Strings in C

- How does a function know that how many elements are there in an array
- function(int *array){
- }
- In case integer array – we cannot do it without reserving a particular number say – 999
- In the domain of char array – we have a special character ‘\0’ to indicate the end

Character Array

```
char a[10] = {‘a’, ‘b’, ‘c’, …};

char a[ ] = {‘a’, ‘b’, ‘c’};

char a[10];

a[0] = ‘a’;
a[1] = ‘b’;
a[2] = ‘c’;
a[3] = ‘d’;
a[4] = ‘\0’;
```

abcd – is the string that is being created in this example.

Strings

In C programming, a string is a sequence of characters terminated with a null character \0. For example:

```
char c[] = "c string";
```

Strings

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

Memory diagram of strings in C programming

Strings

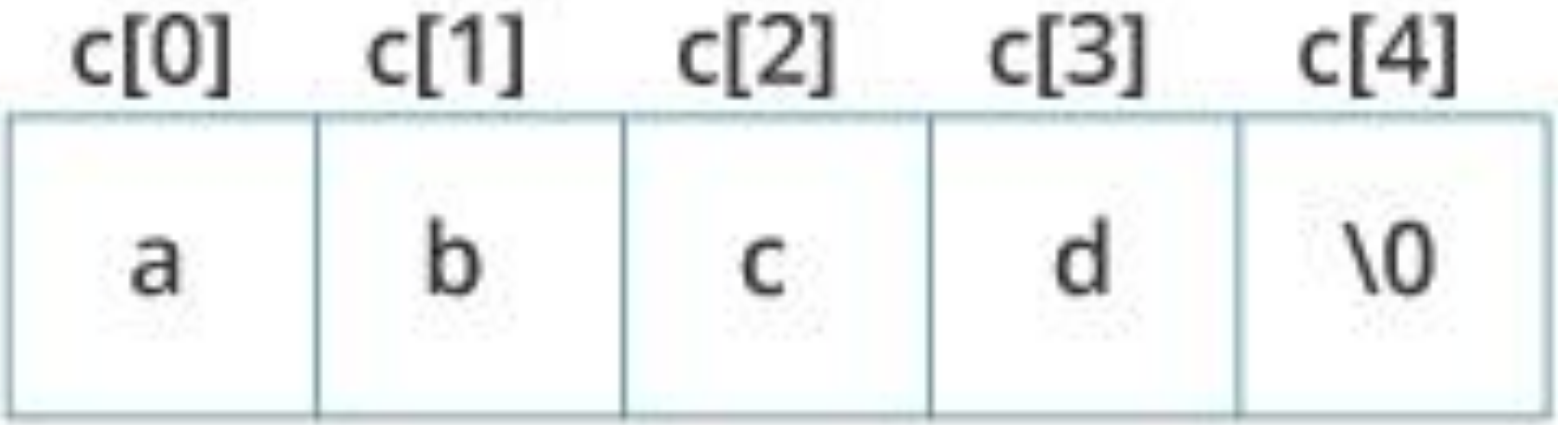
How to initialize strings?
You can initialize strings in a number of ways.

```
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};
```


c[0]
c[1]
c[2]
c[3]
c[4]
a
b
c
d
\0

Let's take another example:

```
char c[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters.

This is bad and you should never do this.

Read String from the user

scanf() function to read a string

The scanf() function reads the sequence of characters until it encounters whitespace

(space, newline, tab etc.).

```c
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Enter name: Dennis Ritchie
Your name is Dennis.

Even though Dennis Ritchie was entered in the above program, only “Dennis" was stored in the name string.

It's because there was a space after Dennis.

How to read a line of text?

You can use the fgets() function to read a line of string / Or use gets()

You can use puts() to display the string.

```c
#include <stdio.h>
int main()
{
    char name[100];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);
        // read string
    printf("Name: ");
    puts(name);    // display string
    return 0;
}
```

Output

Enter name: Tom Hanks
Name: Tom Hanks

scanf() : default input is keyboard

printf() : default output is screen

fscanf(… ,ftpr) :	man fscanf()
fprintf(… ,ftpr);	man fprintf()
gets(…);	// problem
fgets(….,stdin);	// We use this function

Here, we have used fgets() function to read a string from the user.

```
fgets(name, sizeof(name), stdin);

OR gets(name);

// read string
```

The sizeof(name) results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the name string.

To print the string, we have used puts(name);.

Note: The gets() function can also be to take input from the user. However, it is removed from the C standard.

It's because gets() allows you to input any length of characters. Hence, there might be a buffer overflow.

Passing Strings to Functions

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
   char str[50];
   printf("Enter string: ");
   fgets(str, sizeof(str), stdin);
   displayString(str);
// Passing string to a function.
   return 0;
}
```

```
void displayString(char str[])
{
    printf("String Output: ");
        //printf(“%s”, str);
    //puts(str);
}
```

Strings and Pointers

Can use pointers to manipulate elements of the string.

```c
#include <stdio.h>

int main(void) {

  char name[] = "Harry Potter";

  printf("%c", *name);     // Output: H
  printf("%c", *(name+1));   // Output: a
  printf("%c", *(name+7));   // Output: o
```

```
  char *namePtr;

  namePtr = name;
  printf("%c", *namePtr);     // Output: H
  printf("%c", *(namePtr+1));   // Output: a
  printf("%c", *(namePtr+7));   // Output: o
}
```

```
int main(){

        char array[10 ];

        array[0] =  ‘a’;
        array[1] =  ‘b’;
        array[2] =  ‘c’;
        array[3] =  ‘d’;
        array[4] =  ‘\0’;
        array[4] =  ….. /// Some garbage value –
        printf(“%s”, array);
}
```

strlen

The strlen() function takes a string as an argument and returns its length.

The returned value is of type long int.

It is defined in the <string.h> header file.

Example: C strlen() function

```
#include <stdio.h>
#include <string.h>
int main()
{
   char a[20]="Program";
   char b[20]={'P','r','o','g','r','a','m','\0'};

   printf("Length of string a = %ld \n",strlen(a));
   printf("Length of string b = %ld \n",strlen(b));

   return 0;
}
```

Output

Length of string a = 7
Length of string b = 7

Note that the strlen() function doesn't count the null character \0 while calculating the length.

strcpy()

The strcpy() function copies the string to the another character array.

The strcpy() function is defined in the string.h header file.

strcpy()

strcpy() Function prototype

```
char* strcpy(char* destination, const char* source);
```

strcpy()

The strcpy() function copies the string pointed by source (including the **null character**) to the character array destination.

The function also returns the copied array.

```
#include <stdio.h>
#include <string.h>

int main()
{
    char str1[10]=
"awesome";
    char str2[10];
    char str3[10];
    strcpy(str2, str1);
    strcpy(str3, "well");
    puts(str2);
    puts(str3);

    return 0;
}
```

Output

awesome
well

Destination array should be large enough to copy the array.

Otherwise, it may result in undefined behavior.

strcmp()

The strcmp() function compares two strings and returns 0 if both strings are identical.

It is defined in the string.h header file.

C strcmp() Prototype

int strcmp (const char* str1, const char* str2);

The strcmp() function takes two strings and returns an integer.

The strcmp() compares two strings character by character.

If the first character of two strings is equal, the next character of two strings are compared.

This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

Return Value from strcmp()

if both strings are identical (equal)	0
if the ASCII value of the first unmatched character is less than second.	negative
if the ASCII value of the first unmatched character is greater than second.	positive

```c
#include <stdio.h>
#include <string.h>

int main()
{
   char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
   int result;

   // comparing strings str1 and str2
   result = strcmp(str1, str2);
   printf("strcmp(str1, str2) = %d\n", result);

   // comparing strings str1 and str3
   result = strcmp(str1, str3);
   printf("strcmp(str1, str3) = %d\n", result);

   return 0;
}
```

Output

strcmp(str1, str2) = 32
strcmp(str1, str3) = 0

The first unmatched character between string str1 and str2 is third character.

The ASCII value of 'c' is 99 and the ASCII value of 'C' is 67.

Hence, when strings str1 and str2 are compared, the return value is 32.

When strings str1 and str3 are compared, the result is 0 because both strings are identical.

Implementation of strlen()

```
unsigned int my_strlen(char *p)
{
    unsigned int count = 0;

    while(*p!='\0')
    {
        count++;
        p++;
    }

    return count;
}
```

```
int mystrlen(char *str)
{
    int len = 0;
    int i;

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

Implementation of strcpy()

```
char *my_strcpy(char *destination, const char *source)
{
    char *start = destination;

    while(*source != '\0')
    {
        *destination = *source;
        *source = ‘5’;
        destination++;
        source++;
    }

    *destination = '\0'; // add '\0' at the end
    return start;
}
```

Its same as assignment (by copying)

a = b

b is copied to a …

a ←b

The same thing in strcpy

strcpy(array1,array2)

equivalent to array1← array2

Implementation of strcmp()

```
strcmp("a", "a");
// returns 0 as ASCII value of
"a" and "a" are same i.e 97

strcmp("a", "b");
// returns -1 as ASCII value of
"a" (97) is less than "b" (98)
```

```
strcmp("a", "c");
// returns -1 as ASCII value of "a" (97) is less than "c" (99)

strcmp("z", "d");
// returns 1 as ASCII value of "z" (122) is greater than "d" (100)
```

```
strcmp("abc", "abe");
// returns -1 as ASCII value of
"c" (99) is less than "e" (101)

strcmp("apples", "apple");
// returns 1 as ASCII value of
"s" (115) is greater than "\0"
(101)
```

```
int my_strcmp(char *strg1, char *strg2)
{
    while( ( *strg1 != '\0' && *strg2 != '\0' )
            && *strg1 == *strg2 )
    {
        strg1++;
        strg2++;
    }

    if(*strg1 == *strg2) return 0; // identical

    else return *strg1 - *strg2;
}
```

Implementation of strcat()

```
char *my_strcat(char *strg1, char *strg2)
{
    char *start = strg1;

    while(*strg1 != '\0')
    {
        strg1++;
    }

    while(*strg2 != '\0')
    {
        *strg1 = *strg2;
        strg1++;
        strg2++;
    }

    *strg1 = '\0';
    return start;
}
```

Strings handling functions are defined under "string.h" header file.

#include <string.h>

Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

Sort strings in Lexicographical Order (Dictionary Order)

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[5][50], temp[50];
    printf("Enter 5 words: ");


// Getting strings input
    for (int i = 0; i < 5; ++i) {
        fgets(str[i], sizeof(str[i]), stdin);
    }
```

```
// storing strings in the lexicographical order
    for (int i = 0; i < 5; ++i) {
        for (int j = i + 1; j < 5; ++j) {
            if (strcmp(str[i], str[j]) > 0) {
                strcpy(temp, str[i]);
                strcpy(str[i], str[j]);
                strcpy(str[j], temp);
            }
        }
    }                   printf("\nIn the lexicographical order: \n");
                        for (int i = 0; i < 5; ++i) {
                            fputs(str[i], stdout);
                        }
                        return 0;
                    }
```

Enter 5 words: R programming
JavaScript
Java
C programming
C++ programming

In the lexicographical order:
C programming
C++ programming
Java
JavaScript
R programming

Array of pointers

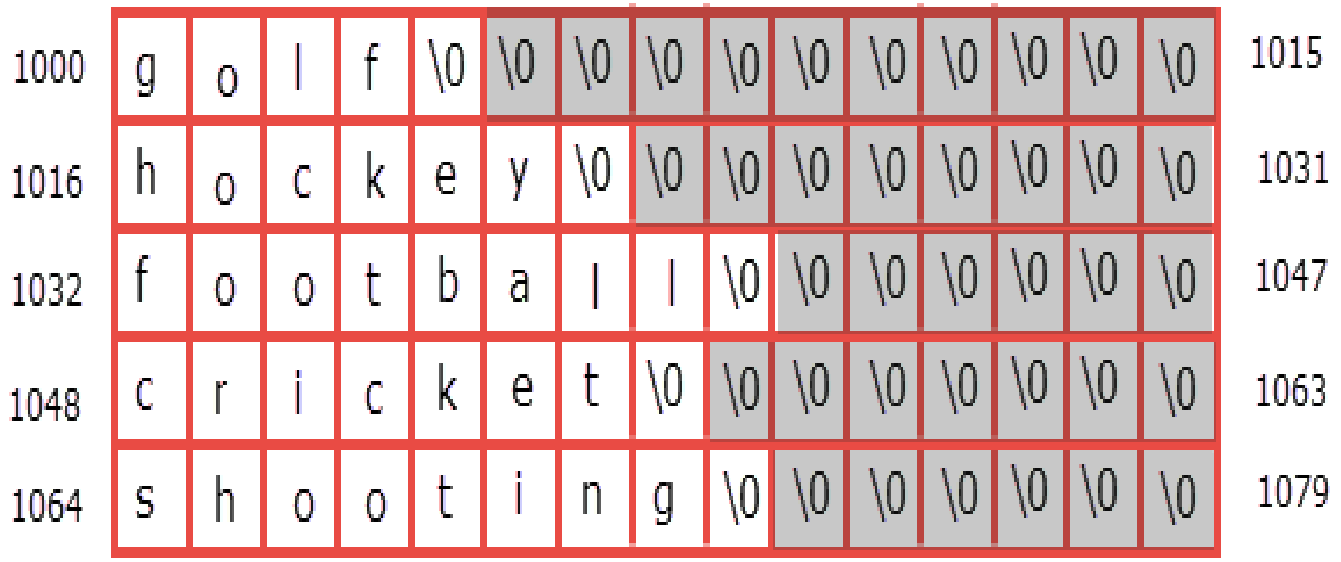
We can use an array of strings or 2-D array of characters.

It may appear to you whenever you need to store more than one string then an array of strings is the way to go, unfortunately, this wastes memory

```
char sports[5][15] = {
                        "golf",
                        "hockey",
                        "football",
                        "cricket",
                        "shooting"
                     };
```

sports array is stored in the memory as follows:

sports[5][15]



Not all strings are long enough to fill all the rows of the array - that’s why compiler fills these empty spaces (highlighted using light grey color) with the null characters ('\0').

The total size of the sports array is 75 bytes but only 34 bytes is used, 41 bytes is wasted.

41 bytes may not appear much but in a large program, a considerable amount of bytes would be wasted.

What we need is a jagged array:

A 2-D array whose rows can be of different length. C doesn't provide jagged arrays but we can simulate them using an **array of pointer to a string.**

An array of pointers to strings is an array of character pointers where each pointer points to the first character of the string or the base address of the string.

Let’s see how we can declare and initialize an array of pointers to strings.

char *p = “5”;

….. 5 \0

printf(“%d”,p);

float f = 1.7;

&6.4f

```
char *a = “one string”;
char b[40] = “one string”;

char *sports[5] = {
                    "golf",
                    "hockey",
                    "football",
                    "cricket",
                    "shooting"
                  };
```

```
char *a = “one string”;
char b[40] = “one string”;

char *sports_ptr[5];


sports_ptr[0] = sports[0];
sports_ptr[1] = sports[2];
…
sports_ptr[4] = sports[3];
```

```
char *a = “one string”;
char b[40] = “one string”;

char *sports[5] = {
                    "cricket",
                    "football",
                     "golf",
                    "hockey",
                   "shooting"
                  };
```

Here sports is an array of pointers to strings.

If initialization of an array is done at the time of declaration then we can omit the size of an array.

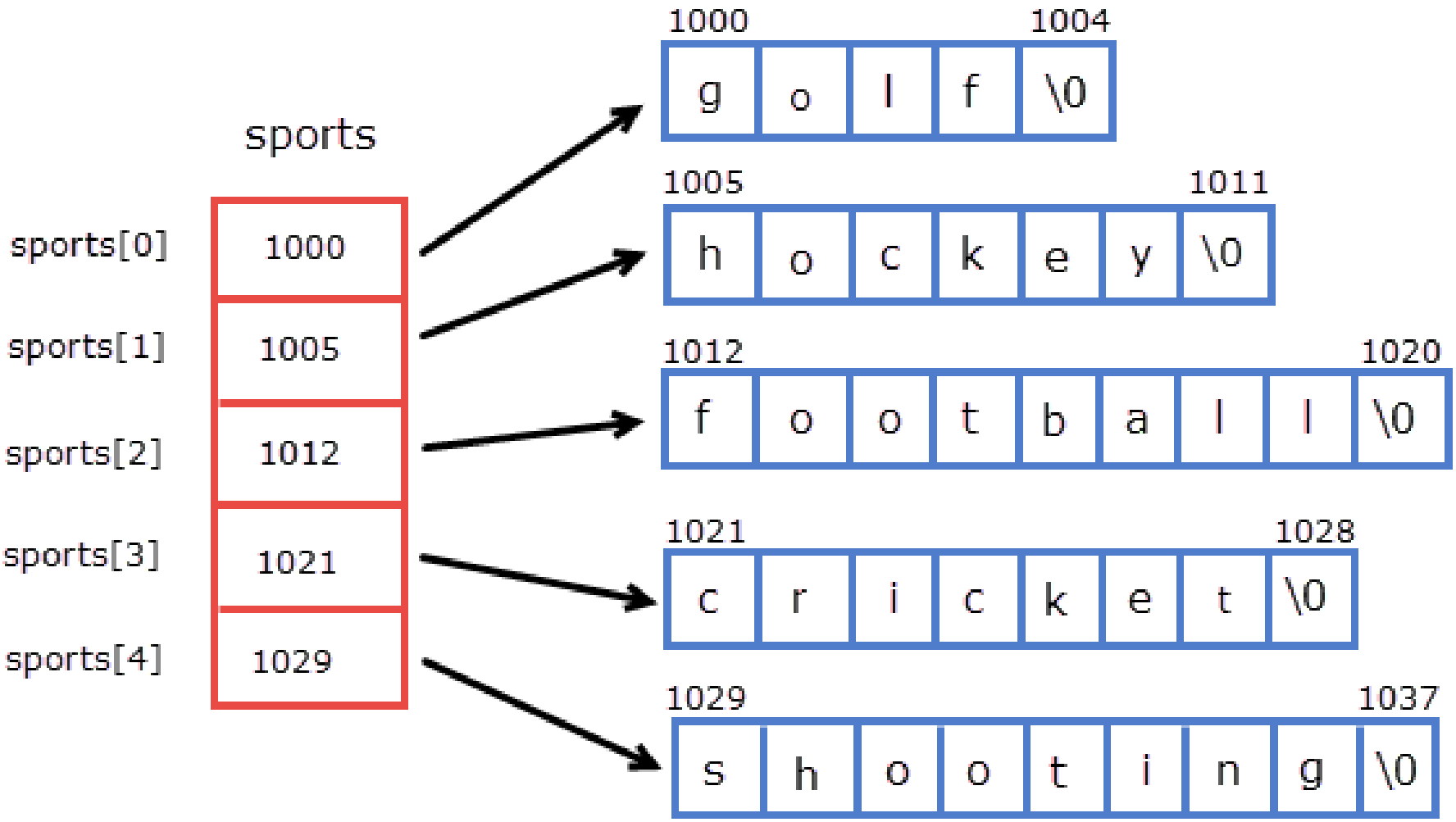
So the above statement can also be written as:

```
char *sports[] = {
                    "golf",
                    "hockey",
                    "football",
                    "cricket",
                    "shooting"
                  };
```

It is important to note that each element of the sports array is a string literal and since a string literal points to the base address of the first character, the base type of each element of the sports array is a pointer to char or (char*).

The 0th element i.e arr[0] points to the base address of string "golf". Similarly, the 1st element i.e arr[1] points to the base address of string "hockey" and so on.

Here is how an array of pointers to string is stored in memory.



34 + 20 = 54

In this case, all string literals occupy 34 bytes and 20 bytes are occupied by the array of pointers i.e sports.

So, just by creating an array of pointers to string instead of array 2-D array of characters we are saving 21 bytes (75-54=21) of memory.

It is important to emphasize that in an array of pointers to strings, it is not guaranteed that the all the strings will be stored in contiguous memory locations.

Although the characters of a particular string literal are always stored in contiguous memory location.

```c
#include<stdio.h>
#include<string.h>
int factorial(int );

int main()
{
    int i = 1, *ip = &i;

    char *sports[] = {
                        "golf",
                        "hockey",
                        "football",
                        "cricket",
                        "shooting"
                    };

for(i = 0; i < 5; i++)
    {
        printf("String = %10s",
        sports[i] );
        printf("\tAddress of string
        literal = %u\n", sports[i]);
    }

    // signal to operating
   system program ran fine
    return 0;
}
```

String = golf Address of string literal = 4206592
String = hockey Address of string literal = 4206597
String = football Address of string literal = 4206604
String = cricket Address of string literal = 4206613
String = shooting Address of string literal = 4206621

Dynamic memory allocation

An array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Array size you declared may be insufficient. We can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

Library

To allocate memory dynamically, library functions are

malloc(),
calloc(),
realloc() and
free()

These functions are defined in the <stdlib.h> header file.

malloc()

The name "malloc" stands for memory allocation.

The malloc() function reserves a block of memory of the specified number of bytes.

It returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

The expression results in a NULL pointer if the memory cannot be allocated.

Syntax of calloc()

ptr = (castType*)calloc(n, size);

Example:

ptr = (float*) calloc(25, sizeof(float));
The above statement allocates contiguous space in memory for 25 elements of type float.

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

free(ptr);
This statement frees the space allocated in the memory pointed by ptr.

```
// Program to calculate the sum of n numbers entered by the
user

#include <stdio.h>
#include <stdlib.h>

int main()
{
   int n, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) malloc(n * sizeof(int));

   // if memory cannot be allocated
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
```

```
    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    // deallocating the memory
    free(ptr);
    return 0;
}
```

```
// Program to calculate the sum of n
numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main()
{
   int n, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) calloc(n, sizeof(int));
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }

   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```