q = 6;
```

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

```
int x[4] = {7,8,9,10};
int *p = x, *q = &x[1];
*p = *q+5;
p[2]++;
q[-1]--;
*q = *q-1;
```

5. Rewrite the code from Problem 2 to swap the use of array notation and pointer notation. To get you started, the first three parts are already done.

   (a) `p == &b[0]` translates to `p == b`

   (b) `q == p+2` translates to `q == &p[2]`

   (c) `*p == *q-10` translates to `p[0] == q[0]-10`

   (d) `*p == *(q-10)` translates to

   (e) `p[0] == p[1]` translates to

   (f) `q == &p[2]` translates to

   (g) `*p == *(p+1)` translates to

   (h) `p != p+1` translates to

6. Repeat the previous problem, rewriting the code for Problem 3.

7. Repeat, for the code in Problem 4. (Hint about precedences: The parentheses in `(*ptr)++` are required; the parentheses in `*(ptr++)` are optional.)

8. What's the relationship between `p` and `&x[p-x]` where `x` is an array and `p` points to an element in `x`?

 © James Sasaki, 2016

Activity 8 Solution

1. `int b[4] = {10,10,20}; int *p,*q; p = &b[0]; q = &b[2];`

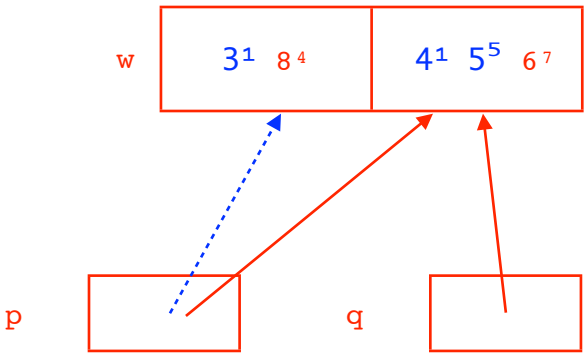
   There exist other solutions. For example,

   - You can initialize with the declaration: `int *p = &b[0];`
   - You can use b as an address: `p = b; q = b+2;`
   - You can calculate q using p: `q = p+2;`
   - You can calculate p using q: `q = &b[2]; p = q-2;`

2. a, b, c, e, f, g, h are true; d may cause a runtime error because that address might be illegal.

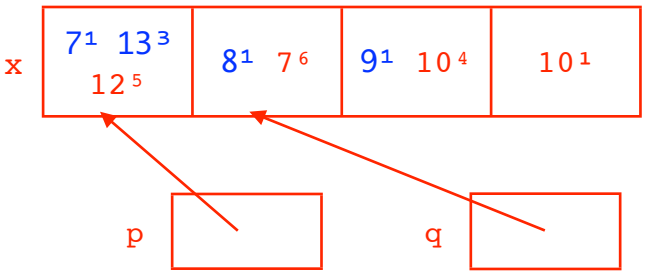
3. (Superseded values/pointers are shown in blue. The superscripts on the values indicate what line of code caused that value to be established.)

```
1  int *p, *q, w[2] = {3,4};
2  p = &w[0];
3  q = p+1;
4  *p = *q + w[1];
5  w[1] = 5;
6  p = w+1;
7  *q = 6;
```



4. (Same convention as previous problem.)

```
1  int x[4] = {7,8,9,10};
2  int *p = x, *q = &x[1];
3  *p = *q+5;
4  p[2]++;
5  q[-1]--;
6  *q = *q-1;
```



p>x0; q>x1, x0=x1+5=8+5=13, x2++ 9 to 10; x0--=12; x1-- 8 to 7

 © James Sasaki, 2016

5. Problem 2's code, rewritten:

   (a) `p == b` (b) `q == &p[2]`

   (c) `p[0] == q[0]-10` (d) `p[0] == q[-10]`

   (e) `*p == *(p+1)` (f) `q == p+2`

   (g) `p[0] == p[1]` (h) `p != &p[1]`

6. Problem 3's code, rewritten

```
int *p, *q, w[2] = {3,4};
p = w;
q = &p[1];
p[0] = q[0] + *(w+1);
*(w+1) = 5;
p = &w[1];
q[0] = 6
```

7. Problem 4's code, rewritten

```
int x[4] = {7,8,9,10};
int *p = &x[0], *q = x+1;
p[0] = q[0]+1;
(*(p+1))++
(*(q-1))--;
q[0] = q[0]-1;
```

8. Subtracting `p-x` gives the number of values between `x[0]` and `p`, so `p` points to `x[p-x]` is the value pointed to by `p`. I.e., `p == &x[p-x]`.

 © James Sasaki, 2016