Pointers

• A pointer is a higher-level version of an address.
• A pointer has type information.

```c
int i;
int *p;  /* declare p to point to type int */
*p = i;  /* dereference p – set what p points to */
p = &i   /* Give p the value of the address of i*/
char *c = p;  /* Warning: initialization from incompatible pointer type */
```
Important!

- `int *p;`
- Memory is allocated to store the **pointer**
- No memory is allocated to store what the pointer points to!
- Also, `p` is **not** initialized to a valid address or null.
- I.e., `*p = 10;` is wrong unless memory has been allocated and `p` set to point to it.
int i = 19;
int *p;
int *q;
P
*p = i; /*error*/
q = &i

Symbol 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>0x80494e0</td>
</tr>
</tbody>
</table>
int i = 19;
int *p;
int *q;

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

Symbol 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>0x80494e0</td>
</tr>
</tbody>
</table>

0x80493e0 19
0x80494dc 0x8049530
0x80494e0 0x80493e0
0x8049530 19
int main() {
    int i = 2;
    int j = 30;

    int a[4];

    int *p;
    int *q;

    p = &i;
    j = *p;
    *p = 1;

    a[0] = 10;
    a[3] = 12;
    a[i] = 11;

    Memory

    Symbol Table

    | Address | label | Value |
    |---------|-------|-------|
    | 0x23c   |       |       |
    | 0x240   |       |       |
    | 0x244   |       |       |
    | 0x248   |       |       |
    | 0x24c   |       |       |
    | 0x250   |       |       |
    | 0x254   |       |       |
    | 0x258   |       |       |
    | 0x25c   |       |       |
    | 0x260   |       |       |
int main() {
    int i = 2;
    int j = 30;

    int a[4];

    int *p;
    int *q;

    p = &i;
    j = *p;
    *p = 1;

    a[0] = 10;
    a[3] = 12;
    a[i] = 11;
}

<table>
<thead>
<tr>
<th>Address</th>
<th>label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x23c</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>0x240</td>
<td>p</td>
<td>0x258</td>
</tr>
<tr>
<td>0x244</td>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>0x248</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>0x24c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x250</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>0x254</td>
<td>j</td>
<td>2</td>
</tr>
<tr>
<td>0x258</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>0x25c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x260</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pointers and Arrays

– 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.
Pointers and Arrays

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

Symbol 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'
Arrays vs. Pointers

• An array name in expression context is interpreted as a pointer to the zero’th element.
• E.g.

    ```c
    int a[3] = {1, 3, 5};
    int *p = a;  p = &a[0];
    p[0] = 10;
    printf("%d %d\n", a[0], *p);
    ```
Example

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

for(i = 0; i < 4; i++) {
    printf("%d\n", *(p + i));
    *(p + 1) == a[1]
}

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

(really adds 4 to ip)  (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:
  
  \[ a[i] \quad == \quad *(a + i) \]

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

```c
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 < ??; 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.
Array Parameters

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.

\[
\begin{array}{cccccccc}
\text{row 0} & \text{row 1} & \text{row 2} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{array}
\]

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

• Arrays in C are stored in row-major order
• row-major access formula
\[
x[i][j] == *(x + i * n + j)
\]
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.
Structs

• A collection of related data items

```c
#define MAXNAME 8
struct record {
    char name[MAXNAME];
    int count;
};
/* The semicolon is important! It terminates the declaration. */

struct record rec1; /*allocates space for the record */
strncpy(rec1.name, ".exe", MAXNAME);
struct record *rec2;
rec2 = malloc(sizeof(struct record));
strncpy(rec2->name, ".gif", MAXNAME);
```
 structs as arguments

/* Remember: pass-by-value */
void print_record(struct record r) {
    printf("Name = %s\n", r.name);
    printf("Count=%d\n", r.count);
}
print_record(rec1);
print_record(*rec2);
Passing pointer or struct?

/* Incorrect */
void incr_record(struct record r) {
    r.count++;
}

/* Correct */
void incr_record(struct record *r) {
    r->count++;
}
struct counter {
    char name[10];
    int value;
};

void f(struct counter d) {
    d.value++;
}

void g(struct counter *d) {
    d->value++;
}

int main() {
    struct counter c;
    c.value = 1;
    c.name[0] = 'x';
    c.name[1] = '\0';
    f(c);
    printf("c.value is %d\n", c.value);
    g(&c);
    printf("c.value is %d\n", c.value);
}