OPERATING SYSTEMS

POINTERS PART 2

Benedict R. Gaster

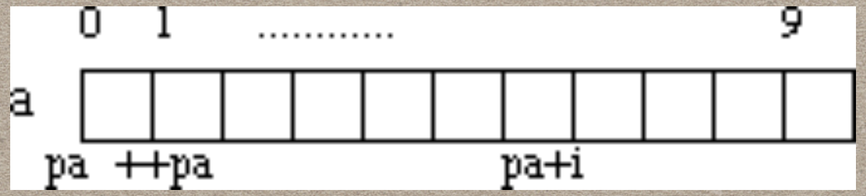
POINTERS AND ARRAYS

Pointers and arrays are very closely linked in C.

Hint: think of array elements arranged in consecutive memory locations.

Consider the following:

```
int a[10]
int x;
int *pa;

pa = &a[0];  // pa pointer to address of a[0]

x = *pa; // x = contents of pa (a[0] in this case)
```



REMEMBER: There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

POINTERS AND ARRAYS

C however is much more subtle in its link between arrays and pointers.

For example, we can just type

```
pa = a;
```

instead of

```
pa = &a[0]
```

and

a[i] can be written as *(a + i).

$$\&a[i] \equiv a + i$$

$$pa[i] \equiv *(pa + i).$$

POINTERS AND ARRAYS

However pointers and arrays are different:

- A pointer is a variable. We can do pa = a and pa++.
- An Array is not a variable. a = pa and a++ ARE ILLEGAL.

This stuff is very important. Make sure you understand it. We will see a lot more of this in Rust and the restrictions it places on pointers, via its Borrow checker.

ARRAYS AND FUNCTIONS

When an array is passed to a function what is actually passed is its initial elements location in memory.

So:

strlen(s) ≡ strlen(&s[0])

This is why we declare the function:

```
int strlen(char s[]);
```

An equivalent declaration is : `int strlen(char *s);`

since char s[] ≡ char *s.

ARRAYS OF POINTERS

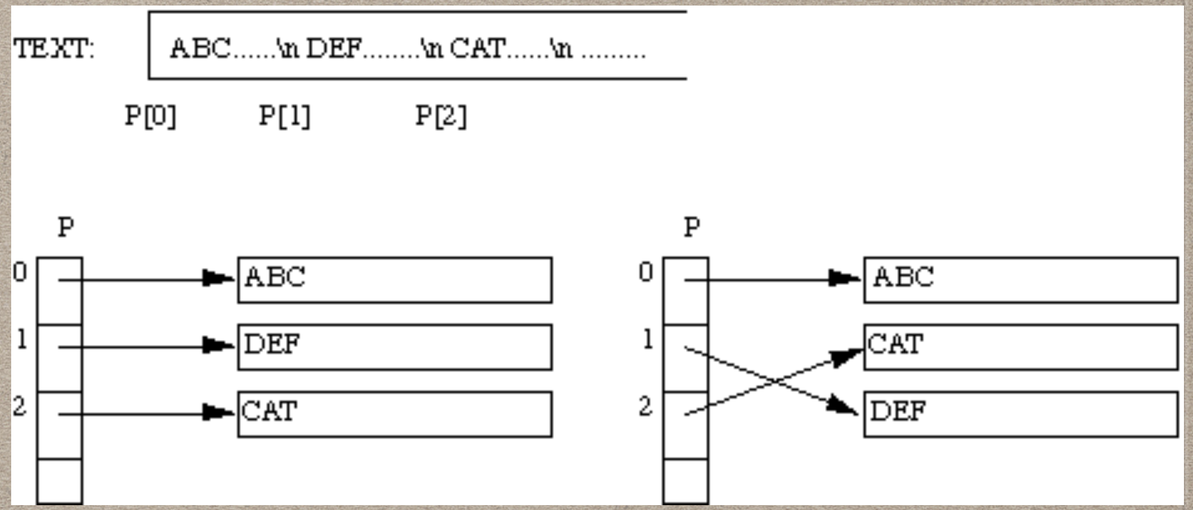
It is possible to have arrays of pointers since pointers are variables.

A simple example is sorting lines of different length.

Arrays of Pointers are a data representation that will cope efficiently and conveniently with variable length text lines.

How can we do this?:

- Store lines end-to-end in one big char array. \n will delimit lines.
- Store pointers in a different array where each pointer points to 1st char of each new line.
- Compare two lines using strcmp() standard library function.
- If 2 lines are out of order -- swap pointer in pointer array (not text).



POINTERS AND STRUCTURES

These are straight forward and are easily defined.

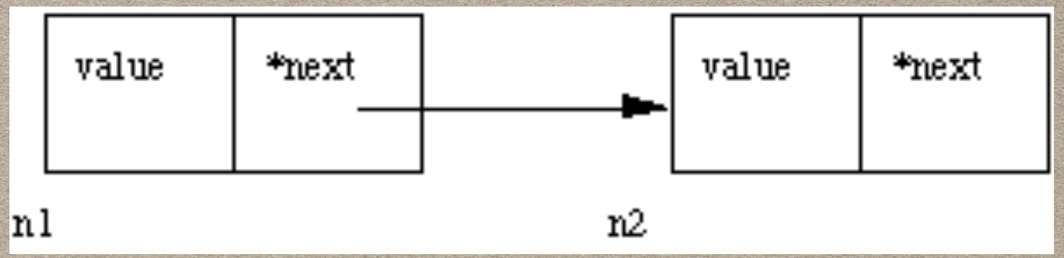
```
struct COORD {
    float x;
    float y;
    float z;
} pt;

struct COORD *pt_ptr;

pt_ptr = &pt; // assigns pointer to pt
```

the -> operator lets us access a member of the structure pointed to by a pointer.

```
pt_ptr->x = 1.0;

pt_ptr->y = pt_ptr->y - 3.0;
```

LINKED LISTS

```
typedef struct {
    int value;
    ELEMENT *next;
} ELEMENT;

ELEMENT n1;
ELEMENT n2;

n1.next = &n2;
```



POINTERS TO FUNCTIONS

```
int (*pf) ();
```

This simply declares a pointer *pf to function that returns an int. No actual function is ***pointed*** to yet.

If we have a function int f() then we write:

```
pf = &f;
```

POINTERS TO FUNCTIONS

For compiler prototyping to fully work it is better to have full function prototypes for the function and the pointer to a function:

```
int f(int);
int (*pf) (int) = &f;
```

f() returns an int and takes one int as a parameter.

We can then do things like:

```
ans = f(5);
ans = pf(5);
```

DYNAMIC MEMORY ALLOCATION

Dynamic allocation is a key feature of C/C++ and other system based programming, and is difference when compared to Python, Javascript, etc. It enables one to create data types and structures of any size and length to suit our programs need within the program.

We will look at two common examples are:

- dynamic arrays
- dynamic data structure, ***e.g.*** linked lists

MALLOC, SIZEOF, AND FREE

The Function *malloc* is most commonly used to attempt to ``grab'' a continuous portion of memory. It is defined by:

```
void *malloc(size_t number_of_bytes);
```

That is to say it returns a pointer of type void,that is the start in memory of the reserved portion of size number_of_bytes. If memory cannot be allocated a NULL pointer is returned.

Since a `void *` is returned the C standard states that this pointer can be converted to any type. The size_t argument type is defined in stdlib.h and is an ***unsigned type***.

MALLOC, SIZEOF, AND FREE

Try to get 100 bytes and assigns the start address to cp.

```
char *cp;
cp = malloc(100);
```

Usually we use the sizeof() function to specify the number of bytes

```
int *ip;
ip = (int *) malloc(100*sizeof(int));
```

MALLOC, SIZEOF, AND FREE

Memory allocated with malloc must be freed,
once it is not longer required.

```
int *ip;
ip = (int *) malloc(100*sizeof(int));
...
// valid to use ip (although must check for NULL)


...
free(ip)
// no longer valid to use ip
```