University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

March 27, 2001

C Pointers and Arrays

Pointer Arithmetic

- You can perform arithmetic on pointers:

```
ptr2 = ptr1 + 3;
```

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

```
((int) ptr) + x * sizeof(type).
```


Pointer Arithmetic...

The resulting address depends on the type:

```
ptr = 100;
ptr = ptr + 3;
```

Type of ptr	Result
`char *`	103
`short *`	106
`int *`	112
`int **`	112
`struct foo *`	100 + 3*sizeof(struct foo)

Arrays

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

```
int *array;
array = (int *) malloc(sizeof(int) * 3);
*(array) = 0;
*(array+1) = 1;
*(array+2) = 2;
```

- This fills in the first three elements of the array with the values 0,1, and 2.

Slide 16–4

Arrays...

- In Mips assembly code this would be:

```
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
```

Slide 16–5

Arrays...

- You have to be careful to allocate enough space to hold the array. Adding the line:

```
*(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!

Slide 16–6

Arrays...

- C offers a convenient short-hand for pointer arithmetic using square- braces ⌜[]⌝. The notation

```
ptr[x]
```

  is equivalent to

```
*(ptr+x).
```

Slide 16–7

Arrays...

- The above code can be written:

```
int *array;
array = (int *) malloc(sizeof(int) * 3);
array[0] = 0;
array[1] = 1;
array[2] = 2;
```

- Arrays can be automatic or global variables:

```
int array[3];
```

  allocates an array of three integers.

Slide 16–8

Arrays. . .

- Addresses can be taken of individual array elements:

  `&array[1]`

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

  `&array[0]`

  is the same address as `array`.

- Arrays of pointers are also possible:

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

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

Slide 16–9

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.

`void main(void) {`	`void foo(int bar[3]) {`
`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.

Slide 16–10

Array Parameters. . .

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

```
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);
}
```

Slide 16–11

Array Parameters. . .

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

```
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);
}
```

Slide 16–12

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};
```

Slide 16–13

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"}
```

Slide 16–14

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}
 };
```

Slide 16–15

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.

Multi-dimensional Arrays...

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

```
int              *foo[2];
int              a[3];
int              b[4];
foo[0] = a;
foo[1] = b;
foo[0][0] = 0;   /* a[0] = 0; */
foo[0][1] = 1;   /* a[1] = 1; */
foo[1][3] = 3;   /* b[3] = 3; */
foo[0][3] = 3;   /* ERROR! */
```


Multi-dimensional Arrays...

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

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

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

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


FirstNonZeroElement.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

```
#include <stdio.h>
main() {
    char str[80];
    char token[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);
    }
}
```

Readings and References

- More online C courses:
  `http://www.phys.unsw.edu.au/~mcba/c001.html`,
  `http://www.oreilly.com/catalog/pcp3/chapter/ch13.html`.