Lecture 6
Pointers-Part 2

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

Section 7.1.*

- Pointer Arithmetic
- Compare *++*cp* and **cp*++
- What is *NULL*?
- What is *void* *

FUNCTIONS AND POINTERS

```
double *maxp(double *xp, double *yp)
  {
    return *xp >= *yp ? xp : yp;
  }
```

given that

```
    double u = 1, v = 2, *mp;
```

the statement

```
    mp = maxp(&u, &v);
```

makes `mp` point to `v`.

Call by Reference

Call by reference can be effected by passing pointers to the variables as arguments to the function. These pointers can then be used by the called function to access the argument variables and change them. Thus, given that the function `exchange` is defined as

```
void exchange(int *ip, int *jp)
  {
    int t;

    t = *ip, *ip = *jp, *jp = t;
  }
```

When a pointer is passed as an argument to a function, the pointer itself is copied but the object pointed to is not copied. Hence, any change made to the pointer parameter by the called function does not affect the pointer supplied as argument to the function, which is consistent with the call by value parameter passing. However, using the pointer supplied as the argument, the called function can access and modify the object pointed to by the pointer in the calling function.

```
#include <stdio.h>

void change(int *ip)
  {
    ++(*ip);  /* increment what ip  is pointing to */
    ip  = NULL;  /* set ip  to NULL  */
  }

int main(void)
  {
    int i = 0, *ip, *pi;

    ip = pi = &i;

    change(ip);
    printf("new value of i = %d\n", i);
    if (ip == pi) printf("no change in ip\n");

    return 0;
  }
```

The output of this program is:

```
new value of i = 1
no change in ip
```

```
void exchange (int i, int j)
  {
    int t;

    t =*(&i), *(&i) =*(&j), *(&j) =t;
  }
```

```c
#include <math.h>
#include <stdio.h>
#define PI 3.1415927

void polar(float x, float y,
           float *pr, float *ptheta);

int main(void)
  {
    float x, y;
    float r, theta;

    scanf("%f %f", &x, &y);

    / * provide addresses of r  and theta to polar * /
    polar (x, y, &r, &theta);

    printf ("%f %f\n", r, theta);
    return 0;
  }
```

```
void polar(float x, float y,
            float *pr, float *ptheta)
  {
    if (x == 0 && y == 0) /* origin */
        *pr = *ptheta = 0;
    else
      {
        *pr = sqrt(x*x + y*y);
        *ptheta = atan2(y,x); .
      }
  }
```

ARRAYS AND POINTERS

Any operation that can be achieved by array subscripting can also be done with pointers.

C treats a variable of type "array of T" as "pointer to *T*", whose value is the address of the first element of the array. Thus, given that

```
char c[MAX];
```

the array name `c` is a synonym for the pointer to the first element of the array, that is, the value of `c` is the same as `&c [ 0 ]`. Hence, if `cp` is a character pointer, the two assignments

```
cp = c;
```

and

```
cp = &c [0] ;
```

are equivalent.

Array subscripting has also been defined in terms of pointer arithmetic. Thus, the expression

```
c [i]
```

is defined to be the same as

```
*(c+i)
```

Applying the operator & to both, it follows that

```
&c[i]
```

and

```
c+i
```

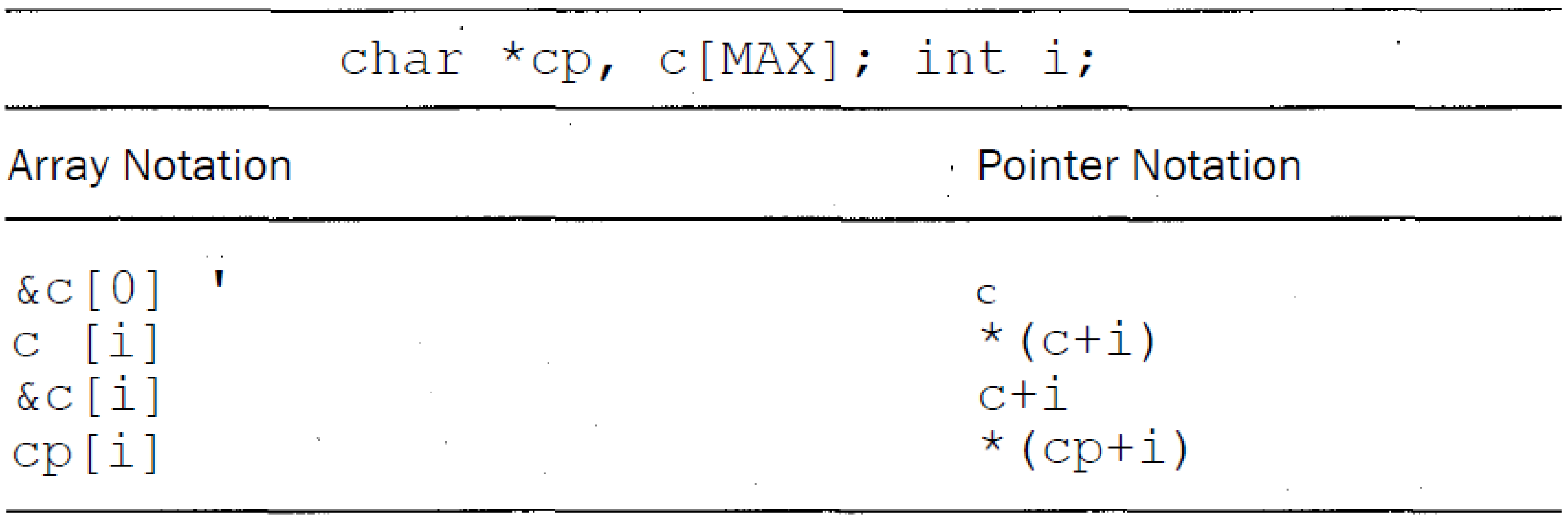
are also equivalent.

This equivalence means that the pointers may also be subscripted. Thus,

```
cp [i]
```

is equivalent to

```
*(cp+i)
```



`char *cp, c[MAX]; int i;`

Array Notation	Pointer Notation
`&c[0]`	`c`
`c [i]`	`*(c+i)`
`&c[i]`	`c+i`
`cp[i]`	`*(cp+i)`

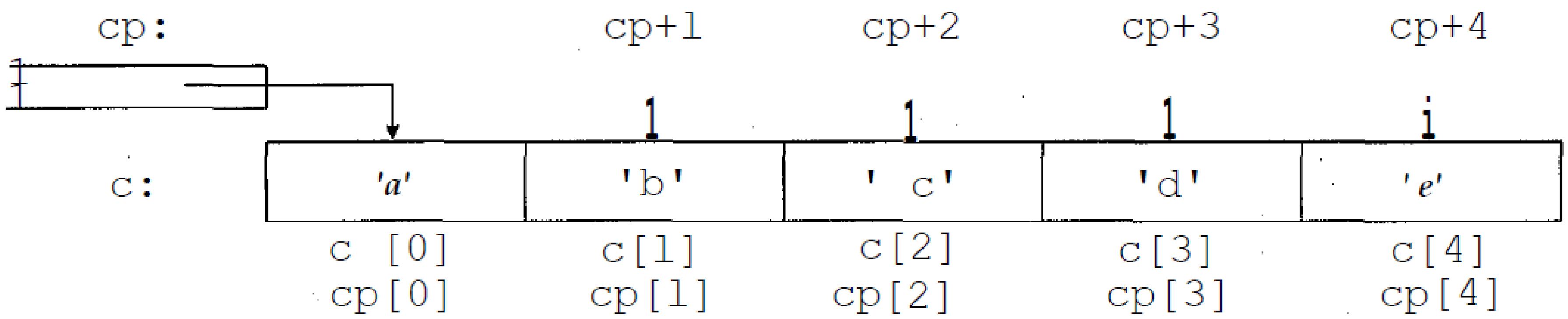
For example, given that

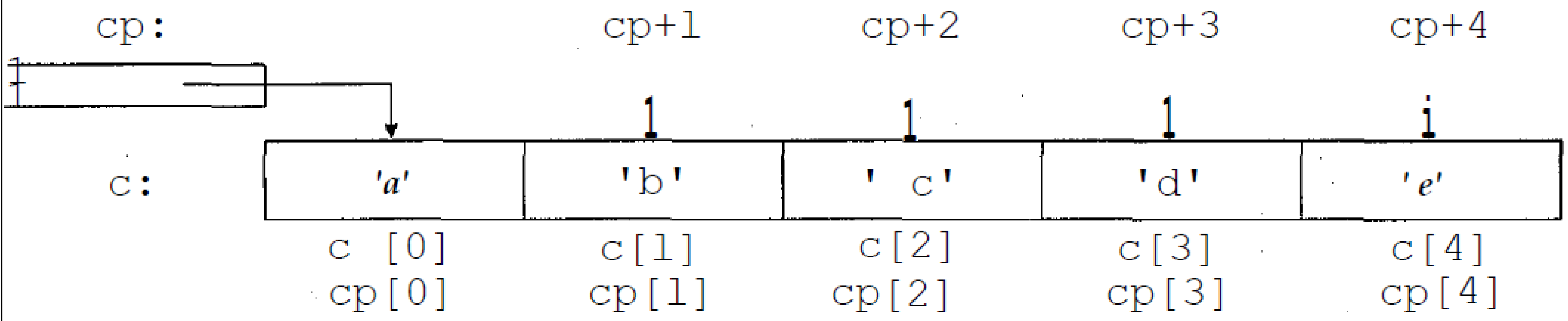
```
    char c [ 5] =    { 'a', 'b', 'c', 'd', 'e' };
    char *cp;
```

and

```
    cp =  c;
```

we have





c[0]	is	'a',	and so are	*cp and cp[0]
c[1]	is	'b',	and so are	*(cp+1) and cp [1].
c[2]	is	'c',	and so are	*(cp+2) and cp [2].
c[3]	is	'd',	and so are	*(cp+3) and cp [3].
c[4]	is	'e',	and so are	*(cp+4) and cp[4]

Although array names and pointers have strong similarities, array names are not variables. Thus, if `a` is an array name and `ap` a pointer, then the expressions like

```
    a = ap;
```

and

```
    a++;
```

are illegal. One way of viewing array names is that they are constant pointers whose values may not be changed.

Arrays as Function Arguments

An array name, as we have seen in Section 6.2.2, can be passed as argument to a function. When an array name is specified as argument, it is actually the address of the first element of the array that is passed to the called function. Hence, a formal parameter declared to be of type "array of T" is treated as if it were declared to be of type "pointer to *T*". Thus, the declaration for `x` in

```
void array_cuberoot(double x[], int length);
```

can equivalently be written as

```
void array_cuberoot(double *x, int length);
```

```
int max(int a[], int length)
  {
    int i, maxv;

    for (i = 1, maxv = a[0]; i < length; i++)
        if (a[i] > maxv) maxv = a[i];
    return maxv;
  }
```

Using pointers, this function can equivalently be written as

```
int max(int *a, int length)
{
    int i, maxv;

    for (i = 1, maxv = *a; i < length; i++)
        if (*(a+i) > maxv)' maxv = *(a+i);
    return maxv;
 }
```

Since pointers can be subscripted, the preceding function can also be written as

```
int max(int *a, int length)
  {
    int i, maxv;

    for (i = 1, maxv = a[0]; i < length; i++)
        if (a[i] > maxv) maxv = a[i];
    return maxv;
  }
```

Yet another way to write this function is

```
int max(int *a, int length)
  {
    int maxv, *end = a + length;

    for (maxv = *a; a < end; a++)
        if (*a > maxv) maxv = *a;
    return maxv;
  }'
```

In this version, `end` is initialized to point to the first element beyond the end of the array, and the pointer variable `a` is repeatedly incremented to step through all the elements of the array.

With any of the preceding function definitions and the declaration

```
int inp [5] = {10, 6, 4, 2, 8};
```

the function call

```
printf("%d\n", max(inp, 5));
```

causes 10 to be printed.

```c
#include <math.h>

void array_cuberoot(double x[], int length)
  {
     int i;

     for (i =0; i < length; i++)
          x[i] = pow(x[i], 1.0/3.0);
  }
```

is equivalent to

```
#include <math.h>

void array_cuberoot(double *x, int length)
  {
    int i;

    for (i =0; i < length; i++)
          *(x+i) = pow(*(x+i), 1.0/3.0);
  }


double y[5] = {0.1, 0.2, 0.3, 0.4, 0.5);
array_cuberoot(y, 5);
```

STRINGS AND POINTERS

String manipulation is often a large part of any programming task. C does not have a built-in string data type, but uses null-terminated arrays of characters to represent strings. To create a *string variable,* you must allocate sufficient space for the number of characters in the string and the null character ' \ 0'. The null character helps find the end of a string. For example, you can define a string variable capable of storing the word "r2d2" as follows:

```
char robot [5];
```

This variable, as we discussed in Section 6.1.3, can be initialized at the time of its definition as

```
char robot [5] = {'r', '2', ' d', '2', '.\0'};
```

or simply as

```
char robot [5] = "r2d2";
```

A *string constant,* syntactically, is a sequence of characters enclosed in double quotation marks. The storage required for a string constant is one character more than the number of characters enclosed within the quotation marks, since the compiler automatically places the null character at the end. The type of a string constant is an "array of `char`." However, when a string constant appears anywhere (except as an initializer of a character array or an argument to the `sizeof` operator), the characters making up the string together with the null character are stored in consecutive memory locations, and the string constant becomes a pointer to the location where the first character of the string has been stored.

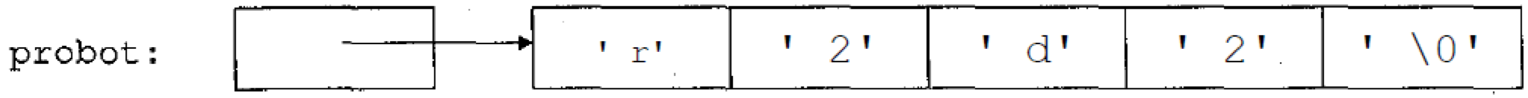
For example, given the declaration for a character pointer

```
char *probot;
```

after the assignment

```
probot = "r2d2";
```

the pointer variable `probot` contains the address of the first memory location where the string `"r2d2"` has been stored, as shown in the following:



The characters of the string can then be accessed using this pointer.

Before giving examples of how strings can be manipulated using pointers, we must emphasize the difference between a character array initialized to a string constant and a character pointer initialized to a string constant. The name of a character array, such as `robot`, always refers to the same storage, although the individual characters within the array may be changed by new assignments. On the other hand, a character pointer, such as `probot`, may be reassigned to point to new memory locations, but if it is pointing to a string constant and you try to modify the contents of the locations accessible through this pointer, the result is undefined.

To illustrate string handling, we write a function `strlen` that computes the length of a given string, and returns the number of characters that precede the terminating null character '\ 0'. This function can be written using arrays as

```
int strlen(char str[])
  {
    int i = 0;

    while (str[i] != '\0') i++;
    return i;
  }
```

Here is a pointer version of the same function:

```
int strlen(char *str)
  {
    char *first = str;

    while (*str != '\0') str++;
    return str - first;
  }
```

As another example of a string handling function, consider the following function `strcpy` that copies a string into another string:

```
void strcpy(char *to, char *from)
  {
    while(*to = *from) to++, from++;
  }
```

The preceding function can be further abbreviated to

```
void strcpy(char *to, char *from)
    {
        while(*to++ = *from++);
    }
```

Finally, here is a function strcat that concatenates the string addend to the end of the string str :

```
void strcat(char *str, char *addend)
  {
    strcpy(str+strlen(str), addend);
  }
```

Note that str+strlen (str) points to the position in str that contains the terminating '\0', and strcpy copies addend into str starting at this position.

Library Functions for Processing Strings

Although C does not have built-in string data type, it provides a rich set of library functions for processing strings. The standard header file <string. h> contains the prototypes for these functions. We now discuss the most frequently used of these functions.

In the following discussion, size_t is an unsigned integral type defined in the standard header file <stddef. h>. String parameters that are not modified by the function are declared to be of type const char *.

```
size_t strlen(const char *s);
```

computes the length of the string s, and returns the number of characters that precede '\ 0'.

char *strcpy(char *sl, const char *s2) ;

copies the string s2 to sl (including '\0'), and returns sl

```
char *strncpy (char *s1, const char *s2, size_t n);
```

copies at most the first *n* characters of the string s2 to *si* (stopping after ' \ 0' has been copied and padding the necessary number of ' \ 0' at the end, if *s2* has fewer than *n* characters), and returns `s1`

```
char *strcat (char *s1, const char *s2);
```

concatenates the string s2 to the end of *si,* placing '\0' at the end of the concatenated string, and returns s1 `char *strncat (char *sl, const char *s2, size_t n);`

concatenates at most the first *n* characters of the string s2 to the end of *sl* (stopping before a ' \ 0' has been appended), places ' \ O ' at the end of the concatenated string, and returns *sl*.

```
int strcmp(const char *sl, const char *s2) ;
```

compares the string sl to s2, and returns a negative value if sl is lexicographically less than s2, zero if sl is equal to s2, and a positive value if sl is lexicographically greater than s2.

```
int *strchr(const char *S, int c) ;
```

locates the first occurrence of *c* (converted to a `char)` in the string `s`, and returns a pointer to the located character if the search succeeds and `NULL` otherwise.

```
int *strrchr (const char *s, int C) ;
```

locates the last occurrence of `c` (converted to a `char)` in the string `s`, and returns a pointer to the located character if the search succeeds and `NULL` otherwise.

`char si[MAX], s2[MAX];`

Statement	Result
`printf("%d",    strlen ("cord") )';`	4
`printf("%s",    strcpy(sl, "string"));`	string
`printf("%s",    strncpy(s2, "endomorph", 4));`	endo
`printf("%s",    strcat(si, s2));`	stringendo
`printf("%d",    strcmp (si, s2));`	1
`printf("%d".    strncmp(sl+6, s2, 4));`	0
`printf("%s",    strchr (si, 'n'));`	ngendo
`printf ("%s".   strrchr (si, 'n' ) ) ;`	ndo