

05 - Lecture - arrays

Reading

- Chapters 5 and 6

Array basics

- declaration and access

```
int a[10]; // 10 integers in contiguous memory, from 0th to 9th
```

```
a[0] = 100;
```

```
a[9] = 200;
```

```
a[10] = 300; // out of bounds: compiles, but runtime error
```

- initialization

```
int a[] = { 100, 200, 300 }; // same as int a[3] = { ...
```

```
int b[10] = { -1 }; // b[1] - b[9] are initialized to zero
```

```
char c[] = "abc"; // short-hand for: char c[] = {'a','b','c','\0'};  
// and it's DIFFERENT from: char *c = "abc";
```

- multi-dimensional arrays

```
double matrix[300][200];
```

- sizeof operator

```
int x;
```

```
int a[10];
```

```
printf("%d\n", sizeof(x));
```

```
printf("%d\n", sizeof(int));
```

```
printf("%d\n", sizeof(a[0]));
```

```
printf("%d\n", sizeof(a));
```

Arrays and pointers

```
int a[10]; // 10 integers in contiguous memory, from 0th to 9th
```

```
int *p = &a[0]; // p points to the 1st element of a
```

- you can access the array elements by moving the pointer:

p + 1 means p + sizeof(*p) in terms of number of bytes.

```
x = *p; // *p is same as a[0]
```

```
x = *(p+1); // *(p+1) is same as a[1]
```

```
...
```

```
x = *(p+9); // *(p+9) is same as a[9]
```

```

for (int i = 0; i < 10; i++)
    printf("%d\n", *p++);

```

- the array name ("a" in our example) is converted to a pointer to the 1st element in most expressions (sizeof is one of the exceptions)

```

x = *a;        // *a    is same as a[0]
x = *(a+1);    // *(a+1) is same as a[1]
x = *(a+9);    // *(a+9) is same as a[9]

```

In fact, compiler automatically converts `a[b]` to `*(a+b)`

- but unlike a pointer, an array name is a constant, not a variable:

```

a++; // compiler error

```

- when an array name is passed as a function argument, it is converted to a pointer to the 1st element; the following 3 function declarations are equivalent:

```

int foo(int a[10]);
int foo(int a[]);
int foo(int *a);

```

- some more examples of pointer arithmetic:

```

int *p = a;
int *q = &a[9];
q--;
int x = q - p; // what is the value of x?

```

- To summarize:

These are the same:

```

a
&a[0]

```

And these are the same:

```

a+5
&a[5];
&a[0]+5;

```

And all these are same:

```

a[0]
*a
*&a[0]
*(a+0)

```

How about these? Try them!

```

*(0+a)
0[a]

```

char array, aka the string

Recall:

```
char c[] = "abc"; // short-hand for: char c[] = {'a','b','c','\0'};
```

Everywhere else, "abc" is an expression whose value is a pointer:

```
char *p = "abc"; // p points to the 1st element of 4-char array
```

String literals such as "abc" are stored in code section or static data section of the process memory, depending on compiler and OS.

```
*p = 'A'; // result undefined - probably segmentation fault
```

Different ways to implement strcpy from K&R2, p105-106:

```
while ((s[i] = t[i]) != 0) i++;  
while ((*s = *t) != 0) { s++; t++; }  
while ((*s++ = *t++) != 0) ;
```

Heap memory allocation

Recall:

- stack arrays are transient
- static arrays are fixed in size

We want dynamic, yet persistent arrays.

malloc(n) allocates n bytes of memory on the heap, and returns a pointer to the beginning of the memory.

```
int *p = (int *) malloc(100 * sizeof(int));  
  
// malloc returns NULL if it cannot allocate the requested memory  
if (p == NULL) {  
    perror("malloc failed");  
    exit(1);  
}  
  
// initialize all elements to 0  
for (int i = 0; i < 100; i++)  
    p[i] = 0;  
  
// another way to do the same thing  
memset(p, 0, 100 * sizeof(int));
```

free() deallocates the memory block previously returned by malloc.

```
free(p);
```

Pointer to pointer

Array of pointers:

```
char *a[] = { "hello", "world" };

char **p = a;
printf("%s %s\n", p[0], p[1]);

// the following is a little different
// see K&R2, p114 for an illuminating picture
char a[][10] = { "hello", "world" };
```

Command line arguments are passed to `main()` as an array of char pointers (see K&R2, p115 for a picture).

Different ways to implement 'echo' program (K&R2, p115):

```
for (i = 1; i < argc; i++)
    printf("%s\n", argv[i]);

while (--argc > 0)
    printf("%s\n", *++argv);

argv++;
while (*argv)
    printf("%s\n", *argv++);
```