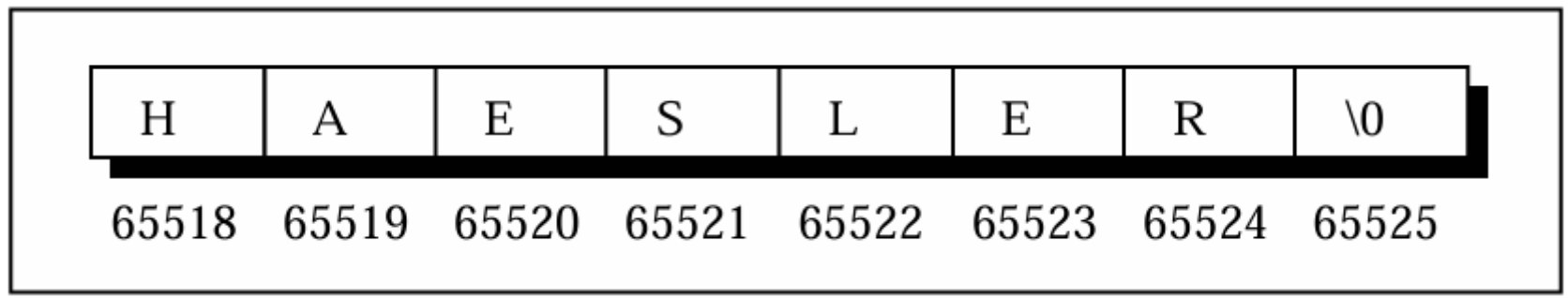
Strings

What are Strings

- The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array
- Character arrays are many a time also called strings
- Many languages internally treat strings as character arrays, but somehow conceal this fact from the programmer
- Character arrays or strings are used by programming languages to manipulate text such as words and sentences
- A string constant is a one-dimensional array of characters terminated by a null (‘\0’). For example,
- char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
- Each character in the array occupies one byte of memory and the last character is always ‘\0’

- It looks like two characters, but it is actually only one character, with the \ indicating that what follows it is something special
- ‘\0’ is called null character
- Note that ‘\0’ and ‘0’ are not same
- ASCII value of ‘\0’ is 0, whereas ASCII value of ‘0’ is 48
- Figure shows the way a character array is stored in memory.
- Note that the elements of the character array are stored in contiguous memory locations.
- The terminating null (‘\0’) is important, because it is the only way the functions that work with a string can know where the string ends.
- In fact, a string not terminated by a ‘\0’ is not really a string, but merely a collection of characters.


H
A
E
S
L
E
R
\0
65518
65519
65520
65521
65522
65523
65524
65525

String Initialization

- C provides a shortcut for initializing strings
- char name[ ] = "HAESLER" ;
- Note that, in this declaration ‘\0’ is not necessary. C inserts the null character automatically.

```
/* Program to demonstrate printing of a string */
main( )
{
      char  name[ ] = "Klinsman" ;
      int  i = 0 ;

      while ( i <= 7 )
      {
            printf ( "%c", name[i] ) ;
            i++ ;
      }

}
```

And here is the output...

```
Klinsman
```

```
main( )
{
      char  name[ ] = "Klinsman" ;
      int  i = 0 ;

      while ( name[i] != `\0' )
      {
            printf ( "%c", name[i] ) ;
            i++ ;
      }
}
```

And here is the output...

Klinsman

```
main( )
{
        char  name[ ] = "Klinsman" ;
        char  *ptr ;

       ptr = name ;  /* store base address of string */

       while ( *ptr != `\0' )
       {
              printf ( "%c", *ptr ) ;
              ptr++ ;
       }
}
```

More about Strings

- the character array elements can be accessed exactly in the same way as the elements of an integer array.
- Thus, all the following notations refer to the same element:
- name[i]
- *(name + i)
- *(i + name)
- i[name]

More about Strings

- Even though there are so many ways (as shown above) to refer to the elements of a character array, rarely is any one of them used.
- This is because printf( ) function has got a sweet and simple way of doing it, as shown below.
- Note that printf( ) doesn’t print the ‘\0’.

```
main( )
{
      char  name[ ] = "Klinsman" ;
      printf ( "%s", name ) ;
}
```

```
main( )
{
      char  name[25] ;

      printf ( "Enter your name " ) ;
      scanf ( "%s", name ) ;
      printf ( "Hello %s!", name ) ;
}
```

And here is a sample run of the program...

```
Enter your name Debashish
Hello Debashish!
```

scanf() and strings

- Note that the declaration char name[25] sets aside 25 bytes under the array name[ ], whereas the scanf( ) function fills in the characters typed at keyboard into this array until the enter key is hit.
- Once enter is hit, scanf( ) places a ‘\0’ in the array.
- Naturally, we should pass the base address of the array to the scanf( ) function.

scanf() and strings

- While entering the string using scanf( ) we must be cautious about two things:
- The length of the string should not exceed the dimension of the character array.
- This is because the C compiler doesn’t perform bounds checking on character arrays.
- Hence, if you carelessly exceed the bounds there is always a danger of overwriting something important

- scanf( ) is not capable of receiving multi-word strings.
- Therefore names such as ‘Debashish Roy’ would be unacceptable.
- The way to get around this limitation is by using the function gets( ).
- The usage of functions gets( ) and its counterpart puts( ) is shown below.

```
main( )
{
    char  name[25] ;

    printf ( "Enter your full name " ) ;
    gets ( name ) ;
    puts ( "Hello!" ) ;
    puts ( name ) ;
}
```

And here is the output...

```
Enter your name Debashish Roy
Hello!
Debashish Roy
```

Explanation

- puts( ) can display only one string at a time (hence the use of two puts( ) in the program above).
- Also, on displaying a string, unlike printf( ), puts( ) places the cursor on the next line.
- Though gets( ) is capable of receiving only one string at a time, the plus point with gets( ) is that it can receive a multi-word string.

Pointers and Strings

- Suppose we wish to store “Hello”.
- We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a char pointer.
- This is shown below:
- char str[ ] = "Hello" ;
- char *p = "Hello" ;
- There is a subtle difference in usage of these two forms.
- For example, we cannot assign a string to another, whereas, we can assign a char pointer to another char pointer

```
main( )
{
        char  str1[ ] = "Hello" ;
        char  str2[10] ;

        char  *s = "Good Morning" ;
        char  *q ;

       str2 = str1 ; /* error */
       q = s ; /* works */
}
```

Pointers and Strings

- Also, once a string has been defined it cannot be initialized to another set of characters.
- Unlike strings, such an operation is perfectly valid with char pointers.

```
main( )
{
      char  str1[ ] = "Hello" ;
      char  *p = "Hello" ;
      str1 = "Bye" ; /* error */
      p = "Bye" ; /* works */
}
```

Standard Library String Functions

- With every C compiler a large set of useful string handling library functions are provided.
- Figure lists the more commonly used functions along with their purpose.

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another

strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

strlen( )

- This function counts the number of characters present in a string.

```
main( )
{
        char  arr[ ] = "Bamboozled" ;
        int  len1, len2 ;

        len1 = strlen ( arr ) ;
        len2 = strlen ( "Humpty Dumpty" ) ;

        printf ( "\nstring = %s length = %d", arr, len1 ) ;
        printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

The output would be...

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

- Note that in the first call to the function strlen( ), we are passing the base address of the string, and the function in turn returns the length of the string.
- While calculating the length it doesn’t count ‘\0’.
- Even in the second call,
- len2 = strlen ("Humpty Dumpty") ;
- what gets passed to strlen( ) is the address of the string and not the string itself.
- Can we not write a function xstrlen( ) which imitates the standard library function strlen( )

```
/* A look-alike of the function strlen( ) */
main( )
{
      char  arr[ ] = "Bamboozled" ;
      int  len1, len2 ;

      len1 = xstrlen ( arr ) ;
      len2 = xstrlen ( "Humpty Dumpty" ) ;

      printf ( "\nstring = %s length = %d", arr, len1 ) ;
      printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

```
xstrlen ( char  *s )
{
      int  length = 0 ;

      while ( *s != '\0' )
      {
            length++ ;
            s++ ;
      }

      return ( length ) ;
}
```

The output would be...

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

strcpy( )

- This function copies the contents of one string into another.
- The base addresses of the source and target strings should be supplied to this function

```
main( )
{
      char  source[ ] = "Sayonara" ;

       char  target[20] ;

       strcpy ( target, source ) ;
       printf ( "\nsource string = %s", source ) ;
       printf ( "\ntarget string = %s", target ) ;
 }
```

And here is the output...

```
source string = Sayonara
target string = Sayonara
```

- On supplying the base addresses, strcpy( ) goes on copying the characters in source string into the target string till it doesn't encounter the end of source string (‘\0’).
- It is our responsibility to see to it that the target string’s dimension is big enough to hold the string being copied into it.
- Thus, a string gets copied into another, piece-meal, character by character.
- There is no short cut for this.
- Let us now attempt to mimic strcpy( ), via our own string copy function, which we will call xstrcpy( ).

```
main( )
{
      char  source[ ] = "Sayonara" ;
      char  target[20] ;

      xstrcpy ( target, source ) ;
      printf ( "\nsource string = %s", source ) ;
      printf ( "\ntarget string = %s", target ) ;
}
```

```
xstrcpy ( char  *t, char  *s )
{
      while ( *s != '\0' )
      {
            *t = *s ;
            s++ ;

            t++ ;
      }
      *t = '\0' ;
}
```

The output of the program would be...

```
source string = Sayonara
target string = Sayonara
```

- If you look at the prototype of strcpy( ) standard library function, it looks like this…
- strcpy (char *t, const char *s) ;
- By declaring char *s as const we are declaring that the source string should remain constant (should not change).
- Thus the const qualifier ensures that your program does not inadvertently alter a variable that you intended to be a constant.
- It also reminds anybody reading the program listing that the variable is not intended to change.

- We can use const in several situations.
- The following code fragment would help you to fix your ideas about const further.

```
char *p = "Hello" ;  /* pointer is variable, so is string */
*p = 'M' ;  /* works */
p = "Bye" ;  /* works */

const char *q = "Hello" ;  /* string is fixed pointer is not */
*q = 'M' ;  /* error */
q = "Bye" ;  /* works */

char const *s = "Hello" ;  /* string is fixed pointer is not */
*s = 'M' ;  /* error */
s = "Bye" ;  /* works */
```

```
char * const t = "Hello" ;  /* pointer is fixed string is not */
*t = 'M' ;  /* works */
t = "Bye" ;  /* error */

const char * const u = "Hello" ;  /* string is fixed so is pointer */
*u = 'M' ;  /* error */
u = "Bye" ;  /* error */
```

- The keyword const can be used in context of ordinary variables like int, float, etc.
- The following program shows how this can be done.

```
main( )
{
      float r, a ;
      const float pi = 3.14 ;

      printf ( "\nEnter radius of circle " ) ;
      scanf  ( "%f", &r ) ;
      a = pi * r * r ;
      printf ( "\nArea of circle = %f", a ) ;
}
```

strcat( )

- This function concatenates the source string at the end of the target string.
- For example, “Bombay” and “Nagpur” on concatenation would result into a string “BombayNagpur”.
- Here is an example of strcat( ) at work.
- Note that the target string has been made big enough to hold the final string.

```
main( )
{
        char  source[ ] = "Folks!" ;
        char  target[30] = "Hello" ;

        strcat ( target, source ) ;
        printf ( "\nsource string = %s", source ) ;
        printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

```
source string = Folks!
target string = HelloFolks!
```

Exercise

- develop your own xstrcat( ) on lines of xstrlen( ) and xstrcpy( ).

strcmp( )

- This is a function which compares two strings to find out whether they are same or different.
- The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first.
- If the two strings are identical, strcmp( ) returns a value zero.
- If they’re not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters.

```
main( )
{
        char  string1[ ] = "Jerry" ;
        char  string2[ ] = "Ferry" ;
        int  i, j, k ;

        i = strcmp ( string1, "Jerry" ) ;
        j = strcmp ( string1, string2 ) ;
        k = strcmp ( string1, "Jerry boy" ) ;

        printf ( "\n%d %d %d", i, j, k ) ;
}
```

And here is the output...

```
0 4 -32
```

- In the first call to strcmp( ), the two strings are identical—“Jerry” and “Jerry”—and the value returned by strcmp( ) is zero.
- In the second call, the first character of “Jerry” doesn't match with the first character of “Ferry” and the result is 4, which is the numeric difference between ASCII value of ‘J’ and ASCII value of ‘F’.
- In the third call to strcmp( ) “Jerry” doesn’t match with “Jerry boy”, because the null character at the end of “Jerry” doesn’t match the blank in “Jerry boy”.
- The value returned is -32, which is the value of null character minus the ASCII value of space, i.e., ‘\0’ minus ‘ ’, which is equal to -32.

- The exact value of mismatch will rarely concern us.
- All we usually want to know is whether or not the first string is alphabetically before the second string.
- If it is, a negative value is returned; if it isn’t, a positive value is returned.
- Any non-zero value means there is a mismatch.
- Try to implement this procedure into a function xstrcmp( ).

Two-Dimensional Array of Characters

- In the last chapter we saw several examples of 2-dimensional integer arrays.
- Let’s now look at a similar entity, but one dealing with characters.
- Our example program asks you to type your name.
- When you do so, it checks your name against a master list to see if you are worthy of entry to the palace.

```
#define FOUND 1
#define NOTFOUND 0
main( )
{
      char  masterlist[6][10] = {
                                    "akshay",
                                    "parag",
                                    "raman",
                                    "srinivas",
                                    "gopal",
                                    "rajesh"
                                  } ;
      int  i, flag, a ;
      char  yourname[10] ;
```

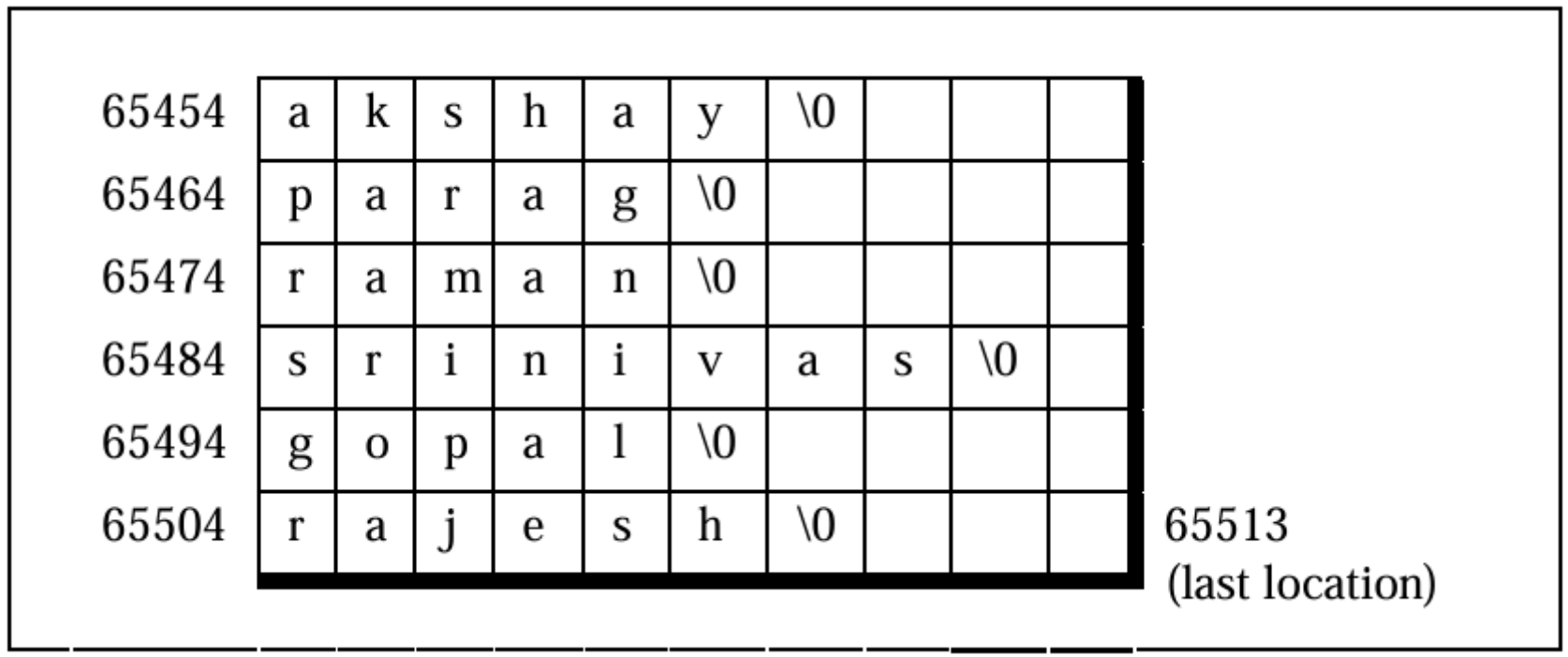
```
    printf ( "\nEnter your name " ) ;
    scanf ( "%s", yourname ) ;

    flag = NOTFOUND ;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        a = strcmp ( &masterlist[i][0], yourname ) ;
        if ( a == 0 )
        {
            printf ( "Welcome, you can enter the palace" ) ;
            flag = FOUND ;
            break ;
        }
    }

    if ( flag == NOTFOUND )
        printf ( "Sorry, you are a trespasser" ) ;
}
```

- And here is the output for two sample runs of this program...
- Enter your name dinesh
- Sorry, you are a trespasser
- Enter your name raman
- Welcome, you can enter the palace

- Notice how the two-dimensional character array has been initialized.
- The order of the subscripts in the array declaration is important.
- The first subscript gives the number of names in the array, while the second subscript gives the length of each item in the array.
- Instead of initializing names, had these names been supplied from the keyboard, the program segment would have looked like this...

```
for ( i = 0 ; i <= 5 ; i++ )
      scanf ( "%s", &masterlist[i][0] ) ;
```

- The names would be stored in the memory as shown in Figure
- Note that each string ends with a ‘\0’.
- The arrangement as you can appreciate is similar to that of a two-dimensional numeric array.
- As seen from the above pattern some of the names do not occupy all the bytes reserved for them.
- For example, even though 10 bytes are reserved for storing the name “akshay”, it occupies only 7 bytes.
- Thus, 3 bytes go waste. Similarly, for each name there is some amount of wastage.
- In fact, more the number of names, more would be the wastage.
- Can this not be avoided? Yes, it can be... by using what is called an ‘array of pointers’


65454
a k s h a y \0
65464
p a r a g \0
65474
r a m a n \0
65484
s r i n i v a s \0
65494
g o p a l \0
65504
r a j e s h \0
65513
(last location)

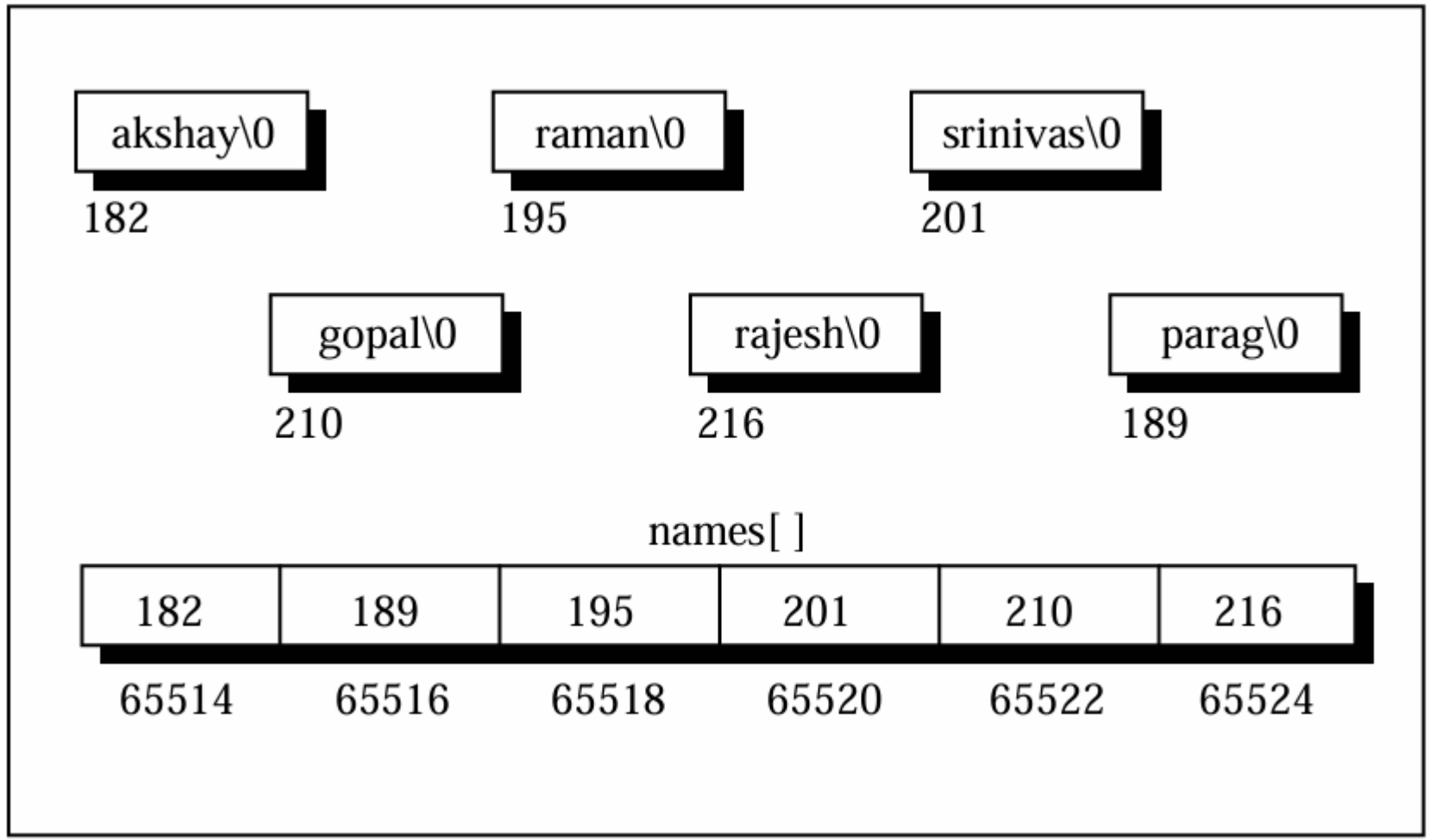
Array of Pointers to Strings

- As we know, a pointer variable always contains an address.
- Therefore, if we construct an array of pointers it would contain a number of addresses.

```
char  *names[ ] = {
                    "akshay",
                    "parag",
                    "raman",
                    "srinivas",
                    "gopal",
                    "rajesh"
                  } ;
```

Array of Pointers to Strings

- In this declaration names[ ] is an array of pointers.
- It contains base addresses of respective names.
- That is, base address of “akshay” is stored in names[0], base address of “parag” is stored in names[1] and so on.
- This is depicted in Figure


akshay\0
182
raman\0
195
srinivas\0
201
gopal\0
210
rajesh\0
216
parag\0
189
names[ ]
182
189
195
201
210
216
65514
65516
65518
65520
65522
65524

Saving of Memory Space

- In the two-dimensional array of characters, the strings occupied 60 bytes.
- As against this, in array of pointers, the strings occupy only 41 bytes—a net saving of 19 bytes.
- A substantial saving, you would agree.
- But realize that actually 19 bytes are not saved, since 12 bytes are sacrificed for storing the addresses in the array names[ ].
- Thus, one reason to store strings in an array of pointers is to make a more efficient use of available memory.
- Another reason to use an array of pointers to store strings is to obtain greater ease in manipulation of the strings.

Ease of Manipulating Strings

- This is shown by the following programs.
- The first one uses a two-dimensional array of characters to store the names, whereas the second uses an array of pointers to strings.
- The purpose of both the programs is very simple.
- We want to exchange the position of the names “raman” and “srinivas”.

```
/* Exchange names using 2-D array of characters */
main( )
{
      char  names[ ][10] = {
                                    "akshay",
                                    "parag",
                                    "raman",
                                    "srinivas",
                                    "gopal",
                                    "rajesh"
                                  } ;
            int  i ;
            char  t ;

            printf ( "\nOriginal: %s %s", &names[2][0], &names[3][0] ) ;

            for ( i = 0 ; i <= 9 ; i++ )
            {
                  t = names[2][i] ;
                  names[2][i] = names[3][i] ;
                  names[3][i] = t ;
            }

            printf ( "\nNew: %s %s", &names[2][0], &names[3][0] ) ;
      }
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

```
main( )
{
      char  *names[ ] = {
                              "akshay",
                              "parag",
                              "raman",
                              "srinivas",
                              "gopal",
                              "rajesh"
                        } ;
      char  *temp ;

      printf ( "Original: %s %s", names[2], names[3] ) ;

      temp = names[2] ;
      names[2] = names[3] ;
      names[3] = temp ;

      printf ( "\nNew: %s %s", names[2], names[3] ) ;
}
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

- The output is same as the earlier program.
- In this program all that we are required to do is exchange the addresses (of the names) stored in the array of pointers, rather than the names themselves.
- Thus, by effecting just one exchange we are able to interchange names.
- This makes handling strings very convenient.
- Thus, from the point of view of efficient memory usage and ease of programming, an array of pointers to strings definitely scores over a two-dimensional character array.
- That is why, even though in principle strings can be stored and handled through a two dimensional array of characters, in actual practice it is the array of pointers to strings, which is more commonly used.

Limitation of Array of Pointers to Strings

- When we are using a two-dimensional array of characters we are at liberty to either initialize the strings where we are declaring the array, or receive the strings using scanf( ) function.
- However, when we are using an array of pointers to strings we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using scanf( ).
- Thus, the following program would never work out.
- The program doesn't work because; when we are declaring the array it is containing garbage values.
- And it would be definitely wrong to send these garbage values to scanf( ) as the addresses where it should keep the strings received from the keyboard.

```
main( )
{
    char  *names[6] ;
    int  i ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\nEnter name " ) ;
        scanf ( "%s", names[i] ) ;
    }
}
```

Solution

```
#include "alloc.h"
main( )
{
      char  *names[6] ;
      char n[50] ;
      int  len, i ;
      char *p ;

      for ( i = 0 ; i <= 5 ; i++ )
      {
            printf ( "\nEnter name " ) ;
            scanf ( "%s", n ) ;
            len = strlen ( n ) ;
            p = malloc ( len + 1 ) ;
            strcpy ( p, n ) ;
            names[i] = p ;
      }

      for ( i = 0 ; i <= 5 ; i++ )
            printf ( "\n%s", names[i] ) ;
}
```

- using malloc( ) we can allocate memory dynamically, during execution.
- The argument that we pass to malloc( ) can be a variable whose value can change during execution.
- Once we have allocated the memory using malloc( ) we have copied the name received through the keyboard into this allocated space and finally stored the address of the allocated chunk in the appropriate element of names[ ], the array of pointers to strings.
- This solution suffers in performance because we need to allocate memory and then do the copying of string for each name received through the keyboard.