C Points and Arrays

You can perform arithmetic on pointers:

- Pointer arithmetic is type-specific: the value of ptr+3 is equal to ptr plus the size of the type to which ptr points.
- Said another way, the result of ptr+x if ptr points to type type is

\[(\text{int})\text{ptr} + x \times \text{sizeof(type)}\].

### Pointer Arithmetic

The resulting address depends on the type:

<table>
<thead>
<tr>
<th>Type of ptr</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>103</td>
</tr>
<tr>
<td>short</td>
<td>106</td>
</tr>
<tr>
<td>int</td>
<td>112</td>
</tr>
<tr>
<td>struct foo</td>
<td>100 + 3*\text{sizeof(struct foo)}</td>
</tr>
</tbody>
</table>

### Arrays

Type-specific pointer arithmetic can be used to implement arrays, e.g.:

- int *array;
- array = (int *) malloc(sizeof(int) * 3);
- *(array+1) = 1;
- *(array+2) = 2;
- This fills in the first three elements of the array with the values 0, 1, and 2.
**Arrays...**

- C offers a convenient short-hand for pointer arithmetic using square-braces `[]`. The notation
  ```c
  ptr[x]
  ```
  is equivalent to
  ```c
  *(ptr+x).
  ```

- The above code can be written:
  ```c
  int *array;
  array = (int *) malloc(sizeof(int) * 3);
  array[0] = 0;
  array[1] = 1;
  array[2] = 2;
  ```

- Arrays can be automatic or global variables:
  ```c
  int array[3];
  ```
  allocates an array of three integers.

**Arrays...**

- In Mips assembly code this would be:
  ```mips
  li   $a0,   12
  jal  malloc
  li   $t0,   0
  sw   $t0,  0($v0)  # *(array) = 0
  li   $t0,   1
  sw   $t0,  4($v0)  # *(array+1) = 1
  li   $t0,   2
  sw   $t0,  8($v0)  # *(array+2) = 2
  ```

- You have to be careful to allocate enough space to hold the array. Adding the line:
  ```c
  *(array+3) = 3;
  ```

to the above code will cause the program to write the value 3 beyond the end of the array and probably trash malloc's data structures.

- A C compiler won’t catch this error!
Array Parameters...

- If a variable-size array must be passed as a parameter, don’t specify the size of the array:

```c
void foo(int count, int bar[]) {
    int i;
    for (i = 0; i < count; i++)
        bar[i] = i;
}
void main(void) {
    int array[3];
    foo(3, array);
}
```

Array Parameters...

- Finally, since an array is the same as a pointer, you can specify an array parameter as such:

```c
void foo(int count, int *bar) {
    int i;
    for (i = 0; i < count; i++)
        bar[i] = i;
}
void main(void) {
    int array[3];
    foo(3, array);
}
```

Arrays...

- Addresses can be taken of individual array elements:

```c
&array[1]
```

is the address of the 2nd element in the array.

```c
&array[0]
```

is the same address as `array`.

- Arrays of pointers are also possible:

```c
int *array[3];
```

- This allocates an array of three pointers to integers, not three integers.

Array Parameters

- Arrays can be passed as parameters, but because an array variable is simply a pointer, the array itself is passed by reference, not by value. A copy of the array is not made for the callee.

```c
void main(void) {
    int array[3];
    bar[0] = 0;
    foo(array);
    bar[1] = 2
}
```

- After `foo` is called, the values in `array` have been changed in the parent.
Multi-dimensional Arrays

- Multi-dimensional arrays are arrays of arrays:
  ```
  int matrix[10][5];
  ```
  matrix is an array of 10 arrays, each containing 5 elements. The array is organized in memory so that
  `matrix[0][1]` is adjacent to `matrix[0][0]`.
- A multi-dimensional array can be initialized:
  ```
  int x[2][3] = {
      {0, 1, 2},
      {3, 4, 5}
  };
  ```

Multi-dimensional Arrays...

- When passing a multi-dimensional array as a parameter all but the first dimension must be specified so the
  correct address calculation code is generated:
  ```
  void foo(int x[][3]);
  ```
- Arrays of pointers to arrays are often used instead of multi-dimensional arrays:
  ```
  int *foo[2];
  ```
  is an array of two pointers to integers.

Initializing Arrays

- You can’t use the `[[]]` notation when defining a variable. You must specify the size (the C compiler
  needs to know how much memory to allocate), unless you initialize the array.
- Arrays are initialized by setting them to a brace-enclosed, comma-separated list of values:
  ```
  int totals[3] = {10, 17, 42};
  ```
- You can leave out the size if you initialize an array:
  ```
  int totals[] = {10, 17, 42};
  ```

Initializing Arrays...

- Older C compilers won’t let you initialize an automatic array variable, i.e. you can’t declare the above arrays in
  a function. You can, however, if the array is static:
  ```
  static int totals[] = {10, 17, 42};
  ```
- You can also initialize arrays of pointers:
  ```
  char *colors[3] = {"red", "green", "blue"};
  ```
Multi-dimensional Arrays...

- We can then create two sub-arrays, possibly of different size, and index them like a multi-dimensional array:

```c
int int
foo[0] = a;
foo[1] = b;
```

Slide 16–16

Multi-dimensional Arrays...

- These arrays of pointers to arrays are especially useful for arrays of strings:

```c
char *colors[3] = {"red", "green", "blue"};
```

colors is an array of pointers to arrays of characters, each a different size:

```c
colors[0][0] = 'r';
colors[1][0] = 'g';
colors[2][0] = 'b';
```

Slide 16–17

FirstNonZeroElement.c

```c
#include <stdio.h>

int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
    array_ptr = array;

    while ((*array_ptr) != 0)
        ++array_ptr;

    printf("# of elements before zero %d\n",
            array_ptr - array);
    return (0);
}
```

StringTokenizer.c

```c
#include <stdio.h>

main()
{
    char str[80];
    char array[10];
    char *p, *q;

    printf("Enter a sentence: "); gets(str);
    p = str;

    while (*p) {
        q = token;
        while (*p != ' ' & *p) {
            *q = *p;
            q++; p++;
        }
        if (*p) p++;
        *q = '0';
        printf("%s\n", token);
    }
}
```

Slide 16–18

Slide 16–19
Readings and References

• More online C courses:
  http://www.physics.unsw.edu.au/~mcba/c001.html
  http://www.oreilly.com/catalog/pcp3/ch03.html

Slide 16–20