Computer Programming & Problem Solving

CS100

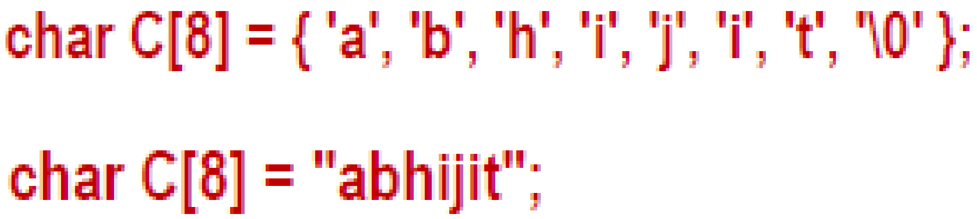
Mrs Sanga G. Chaki
Department of Computer Science and Engineering
National Institute of Technology, Goa

January, 2023

Strings

1. Character arrays
2. To store a group of characters
3. Used to manipulate text such as words and sentences.
4. A string always terminates with a NULL character –
   a) Only way to know where the string ends
5. Collection of characters vs Strings

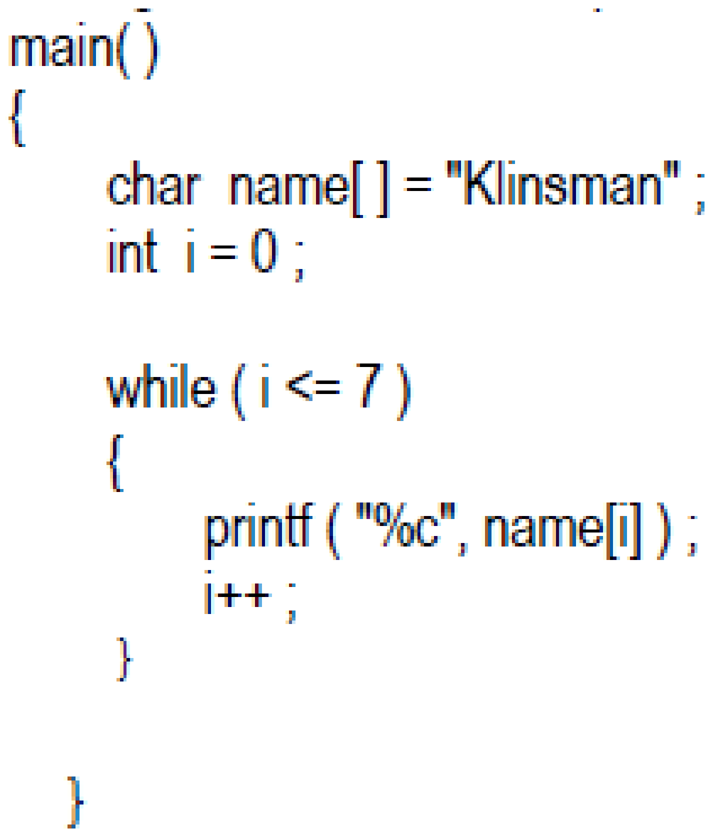
Strings - Initialization



```c
char C[8] = { 'a', 'b', 'h', 'i', 'j', 'i', 't', '\0' };

char C[8] = "abhijit";
```

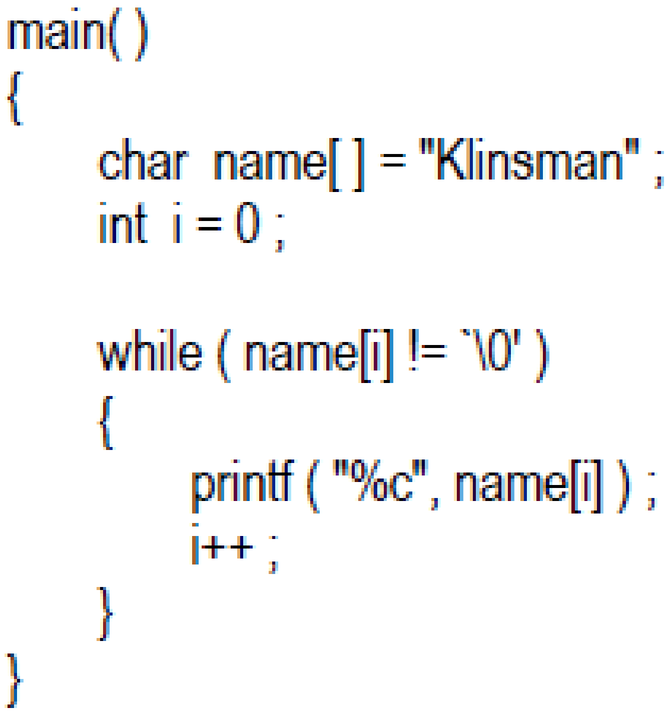
1. C[0] gets the value 'a', C[1] the value 'b', and so on.
2. The trailing null character is missing in the second method.
   a) C automatically puts it at the end if you define it like this
3. Note
   a) for individual characters, C uses single quotes,
   b) for strings, it uses double quotes

Access string elements – Method 1



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

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

 }
```

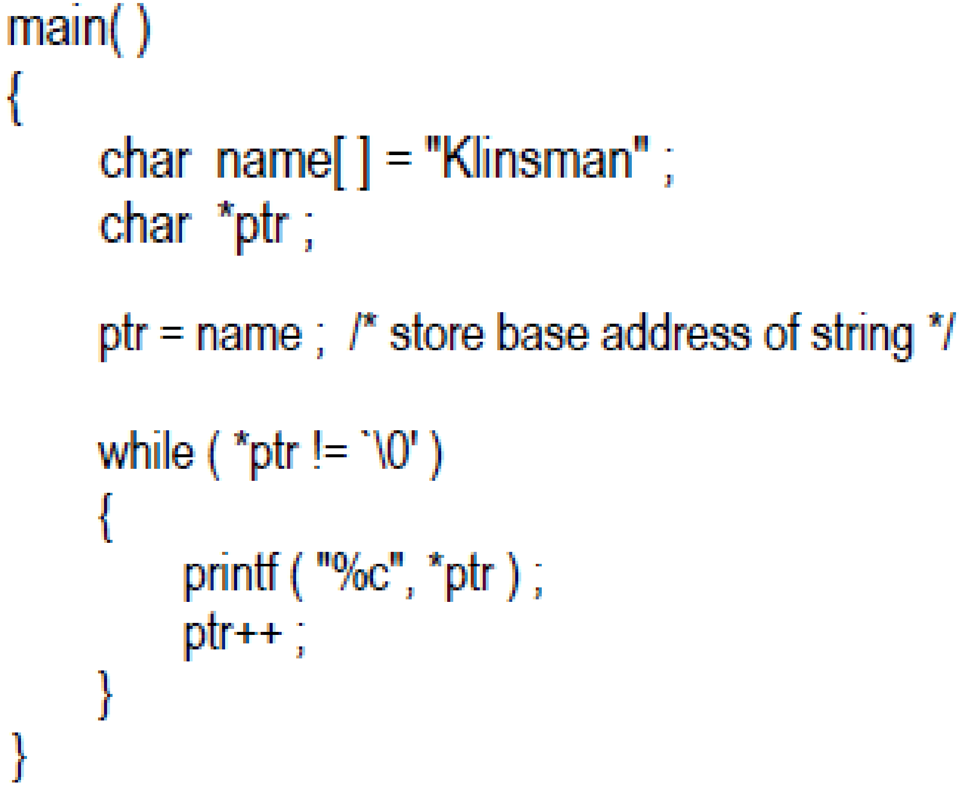
Access string elements – Method 2



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

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

1. **Works even if we do not know the length of the string.**

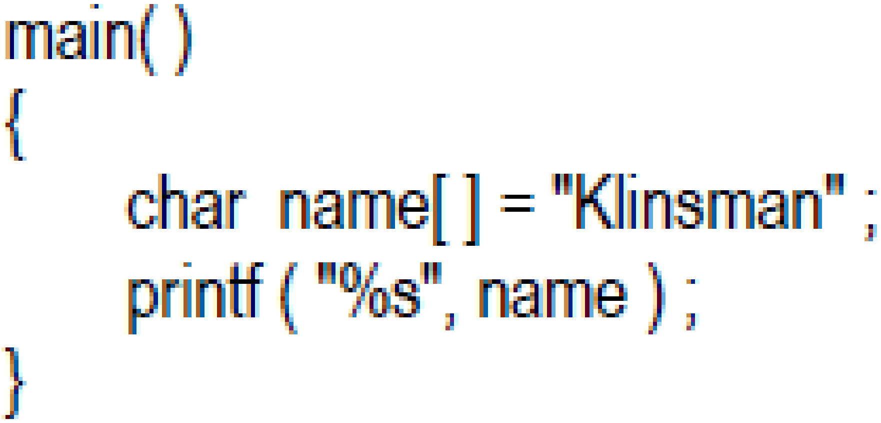
Access string elements – Method 3



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

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

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

1. Using Pointers

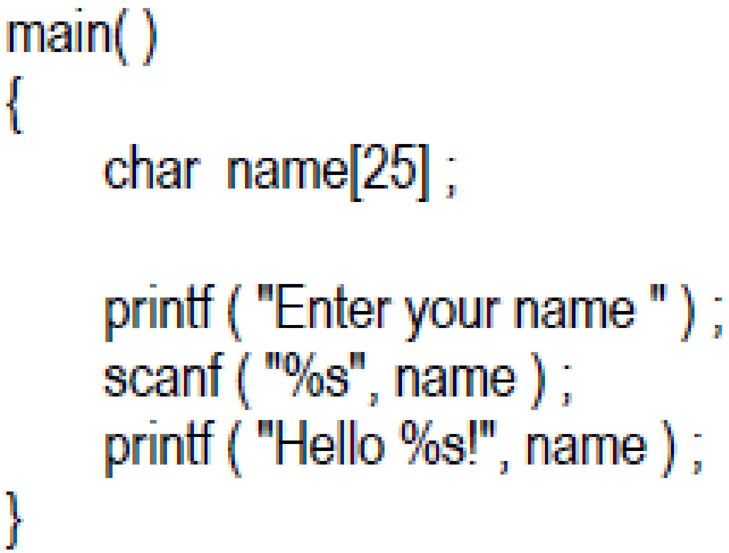
Access string elements – Method 4



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

1. **Using printf()**
2. **%s is the format specifier for the string**

Receive a string from user



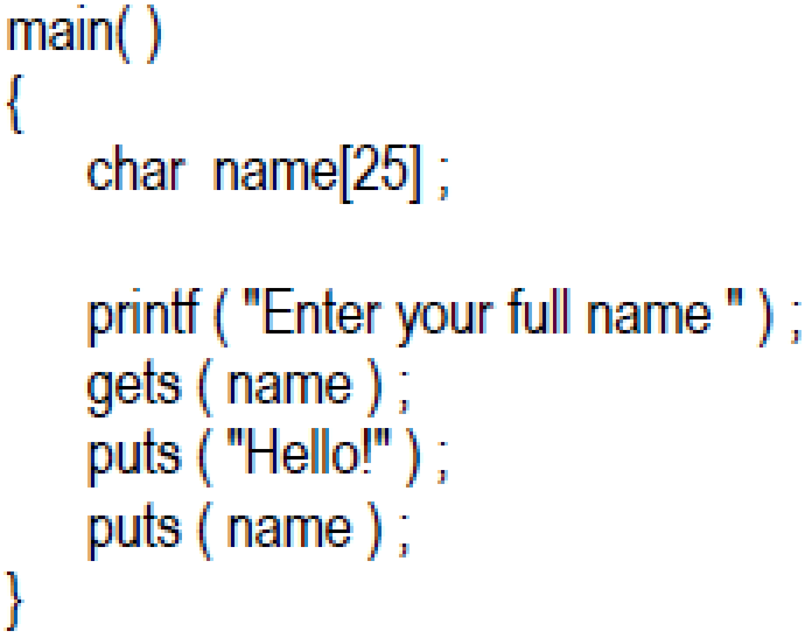
```
main( )
{
        char  name[25] ;

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

1. **Using scanf()**
2. **%s is the format specifier for the string**
3. **Note: No & in scanf(). Why?**
   a) **we are passing the base address of the array to the scanf( )**
   b) **scanf( ) fills in the characters from keyboard until the enter key is hit.**

Some Limitations

1. The length of the string should not exceed the dimension of the character array. – Why?
   a) Something might be overwritten
   b) C compiler doesn’t perform bounds checking
2. scanf( ) is not capable of receiving multi-word strings. So what do we use?
   a) gets()
   b) puts()

gets() and puts()



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

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

1. Why two puts()? – It can display only one string at a time.
2. gets() can receive multi-word strings, though it can receive only 1 string at a time.

Some Differences

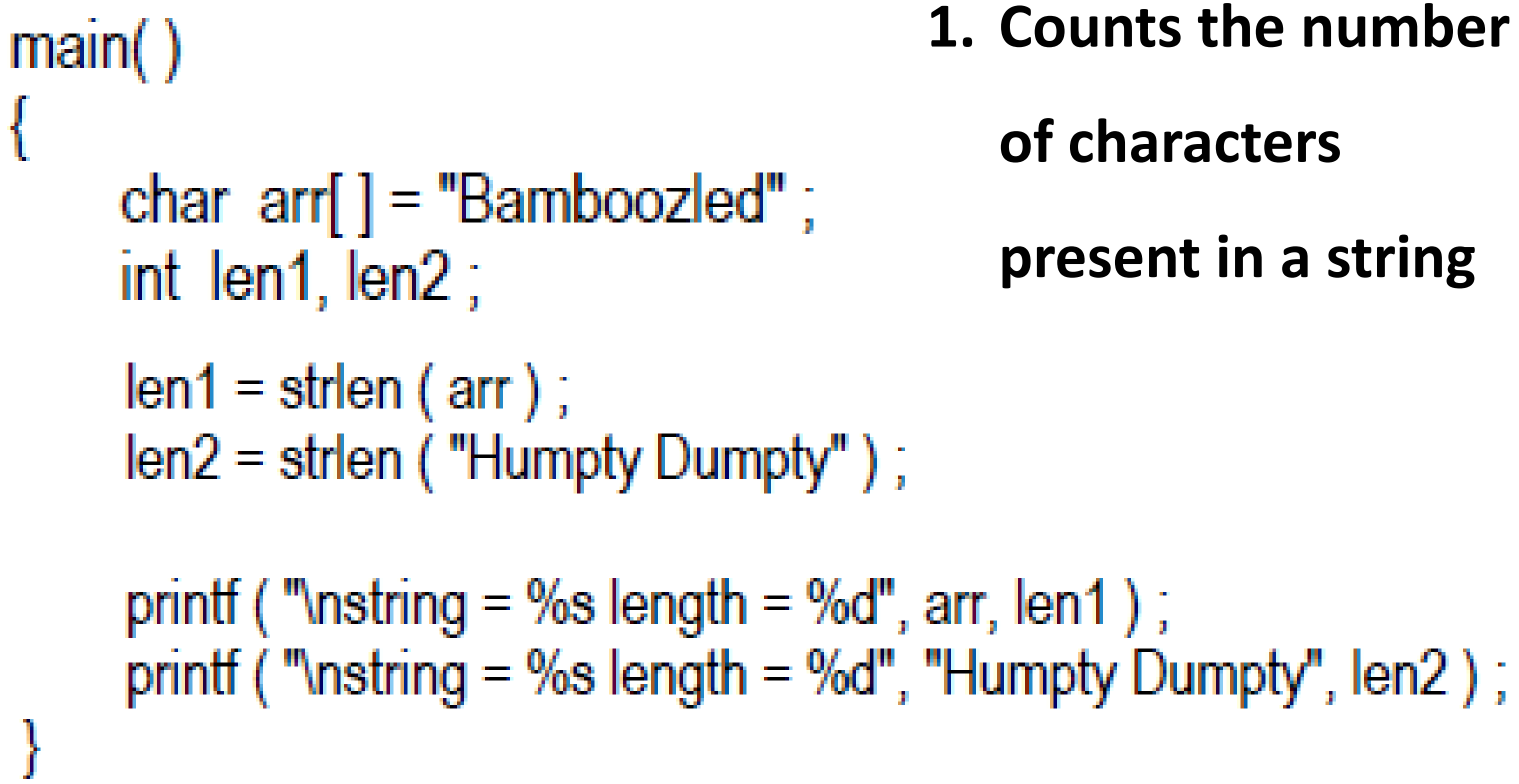
1. “a” versus ‘a’
   a) ‘a’ is a single character value (stored in 1 byte) as the ASCII value for the letter, a
   b) “a” is an array with two characters, the first is a, the second is the character value \0

Standard Library Functions

1. There exists a set of C library functions for character string manipulation.
   1. strcpy :: string copy
   2. strlen :: string length
   3. strcmp :: string comparison
   4. strtcat :: string concatenation
2. It is required to add the line
   1. #include <string.h>

strlen( )

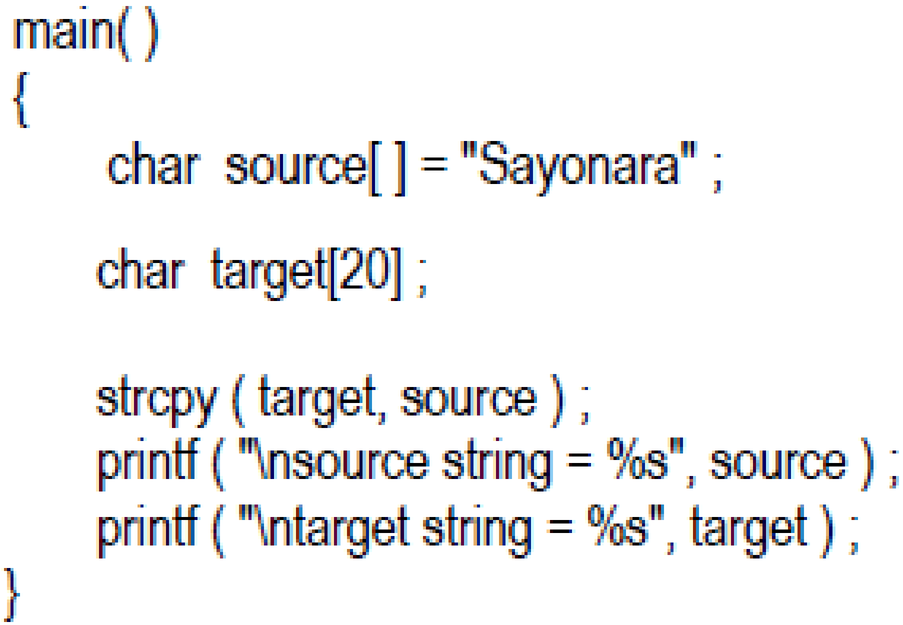
1. Counts the number of characters present in a string



```
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 ) ;
}
```

1. What are the outputs?
2. Note – It doesn’t count the Null character.

strcpy( )

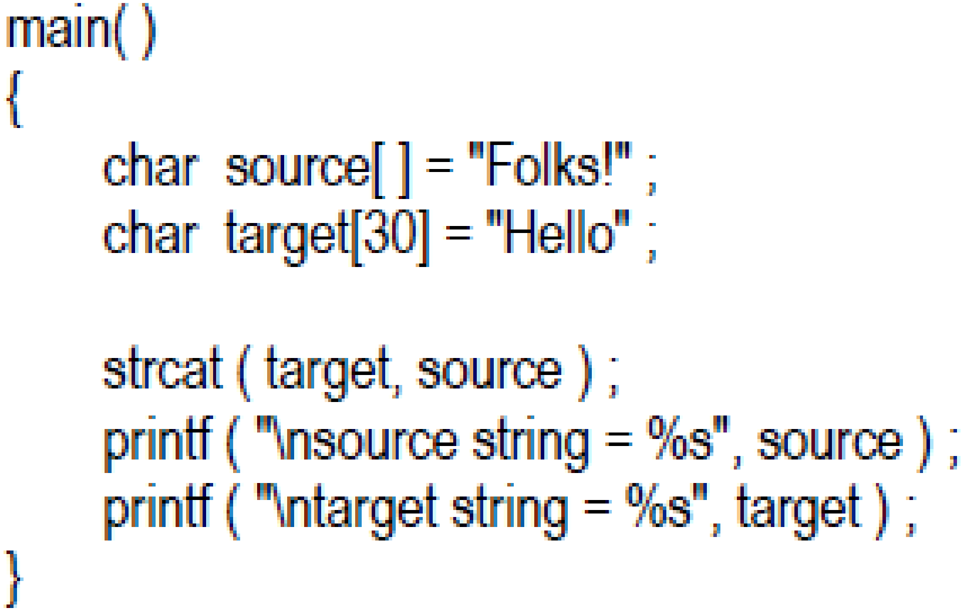


```
main( )
{
     char  source[ ] = "Sayonara" ;
     char  target[20] ;

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

1. Copies the contents of one string into another

1. What are the outputs?

strcat( )

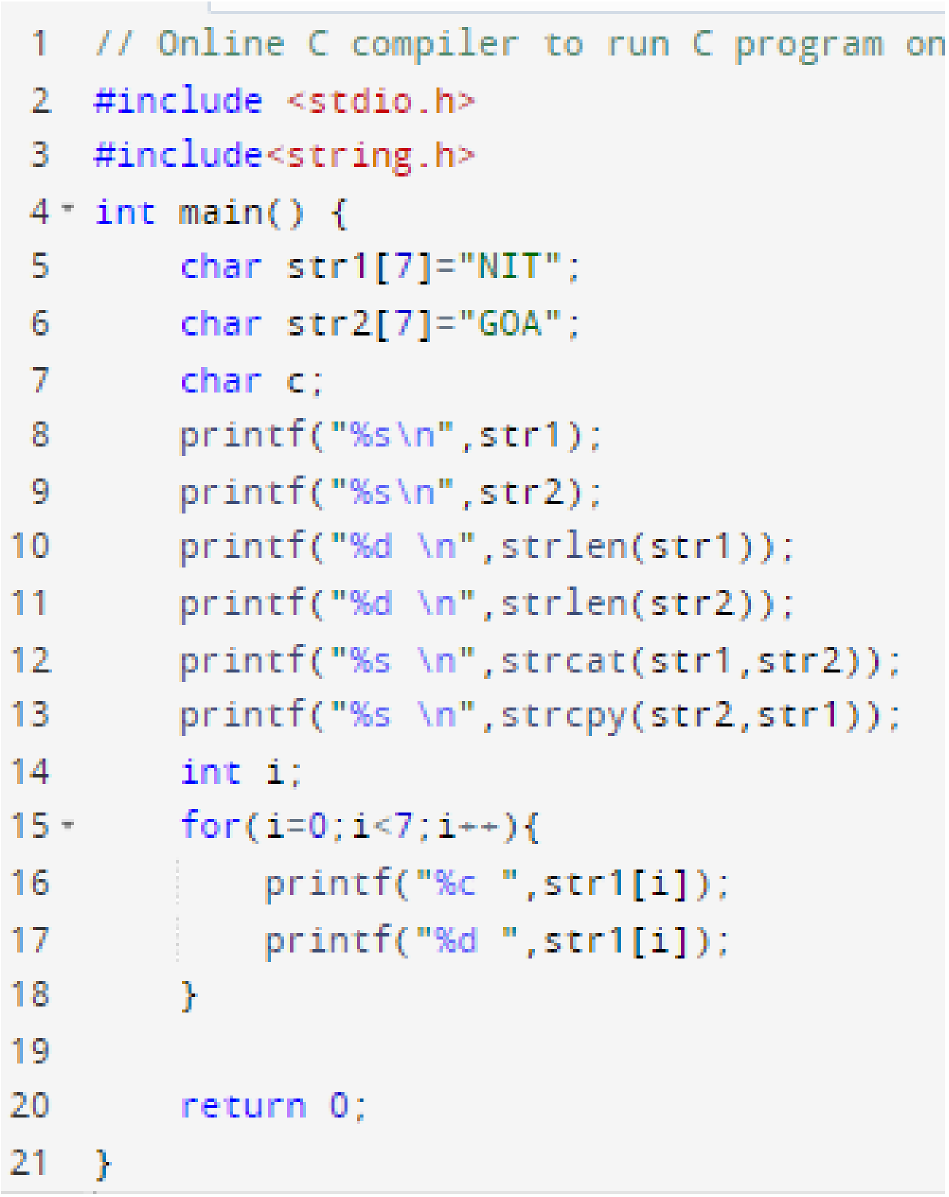


```
main( )
{
    char  source[ ] = "Folks!" ;
    char  target[30] = "Hello" ;

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

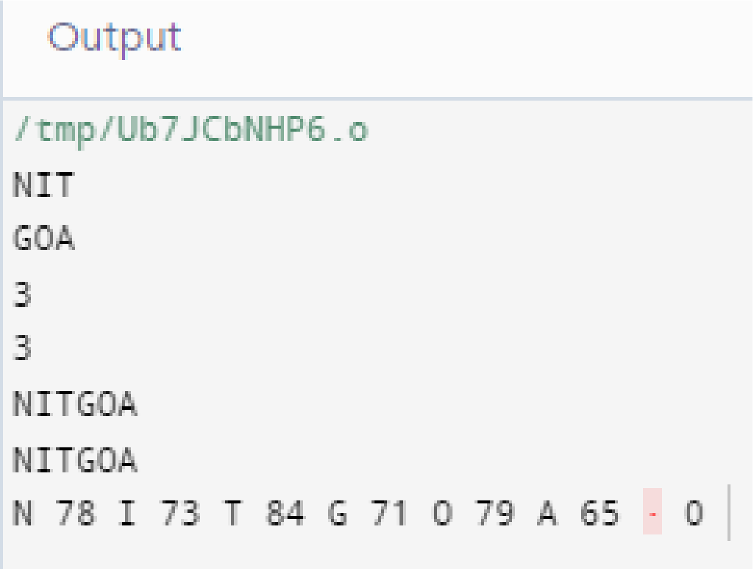
1. Concatenates the source string at the end of the target string

1. What are the outputs?

Example



```c
// Online C compiler to run C program on
#include <stdio.h>
#include<string.h>
int main() {
    char str1[7]="NIT";
    char str2[7]="GOA";
    char c;
    printf("%s\n",str1);
    printf("%s\n",str2);
    printf("%d \n",strlen(str1));
    printf("%d \n",strlen(str2));
    printf("%s \n",strcat(str1,str2));
    printf("%s \n",strcpy(str2,str1));
    int i;
    for(i=0;i<7;i++){
        printf("%c ",str1[i]);
        printf("%d ",str1[i]);
    }

    return 0;
}
```



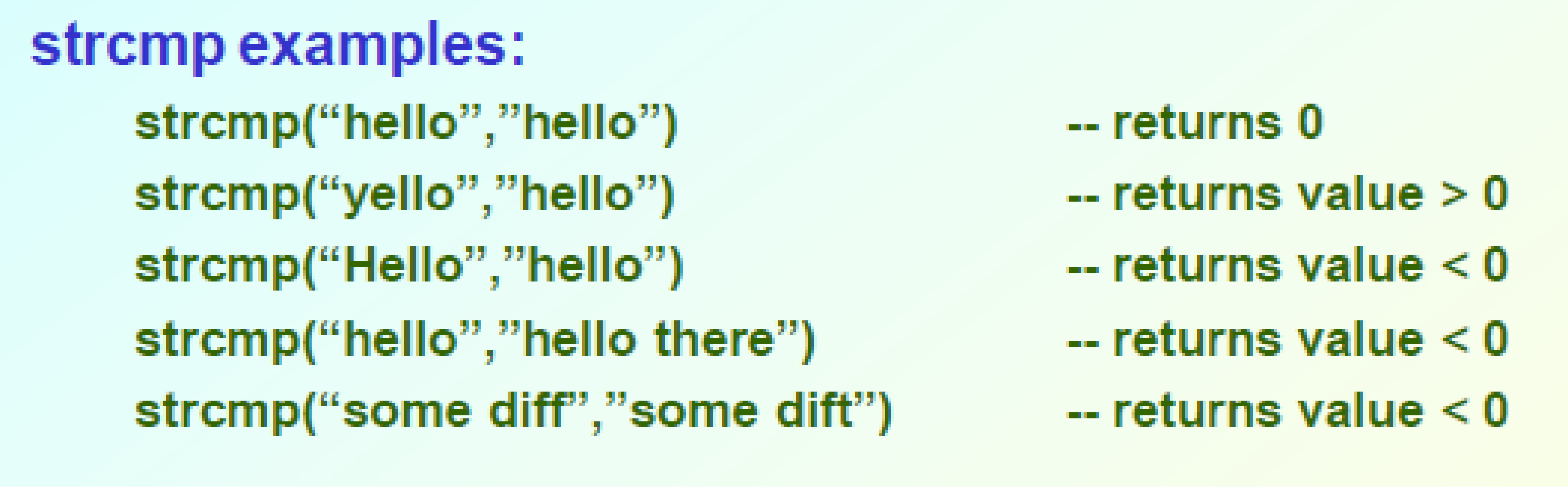
Output

```
/tmp/Ub7JCbNHP6.o
NIT
GOA
3
3
NITGOA
NITGOA
N 78 I 73 T 84 G 71 O 79 A 65 · 0
```

strcmp( )

1. Compares two strings to find out whether they are same or different.
2. Strings are compared character by character until there is a mismatch or end of one of the strings.
3. If the two strings are identical, strcmp( ) returns a value zero.
4. If they’re not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters.

strcmp( )



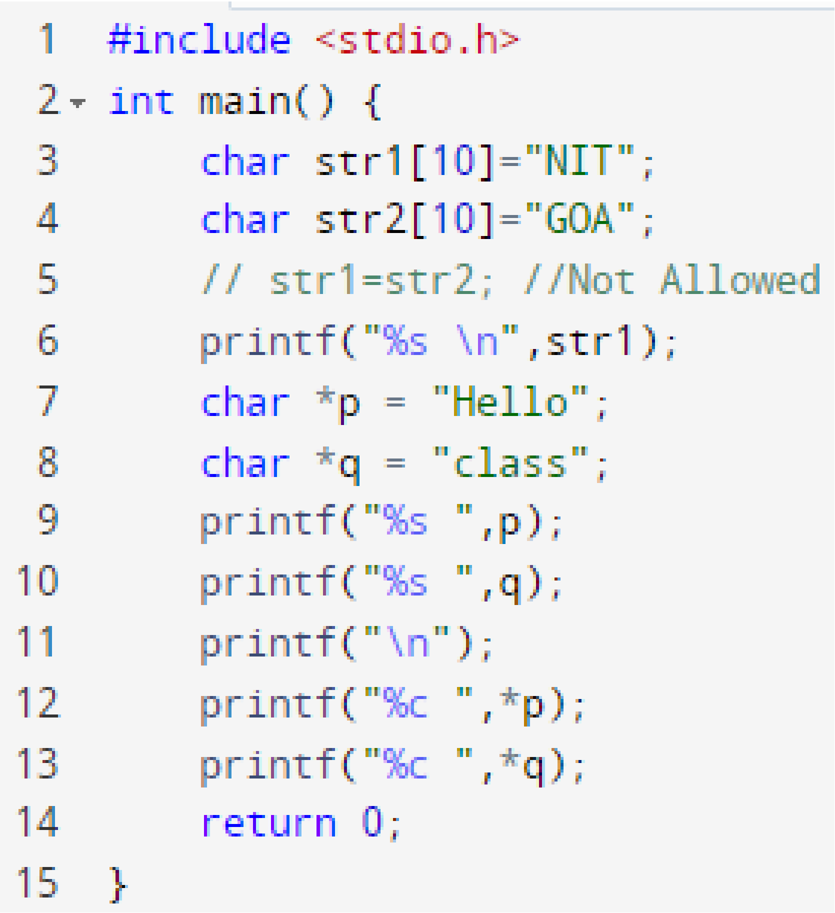
strcmp examples:

strcmp(“hello”,”hello”)	-- returns 0
strcmp(“yello”,”hello”)	-- returns value > 0
strcmp(“Hello”,”hello”)	-- returns value < 0
strcmp(“hello”,”hello there”)	-- returns value < 0
strcmp(“some diff”,”some dift”)	-- returns value < 0

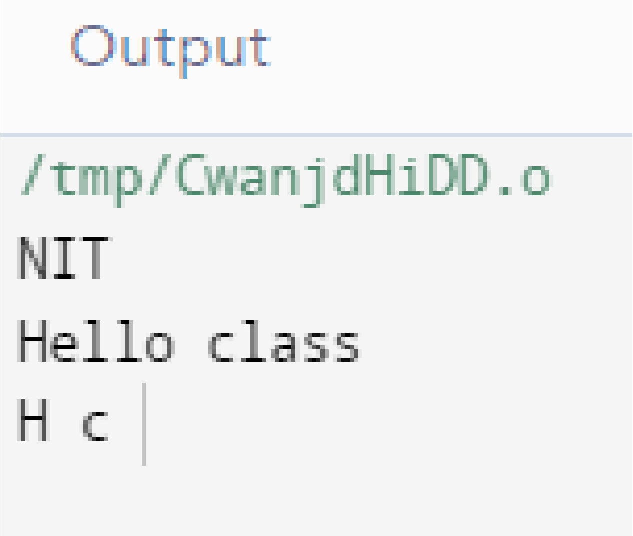
Char Pointers vs Strings – Some Rules

1. char str[ ] = "Hello" ; versus char *p = "Hello" ;
   a) Store Hello in a string
   b) Store Hello at some location in memory and assign the address of the string in a char pointer
2. We cannot assign a string to another, but
3. We can assign a char pointer to another char pointer

Char Pointers vs Strings – Some Rules



```c
#include <stdio.h>
int main() {
    char str1[10]="NIT";
    char str2[10]="GOA";
    // str1=str2; //Not Allowed
    printf("%s \n",str1);
    char *p = "Hello";
    char *q = "class";
    printf("%s ",p);
    printf("%s ",q);
    printf("\n");
    printf("%c ",*p);
    printf("%c ",*q);
    return 0;
}
```

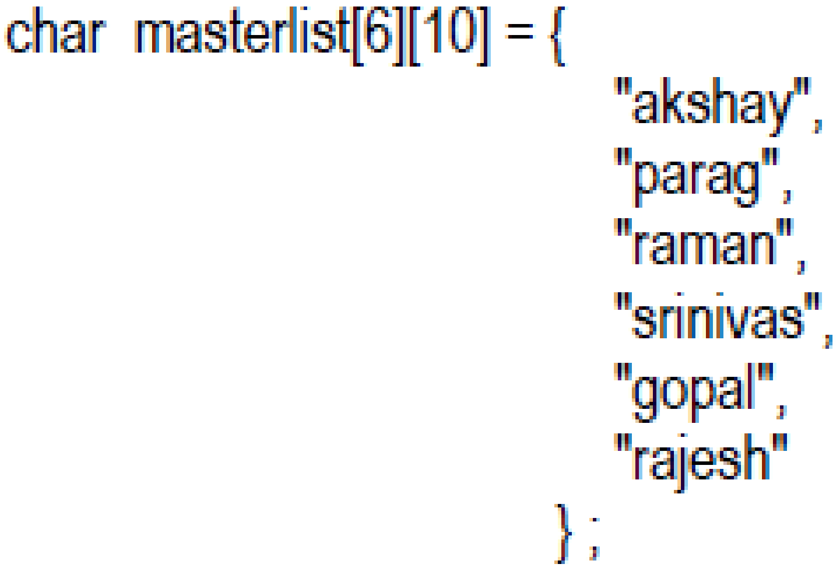


```
/tmp/CwanjdHiDD.o
NIT
Hello class
H c
```

2D Array of Characters

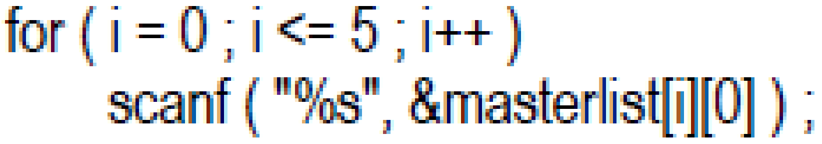
1. Similar concept as 2D array of ints/floats.
2. Order of the subscripts in the array declaration:
   a) first subscript gives the number of strings
   b) second subscript gives the length of each string

2D Strings – Declaration/Initialization



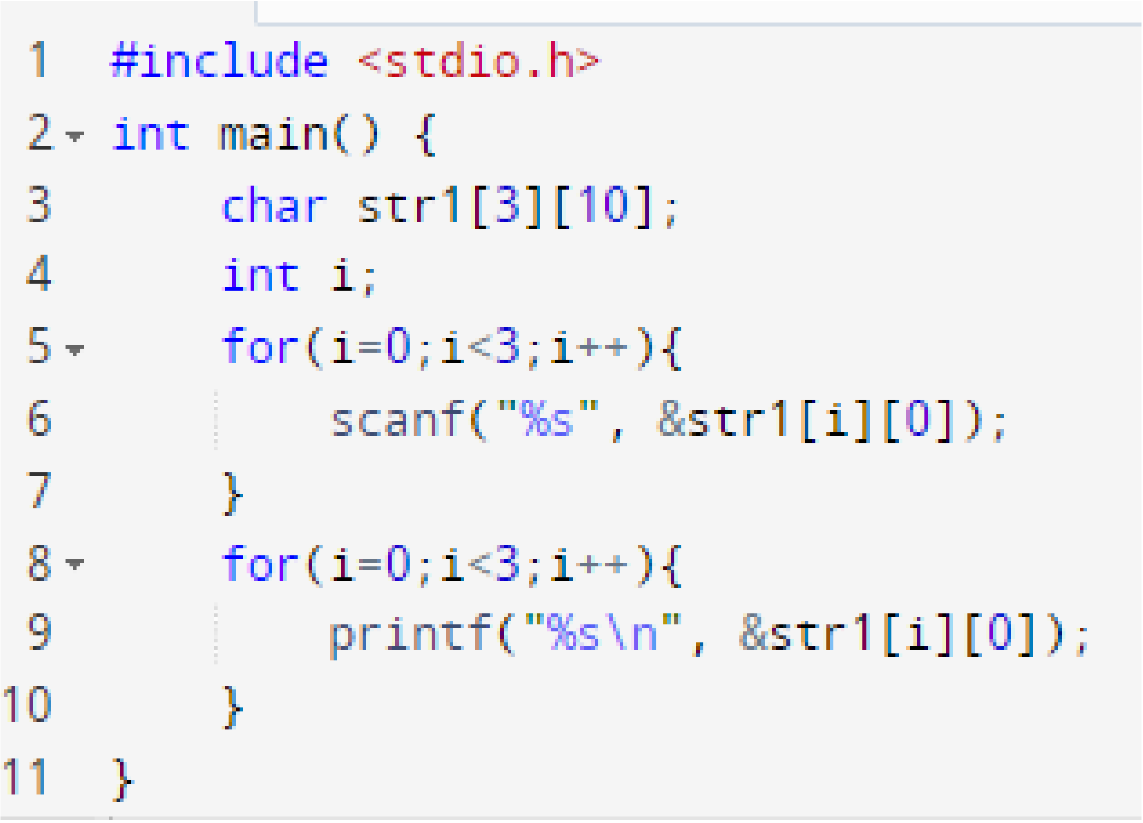
```
char  masterlist[6][10] = {
                        "akshay",
                        "parag",
                        "raman",
                        "srinivas",
                        "gopal",
                        "rajesh"
                      } ;
```

OR



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

2D Strings – Declaration/Initialization



```c
#include <stdio.h>
int main() {
    char str1[3][10];
    int i;
    for(i=0;i<3;i++){
        scanf("%s", &str1[i][0]);
    }
    for(i=0;i<3;i++){
        printf("%s\n", &str1[i][0]);
    }
}
```

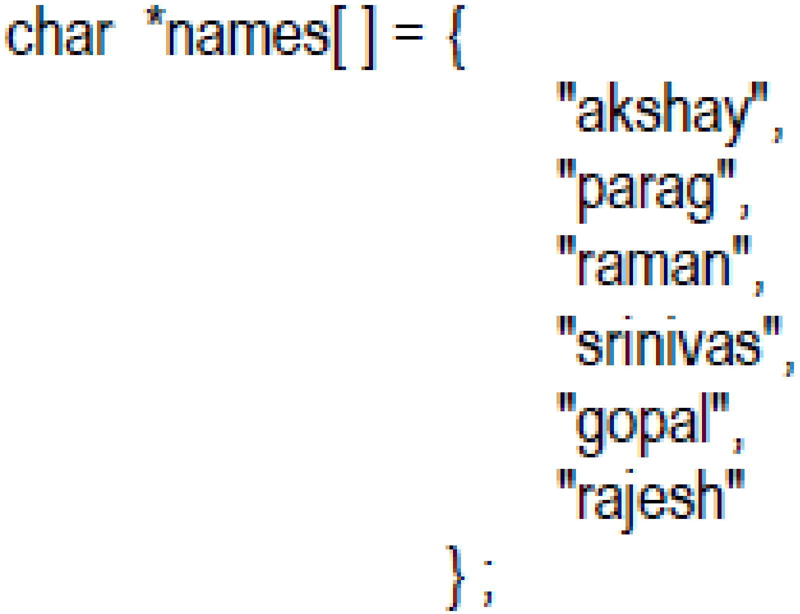
2D Strings – Memory Map

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)

How to avoid this wastage?

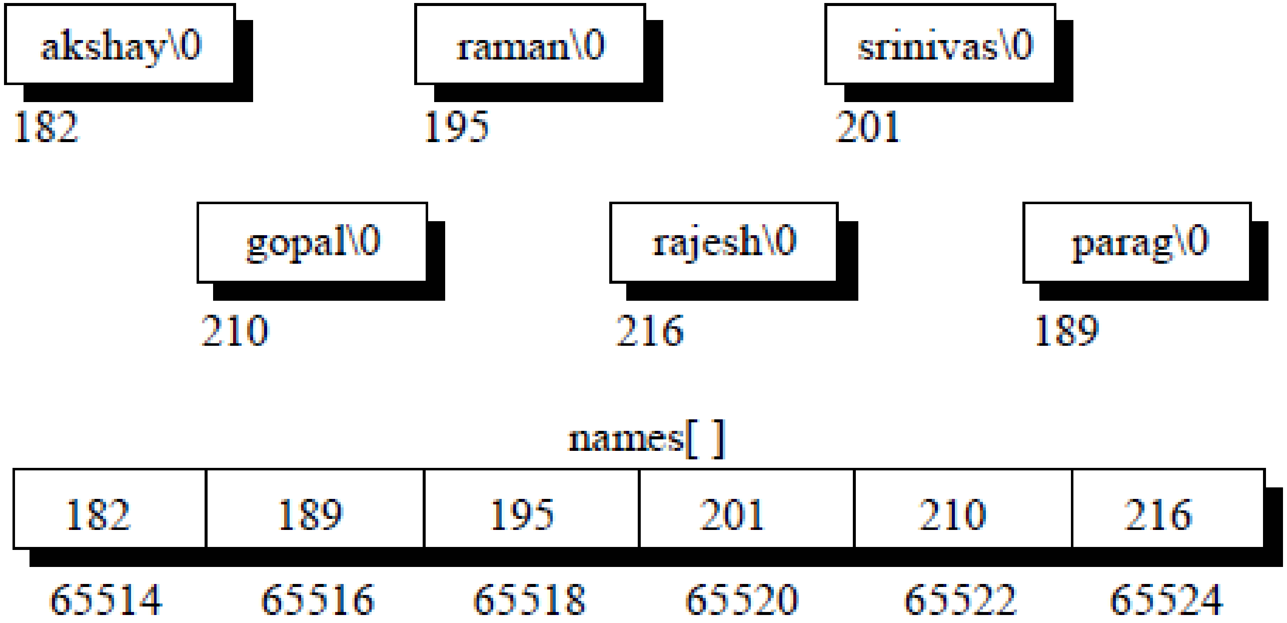
Array of Pointers to Strings



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

1. Array of pointers will contain a number of addresses.
2. Contains base addresses of respective names

Memory Map Now



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

1. How many bytes are needed = 41 + 12 = 53 bytes
2. Storage is saved – wastage avoided.
3. Manipulation of strings is also easier

Limitations

1. When we are using an array of pointers to strings, we cannot receive the strings from keyboard using scanf( ).
2. Why?
   a) When we declare the array it contains garbage values.
   b) Cannot use garbage values as addresses to receive the strings from keyboard
3. What to do?
   a) Either Initialize the strings at the place where the array is declared
   b) Or Use dynamic memory allocation

Dynamic Memory Allocation

1. **When is it used?**
   a) **Amount of data cannot be predicted beforehand.**
   b) **Number of data items keeps changing during program execution**
2. **How is DMA helpful?**
   a) **Memory space required can be specified at the time of execution.**
   b) **C supports allocating and freeing memory dynamically using library routines**

Memory Allocation Functions

1. malloc: Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
2. calloc: Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
3. free: Frees previously allocated space.
4. realloc: Modifies the size of previously allocated space

Allocating a Block of Memory

1. Using the function malloc.

2. Reserves a block of memory of specified size and returns a pointer of type void.

3. The return pointer can be type-casted to any pointer type.

4. General format:

   a) ptr= (type *) malloc(byte_size);

malloc() - examples

1. **cptr= (char *) malloc(20);**

   a) Allocates 20 bytes of space for the pointer cptr of type char

2. **p = (int *) malloc(100 * sizeof(int));**

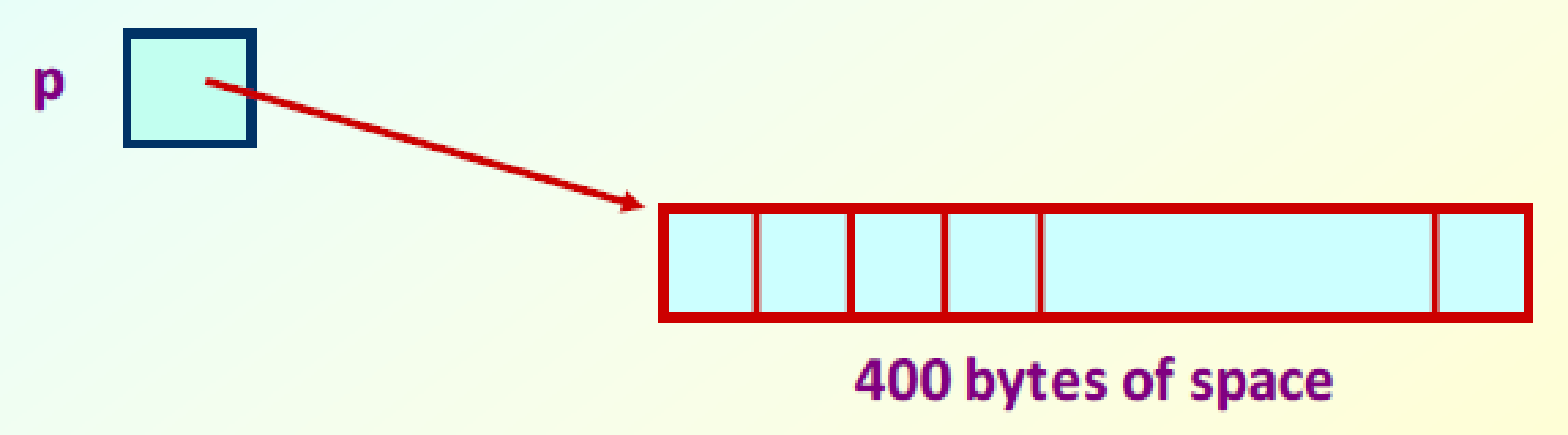
   a) A memory space equivalent to 100 times the size of an int bytes is reserved.

   b) – The address of the first byte of the allocated memory is assigned to the pointer p of type int.

malloc() - examples

1. int *p;

2. p = (int *) malloc(100 * sizeof(int));



malloc() – Points to Note

1. malloc() always allocates a block of contiguous bytes.

2. The allocation can fail if sufficient contiguous memory space is not available.

3. If it fails, malloc returns NULL.

free()

1. When we no longer need the data stored in a block of memory,

   we may release the block for future use.

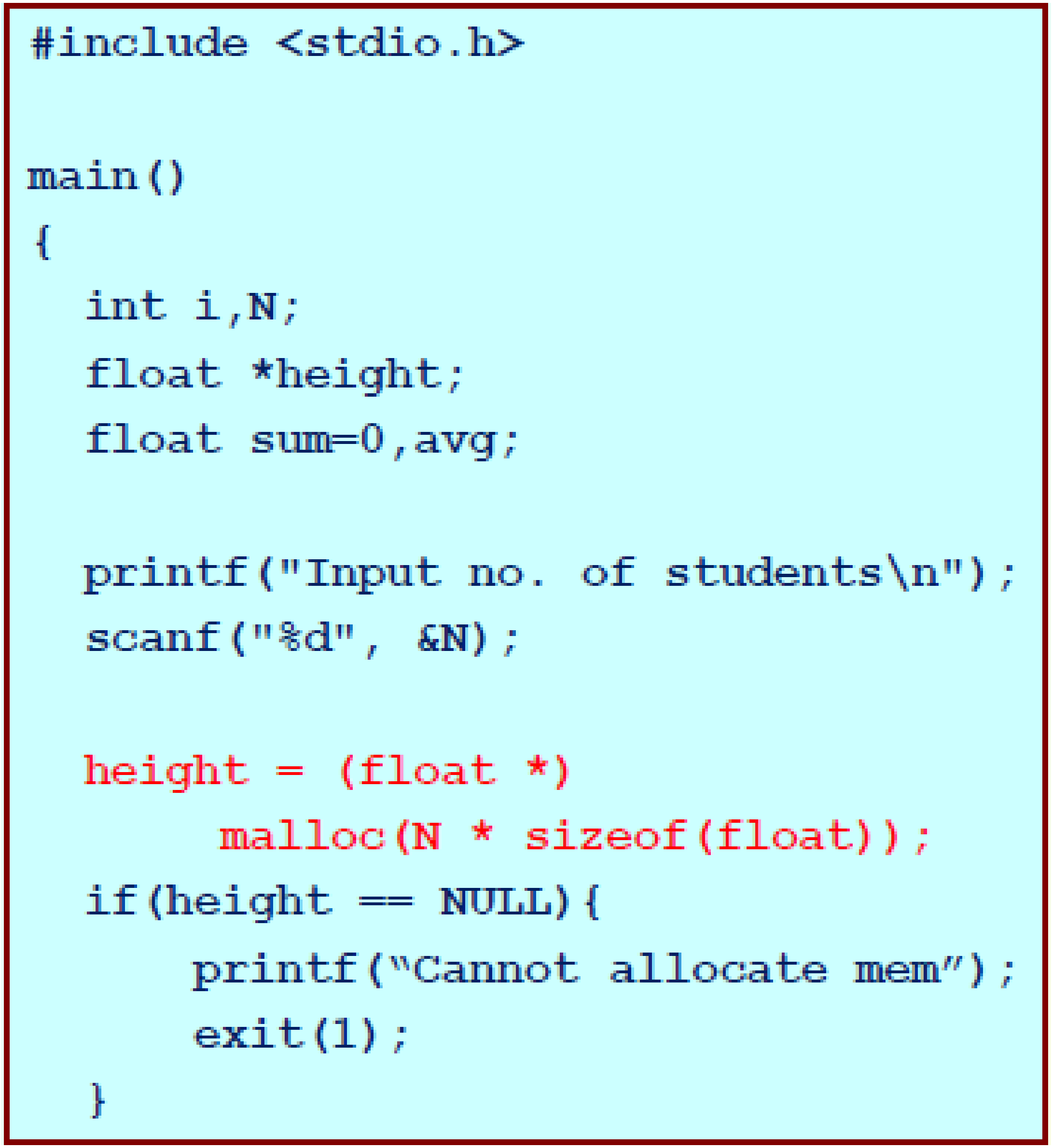
2. How?

```
free (ptr);
```

where ptr is a pointer to a memory block which has been

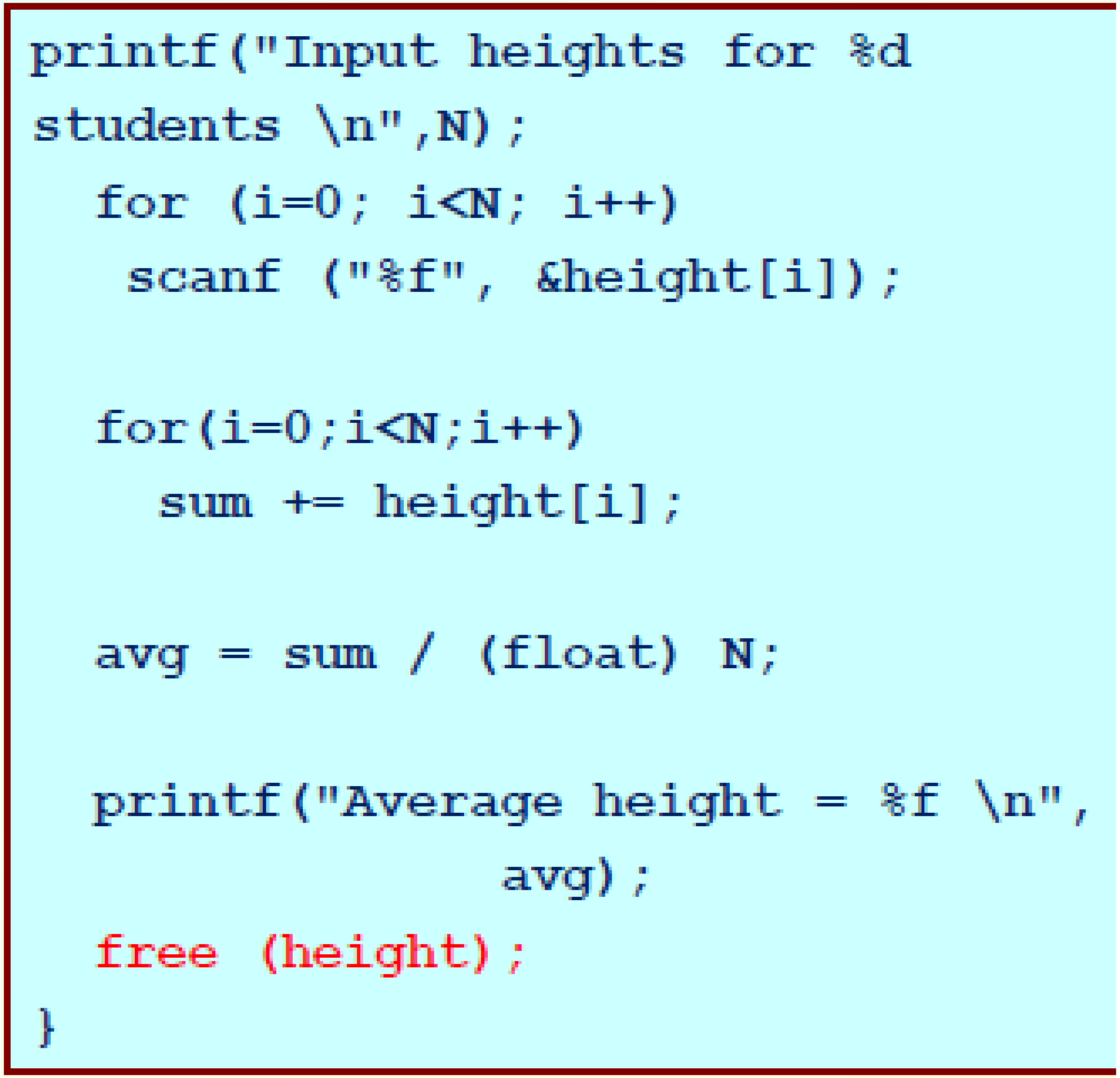
previously created using malloc.

Example1 - 1-D array of floats using DMA



```
#include <stdio.h>

main()
{
  int i,N;
  float *height;
  float sum=0,avg;

  printf("Input no. of students\n");
  scanf("%d", &N);

  height = (float *)
       malloc(N * sizeof(float));
  if(height == NULL){
      printf(“Cannot allocate mem”);
      exit(1);
  }
```

Example1 - 1-D array of floats using DMA



```
printf("Input heights for %d
students \n",N);
   for (i=0; i<N; i++)
    scanf ("%f", &height[i]);

   for(i=0;i<N;i++)
     sum += height[i];

   avg = sum / (float) N;

   printf("Average height = %f \n",
                    avg);
   free (height);
}
```