Pointers
### Regular Variables (what lies beneath)

- What happens underneath when you declare a regular variable?

```
int a = 5;
char c = 'A';
```

<table>
<thead>
<tr>
<th>Memory</th>
<th>0x80493e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>'A'</td>
<td>0x80493cd</td>
</tr>
</tbody>
</table>

- Where are these stored?
Regular Variables (what lies beneath)

• What happens underneath when you declare a regular variable?

```c
int a = 5;
char c = 'A';
```

• How do we access the address at which they are stored?
  – Special operator: & (called the address operator)

```c
printf("The address of variable a is %x\n", &a);
=> prints 0x80493e0
```
Pointers

• Store a memory address
• A pointer is a higher-level version of a memory address

• In other words:
  – Integer value: stored in an int type variable
  – Floating point value: float / double
  – Memory address: pointer
Pointers

int a = 5;
char c = ‘A’;
&a is 0x80493e0
&c is 0x80493cd

• How do we store a memory address in a variable?

   int *p; // p stores the memory address of an int
   p = &a; // p is assigned the address of variable a
   – Pointer p now stores the address of variable a
     (aka, “points” to the memory location of a)
   – Important: pointers have type information!
## Pointers

<table>
<thead>
<tr>
<th>Memory</th>
<th>0x80493e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>‘A’</td>
<td>0x80493cd</td>
</tr>
</tbody>
</table>

```c
int a = 5;
char c = ‘A’;
&a is 0x80493e0
&c is 0x80493cd

int *p; // p stores the memory address of an int
p = &a; // p is assigned the address of variable a
```

- What about the other way around?
  - From a pointer, we can “extract” the value stored at that memory address

```
int i = *p; // dereference pointer p, i stores value 5
```
Pointers

```c
int i;
int *p;  /* declare p to point to type int */
*p = i;  /* dereference p - set what p points to */
    /* Memory must be allocated for this to be legal.
It is an error right now.*/

p = &i  /* Give p the value of the address of i */
    /* Why is this one legal? */

char *c = p; /* Does this work? */
    /* Warning: initialization from incompatible pointer type */
```

Reminder:
* = dereference operator
& = address operator
Important!

- Declaration: `int *p;`
- Memory is allocated to store the **pointer**
- No memory is allocated to store what the pointer points to!
- In most cases, need to allocate memory for what the pointer points to!
- Also, `p` is **not** initialized to a valid address
  
  \[ \Rightarrow *p = 10; \] is wrong, unless `p` has been already set to point to a valid memory location OR new memory has been allocated for `p`.  \[ \]
int i = 19;
int *p;
int *q;
*p = i; /*error*/
q = &i /* valid */

Address Table

<table>
<thead>
<tr>
<th>i</th>
<th>0x80493e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0x80494dc</td>
</tr>
<tr>
<td>q</td>
<td>0x80494e1</td>
</tr>
</tbody>
</table>

printf("%d", *q); => 19

Important:
&q = 0x80494e1 (where q is stored)
q = 0x80493e0 (where q points to)
*q = 19 (the value pointed to by q)
int i = 19;
int *p;
int *q;

q = &i
p = malloc(sizeof(int));
*p = i;

Address Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0x80493e0</td>
</tr>
<tr>
<td>p</td>
<td>0x80494dc</td>
</tr>
<tr>
<td>q</td>
<td>0x80494e1</td>
</tr>
</tbody>
</table>

Again:
&p = 0x80494dc  (where p is stored)
p = 0x8049530   (where p points to)
*p = 19         (the value pointed to by p)
Operators

• We’ve learnt that:
  – * = the dereference operator
  – & = the address operator

• Observe that they cancel each other out:
  – int a;   int *p;
  – *(&a) == a
  – *(&p) == p
Pointers affect variables they refer to!

– Recall the pointer syntax:

- `char *cptr;`
  • declares a pointer to a char
  • allocates space to store a pointer (to a char)
- `char c = 'a';`
- `cptr = &c;`
  • `cptr` gets the value of the address of `c`
  • the value stored at the memory location referred to by `cptr` is the address of the memory location referred to by `c;`
- `*cptr = 'b';` – dereference `cptr`
  • the address stored at `cptr` identifies the memory location where `'b'` will be stored.

• What’s the value of `c` now?
Pointers affect variables they refer to!

```c
char *cptr;
char c = 'a';
cptr = &c;
*cptr = 'b';
```

Address Table

<table>
<thead>
<tr>
<th>cptr</th>
<th>0x80493e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>0x80494dc</td>
</tr>
</tbody>
</table>

0x80493e0 0x80494dc

'b'

0x80494dc

0x804993e0 0x80494dc
Arrays vs. Pointers

• An array name in expression context decays into a pointer to the zero’th element.

• E.g.

```c
int a[3] = {1, 3, 5};
int *p;
p = a; OR p = &a[0]; (the two are equivalent)
*p = 10;
p[1] = 4;
printf("%d %d %d %d\n", a[0], a[1], a[2], *p);
```

What will this print?
Example

```c
int a[4] = {0, 1, 2, 3};
int *p;
p = a;
int i = 0;

for(i = 0; i < 4; i++) {
    printf("%d\n", *(p + i));
}
```

Why does adding 1 to `p` move it to the next spot for an int, when an int is 4 bytes?
Pointer Arithmetic

• Pointer arithmetic respects the type of the pointer.

• E.g.,

```c
int i[2] = {1, 2};
int *ip;
ip = i;
*(ip + 1) += 2;
```

(really adds 4 to `ip`)

```c
char c[2] = {'a','z'};
char *cp;
cp = c;
*(cp + 1) = 'b';
```

(really adds 1 to `cp`)

• C knows the size of what is being pointed at from the `type` of the pointer.
**Pointer Arithmetic**

• The array access operator [ ] is really only a shorthand for pointer arithmetic + dereference

• These are equivalent in C:
  
  a[i]  ==  *(a + i)

  Funny enough, *(i+a) and i[a] are also equivalent!

• C translates the first form into the second.
  
  − pointers and arrays are nearly the same in C!
Passing Arrays as Parameters

int main()
{
    int i[3] = {10, 9, 8};
    printf("sum is %d\n", sum(i)); /*??*/
    return 0;
}

int sum( What goes here? ) {
}

• What is being passed to the function is the name of the array which decays to a pointer to the first element – a pointer of type int.
Passing Arrays as Parameters

```c
int sum( int *a ) {
    int i, s = 0;
    for(i = 0; i < sizeof(a); i++)
        s += a[i]; /* this is legal */
    return s;
}
```

- How do you know how big the array is?
- Remember that arrays are not objects, so knowing where the zero’th element of an array is does not tell you how big it is.
- Pass in the size of the array as another parameter.
int sum(int *a, int size)

• Also legal is:
  int sum(int a[], int size)

• Many advise against using this form.
  – You really are passing a pointer-to-int not an array.
  – You still don't know how big the array is.
  – Outside of a formal parameter declaration int a[]; is illegal

⇒ int a; and int a[10]; are completely different things
Multi-dimensional arrays

- Remember that memory is a sequence of bytes.

```c
int a[3][3] = { {0, 1, 2}, 
                {3, 4, 5}, 
                {6, 7, 8} }; 
```

- Arrays in C are stored in row-major order
- row-major access formula

```c
int *x = (int *)a; 
int x[i][j] = *(x + i * n + j) 
where n is the row size of x
```

But use array notation!
Summary

• The name of an array can also be used as a pointer to the zero’th element of the array.
• This is useful when passing arrays as parameters.
• Use array notation rather than pointer arithmetic whenever you have an array.
Passing by value / reference

• C: “pass by value” semantics

```c
void increment (int x) {
    x++;
}

int main {
    int a = 5;
    printf("a=%d", a);  // a = 5
    increment(a);
    printf("a=%d", a);  // a = 5  Huh?
    return 0;
}
```

• What does this print?
Passing by value / reference

- **Pass by reference:**

  ```c
  void increment (int* x) {
    (*x)++;  
  }
  ```

  ```c
  int main {
    int a = 5;
    printf("a=%d", a);  a = 5
    increment(&a);
    printf("a=%d", a);  a = 6  That’s better.
    return 0;
  }
  ```

- **Pass the address of variable a (0x80493e0).**
- **It’s still pass-by-value (the value of the address!).**
Passing by value / reference

• What about a pointer?

```c
void increment (int* x) {
    (*x)++;}
```

```c
int main {
    int *p = malloc(sizeof(int));
    *p = 5;
    printf("%d", *p);  // 5
    increment(p);
    printf("%d", *p);  // 6
    return 0;
}
```

• No need for ‘&’ anymore.
Remember scanf?

```c
int i;
scanf("%d", &i);

int *p;
scanf("%d", p);  // No need for & here.
```
NEXT UP:
MEMORY ALLOCATION