CMPE 252
C PROGRAMMING

SPRING 2021

WEEK 7-8

STRINGS
CHAPTER 8

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

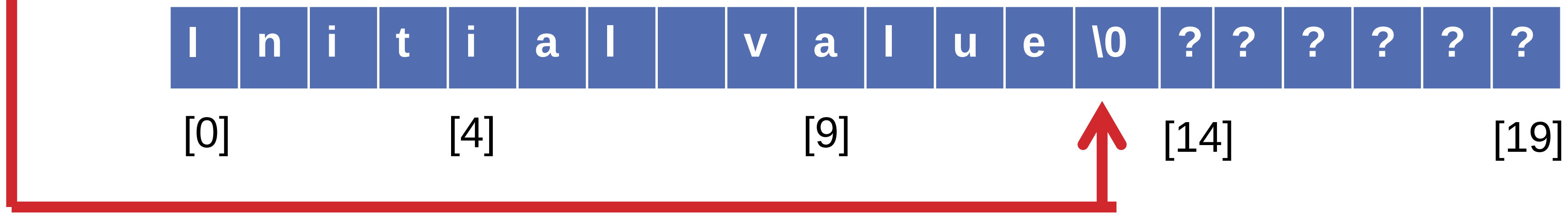
- To understand how a string constant is stored in an array of characters
- To learn about the placeholder %s and how it is used in printf and scanf operations
- To learn some of the operations that can be performed on strings such as copying strings extracting substrings, and joining strings using functions from the library string

Chapter Objectives

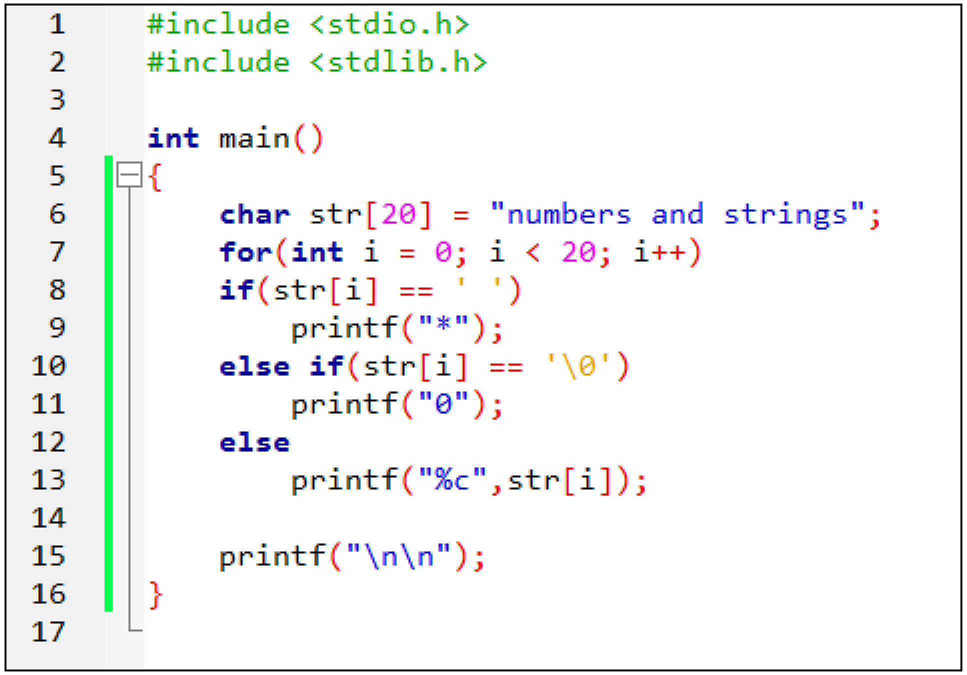
- To understand the buffer overflow dangers inherent in some string library functions
- To learn how C compares two strings to determine their relative order
- To see some of the operations that can be performed on individual characters using functions form the library ctype
- To learn how to write your own functions that perform some of the basic operations of a text editor program
- To understand basic principles of defensive programming

String Basics

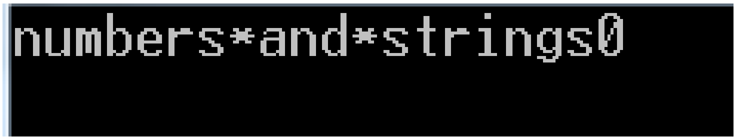
- A blank in a string is a valid character.
- null character
  - character ‘\0’ that marks the end of a string in C
- A string constant can be associated with a symbolic name using #define directive
  - #define ERR_PREFIX   " *******Error- "
- A string in C is implemented as an array.
  - char string_var[30];
  - char str[20] = “Initial value”;



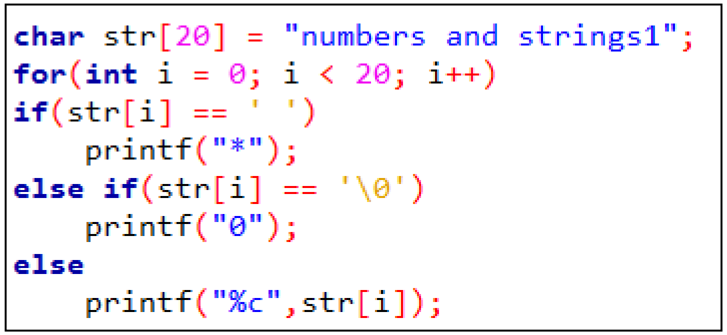
String Basics



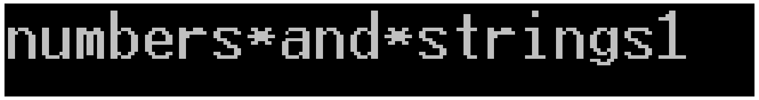
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char str[20] = "numbers and strings";
    for(int i = 0; i < 20; i++)
    if(str[i] == ' ')
        printf("*");
    else if(str[i] == '\0')
        printf("0");
    else
        printf("%c",str[i]);

    printf("\n\n");
}
```



```
numbers*and*strings0
```



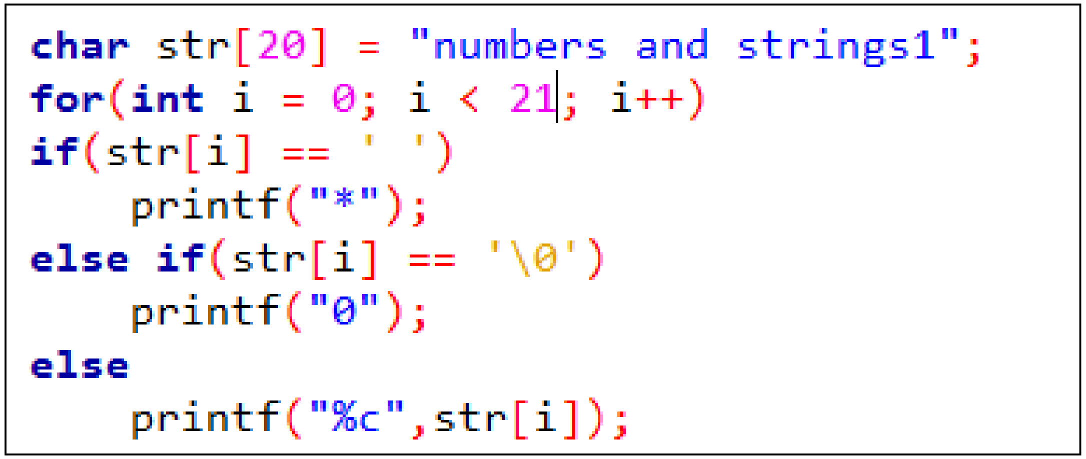
```c
char str[20] = "numbers and strings1";
for(int i = 0; i < 20; i++)
if(str[i] == ' ')
    printf("*");
else if(str[i] == '\0')
    printf("0");
else
    printf("%c",str[i]);
```



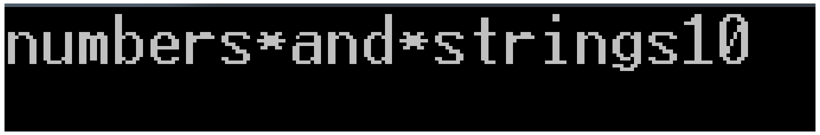
```
numbers*and*strings1
```

Where is \0 then?

String Basics

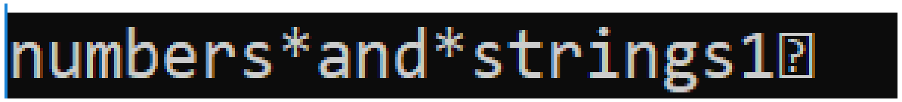


```
char str[20] = "numbers and strings1";
for(int i = 0; i < 21; i++)
if(str[i] == ' ')
    printf("*");
else if(str[i] == '\0')
    printf("0");
else
    printf("%c",str[i]);
```



```
numbers*and*strings10
```

Output in one computer



```
numbers*and*strings1⍰
```

Output in another computer

String Basics

- An array of strings is a 2-dimensional array of characters in which each row is a string.
- **Quick Check:** declare an array of strings which keeps names (max. 25 char) of 30 people
  - char names [30][25]
  - Remember that in multidim. arrays, grouping is done row by row
  - We need 30 rows for people

Array of String Initialization at Declaration

- char month [12] [10] = { "January", "February", "March", "April", " May", " June", " July", " August",

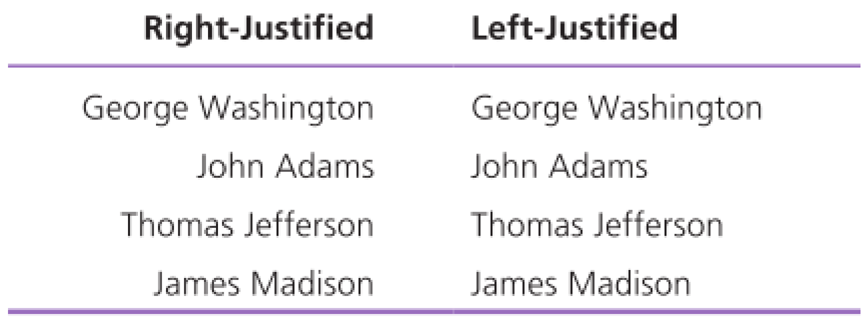
" September", " October", " November", " December" }

Input/Output

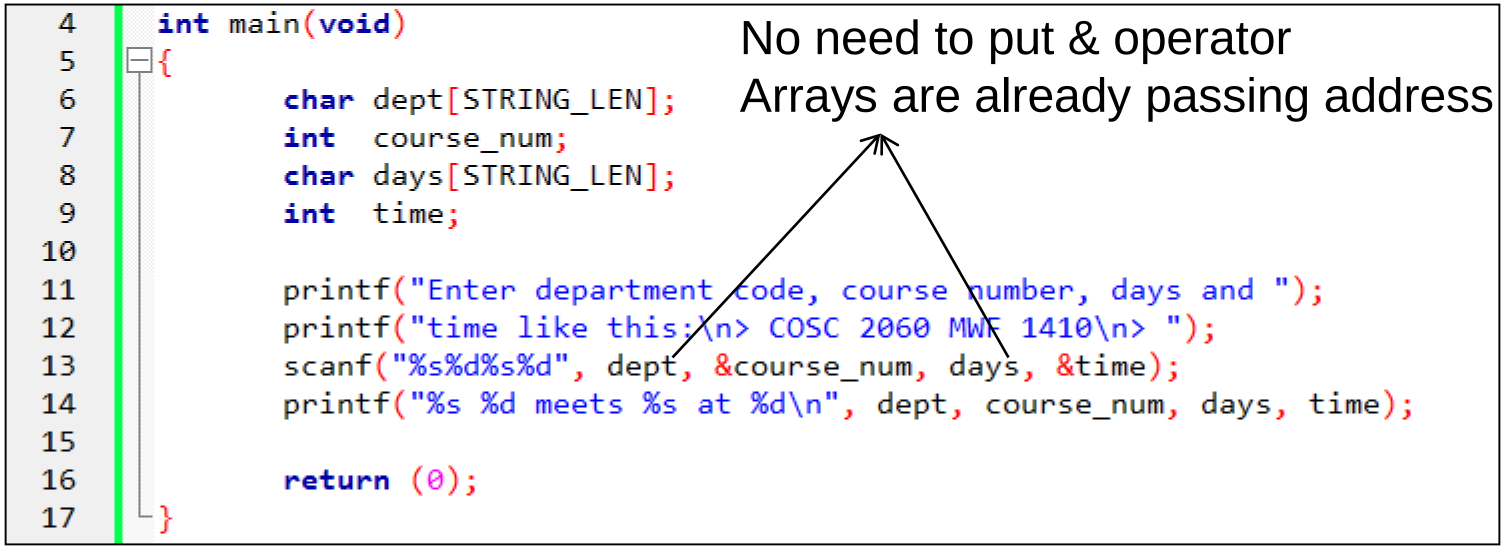
- printf and scanf can handle string arguments
- use %s as the placeholder in the format string
- use a – (minus) sign to force left justification
  - printf(“%-20s\n”, president);

FIGURE 8.1

Right and Left Justification of Strings

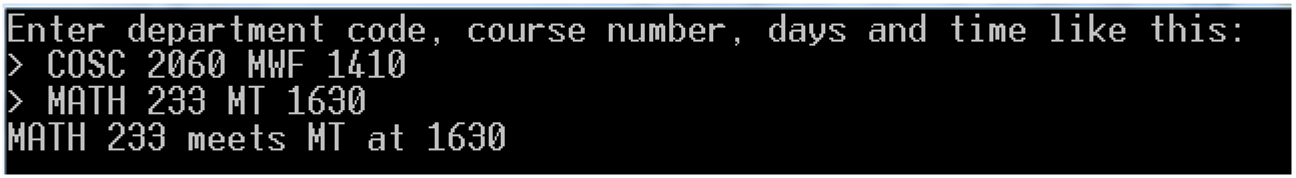


Right-Justified	Left-Justified
George Washington	George Washington
John Adams	John Adams
Thomas Jefferson	Thomas Jefferson
James Madison	James Madison

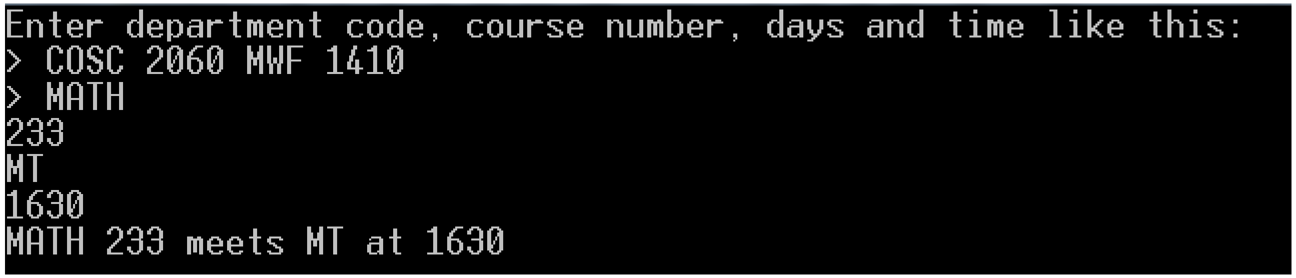


```
int main(void)
{
        char dept[STRING_LEN];
        int  course_num;
        char days[STRING_LEN];
        int  time;

        printf("Enter department code, course number, days and ");
        printf("time like this:\n> COSC 2060 MWF 1410\n> ");
        scanf("%s%d%s%d", dept, &course_num, days, &time);
        printf("%s %d meets %s at %d\n", dept, course_num, days, time);

        return (0);
}
```

No need to put & operator
Arrays are already passing address

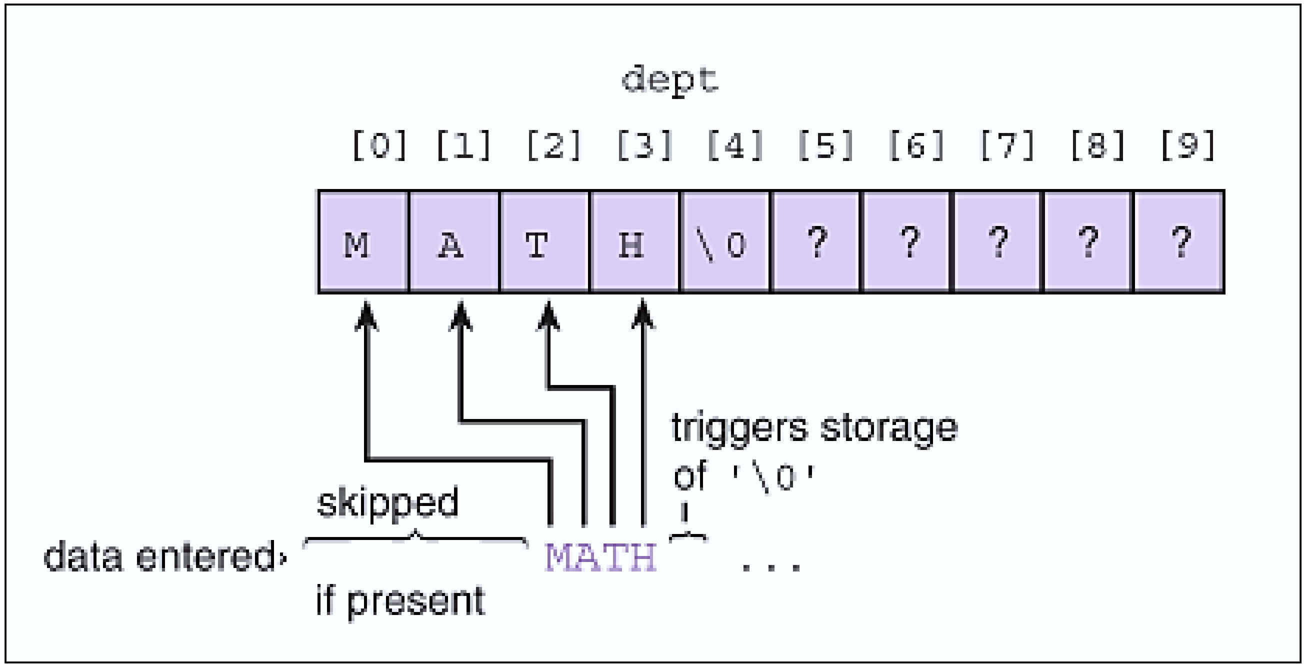


```
Enter department code, course number, days and time like this:
> COSC 2060 MWF 1410
> MATH 233 MT 1630
MATH 233 meets MT at 1630
```



```
Enter department code, course number, days and time like this:
> COSC 2060 MWF 1410
> MATH
233
MT
1630
MATH 233 meets MT at 1630
```

values can be spaced in many ways, treating whitespace is important


dept
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
M
A
T
H
\0
?
?
?
?
?
skipped
if present
data entered›
MATH
triggers storage of '\0'
. . .

Function `scanf` would have difficulty if some essential whitespace between values were omitted or if a nonwhitespace separator were substituted. For example, if the data were entered as

```
> MATH1270 TR 1800
```

`scanf` would store the eight-character string `"MATH1270"` in `dept` and would then be unable to convert `T` to an integer for storage using the next parameter. The situation would be worse if the data were entered as

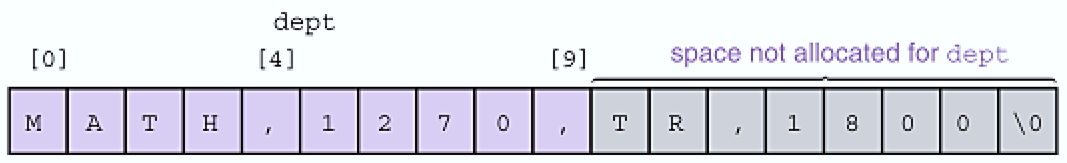
```
> MATH,1270,TR,1800
```

Then the `scanf` function would store the entire 17-character string plus `'\0'` in the `dept` array, causing characters to be stored in eight locations not allocated to `dept`, as shown in Fig. 8.4.

Buffer Overflow

- more data is stored in an array than its declared size allows
- a very dangerous condition
- unlikely to be flagged as an error by either the compiler or the run-time system

FIGURE 8.4 Execution of `scanf("%s%d%s%d", dept, &course_num, days, &time);` on Entry of Invalid Data

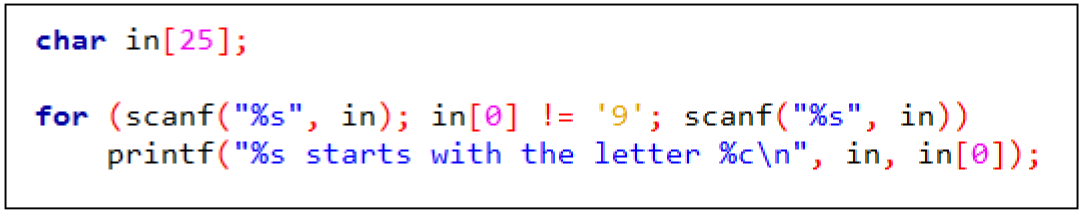

dept
[0]
[4]
[9]
space not allocated for dept
M A T H , 1 2 7 0 , T R , 1 8 0 0 \0

Quick Check

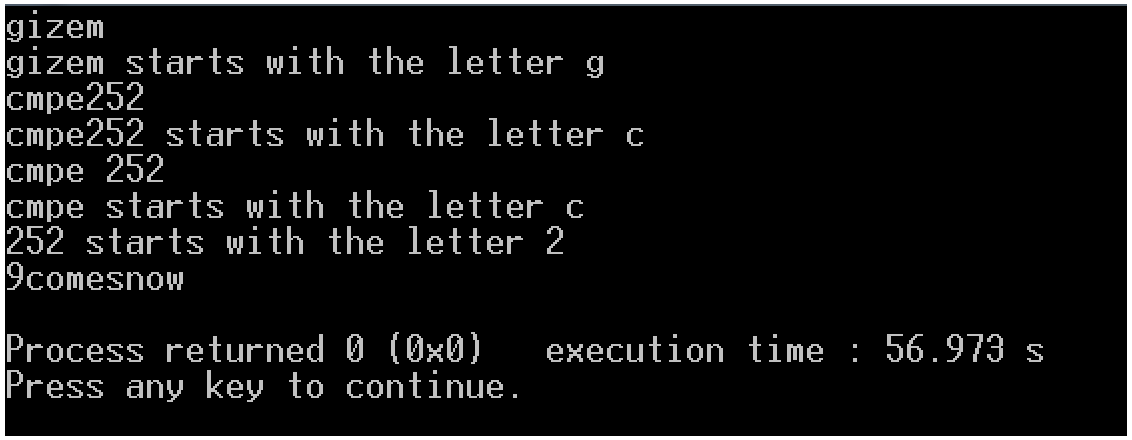
Write a program that takes a word less than 25 characters and prints a statement like this:

fractal starts with letter f

Have the program process words until it encounters a word beginning with the character '9'



```c
char in[25];

for (scanf("%s", in); in[0] != '9'; scanf("%s", in))
    printf("%s starts with the letter %c\n", in, in[0]);
```



```
gizem
gizem starts with the letter g
cmpe252
cmpe252 starts with the letter c
cmpe 252
cmpe starts with the letter c
252 starts with the letter 2
9comesnow

Process returned 0 (0x0)   execution time : 56.973 s
Press any key to continue.
```

= operator

- char one_str[20] = "Test string"; ✓

- char one_str[20] ; ✗
- one_str = "Test string";

Array name with no subscript is an address, a pointer to initial array element. This address is constant which cannot be changed through assignment.

String Terminology

- string length
  - in a character array, the number of characters before the first null character

- empty string
  - a string of length zero
  - the first character of the string is the null character

string.h library

Function	Purpose	Parameters	Result Type
strlen	*Returns the number of characters without null character at the end* *strlen*("hello") returns 5	const char* s1	size_t

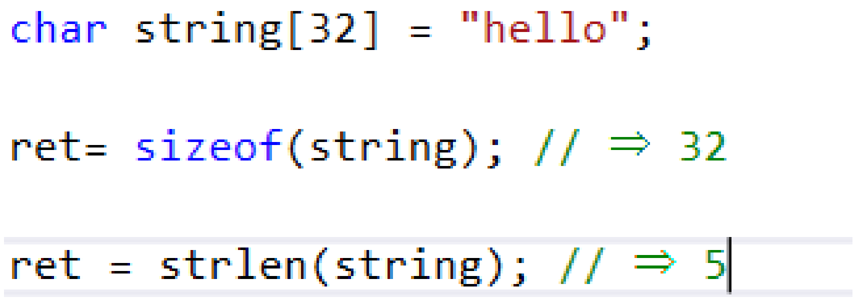
(In other words, it returns the offset of the terminating null byte within the array.)

```
strcpy(dest, "hello");
printf("%d",strlen(dest));
```

```
5
```

strlen

- When applied to an array, the strlen function returns length of the string stored there, not its allocated size.
- You can get the allocated size of the array that holds a string using the sizeof operator:



```
char string[32] = "hello";

ret= sizeof(string); // ⇒ 32

ret = strlen(string); // ⇒ 5
```

```
char *sptr = string;

ret = strlen(sptr); // ⇒ 5

ret = sizeof(sptr); // ⇒ 4
```

string.h library

Function	Purpose	Parameters	Result Type
strcpy	makes a copy of string *source* in the char array *dest* *strcpy(s1, "hello")* (up to and including the terminating null byte)	char* dest const char* source	char* hello\0????? (The return value is the value of *dest*)

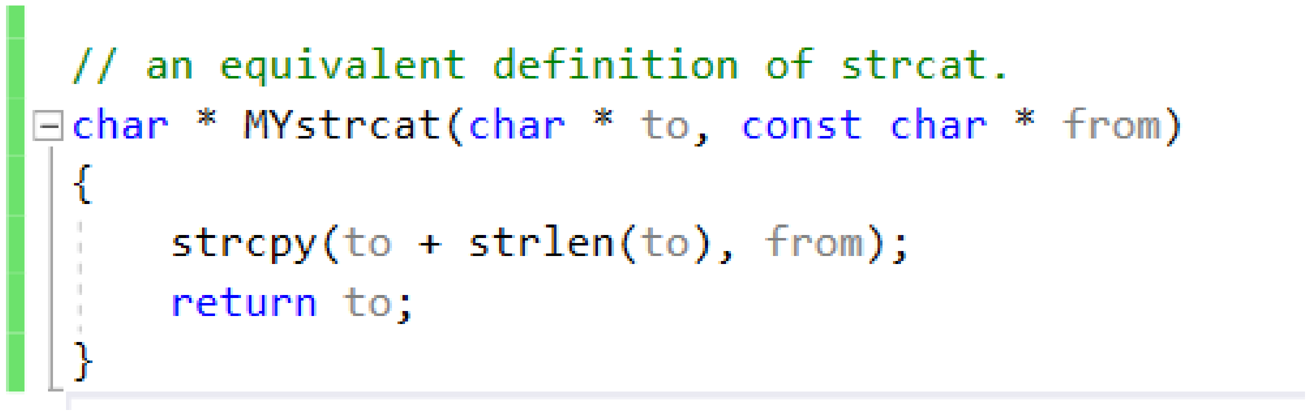
string.h library

Function	Purpose	Parameters	Result Type
strncpy	makes a copy of n characters of string *source* in the char array *dest* without null character *strncpy(s2, "hello",3)* If *source* contains a null byte within its first *n* bytes (i.e. length <n), strncpy copies all of *source*, followed by enough null bytes to add up to *n* bytes in all.	char* dest const char* source size_t n	char* hel?????

The function needs to set all n bytes of the destination, even when n is much greater than the length of source. (GNU C)

string.h library

Function	Purpose	Parameters	Result Type
strcat	appends *source* to the end of *dest* strcat(s1, "*hello*") the first byte from *source* overwrites the null byte marking the end of *dest (concatenates)*	char* dest const char* source	char* hellohello\0??



```
// an equivalent definition of strcat.
char * MYstrcat(char * to, const char * from)
{
    strcpy(to + strlen(to), from);
    return to;
}
```

```
char word[] = "hello";
char dest[6];
strcpy(dest, word);
printf("%s\n", dest);
```

hello

```
char dest[5];
strncpy(dest, "hello", 3);
dest[3] = '\0';
printf("%s\n", dest);
```

hel

```
char dest[10];
strncpy(dest, "hello", 3);
dest[3] = '\0';
strcat(dest, "hello");
printf("%s", dest);
```

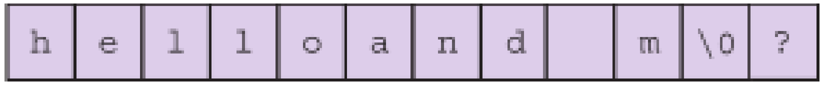
helhello

```
 //one_str has room for 14 characters
 //+ null character
 char one_str[15];

 //Size is enough, no problem exists
 strcpy(one_str, "Test string");

 //Size is not enough, may cause problem
 //of inserting the rest of the characters in
 //another string
 strcpy(one_str, "A very long test string");

//THE BEST APPROACH
size_t len = sizeof(one_str) / sizeof(one_str[0]);
printf("array max size is: %d\n", len);
strncpy(one_str,"A very long test string",(len-1));
one_str[len-1] = '\0';
puts(one_str);
```

string.h library

Function	Purpose	Parameters	Result Type
strncat	appends up to n characters of source to the end of dest, adding the null character if necessary. A single null byte is also always appended to *dest* , so the total allocated size of *dest* must be at least *n* + 1 bytes longer than its initial length.	char* dest const char* source size_t n	char*

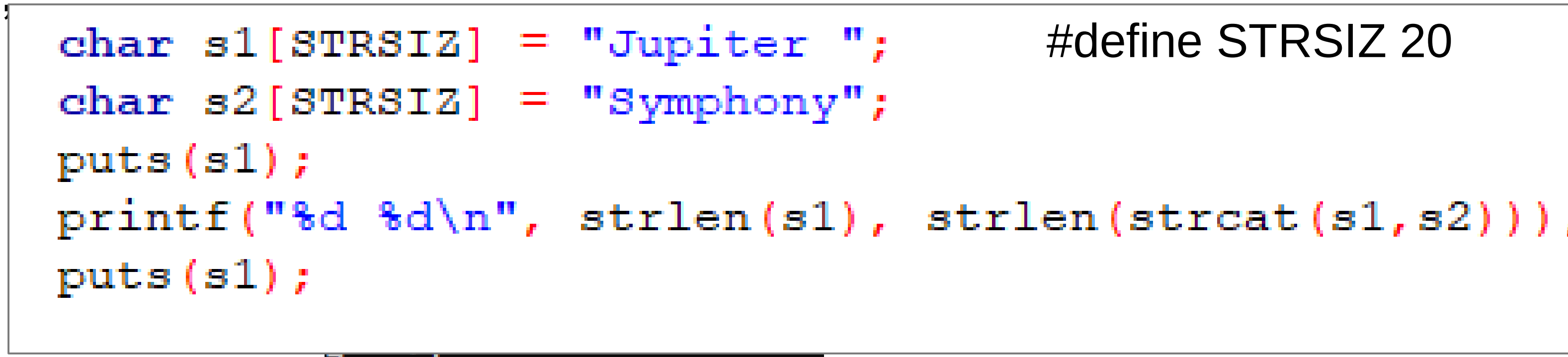
```
char s1[12] = “hello”;
strncat(s1, “and more”, 5);
```



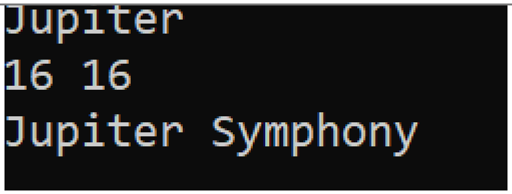
Space Problem

- Always ensure that the size is enough to hold the data and ‘\0’

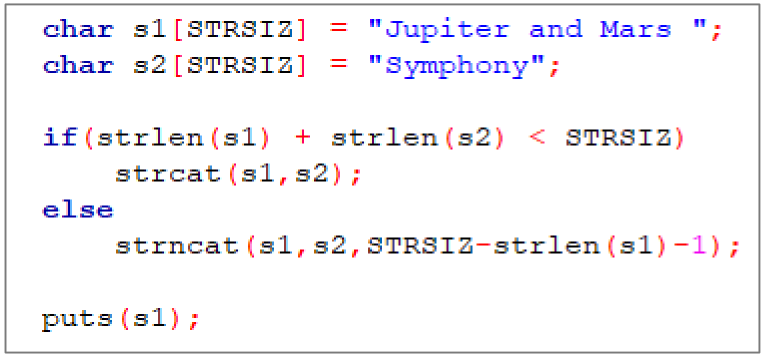
#define STRSIZ 20



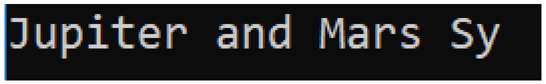
```
char s1[STRSIZ] = "Jupiter ";
char s2[STRSIZ] = "Symphony";
puts(s1);
printf("%d %d\n", strlen(s1), strlen(strcat(s1,s2)))
puts(s1);
```



```
Jupiter
16 16
Jupiter Symphony
```



```
char s1[STRSIZ] = "Jupiter and Mars ";
char s2[STRSIZ] = "Symphony";

if(strlen(s1) + strlen(s2) < STRSIZ)
    strcat(s1,s2);
else
    strncat(s1,s2,STRSIZ-strlen(s1)-1);

puts(s1);
```



```
Jupiter and Mars Sy
```

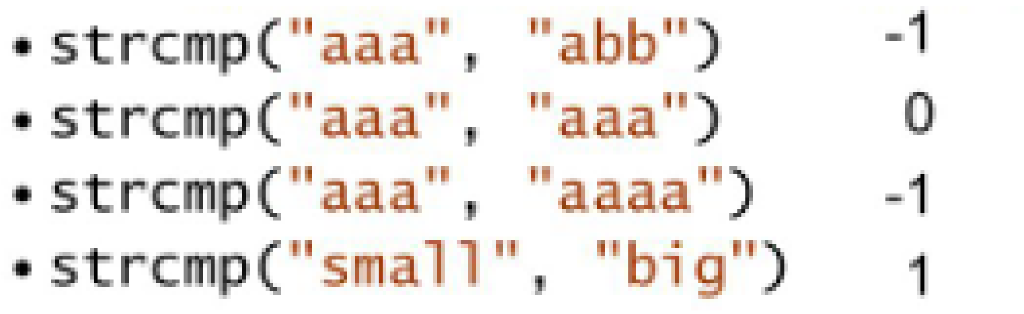
string.h library

Function	Purpose	Parameters	Result Type
strcmp	Compares s1 and s2 alphabetically. • Returns negative if s1 precedes s2, • 0 if equal, • Positive if s2 precedes s1	const char* s1 const char* s2	int

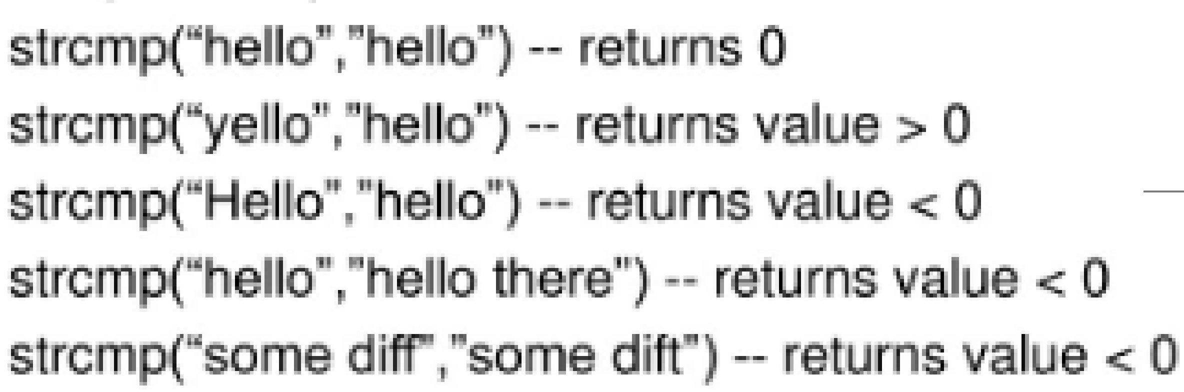
The strcmp function compares the string s1 against s2, returning a value that has the same sign as the **difference between the first differing pair of bytes** (interpreted as unsigned char objects, then promoted to int).

Note : we can not use ==,<, > for strings

Strcmp examples



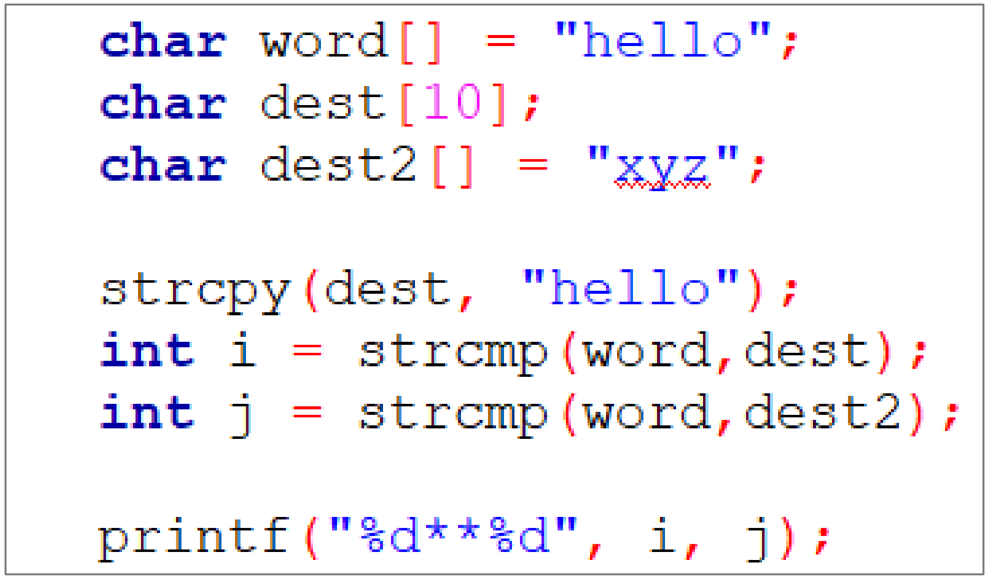
```
• strcmp("aaa", "abb")     -1
• strcmp("aaa", "aaa")      0
• strcmp("aaa", "aaaa")    -1
• strcmp("small", "big")    1
```



```
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
```

Uppercase letters < Lowercase in ASCII Table

Expression !strcmp(s1,s2) -> what does this mean ?



```
char word[] = "hello";
char dest[10];
char dest2[] = "xyz";

strcpy(dest, "hello");
int i = strcmp(word,dest);
int j = strcmp(word,dest2);

printf("%d**%d", i, j);
```

0**-1

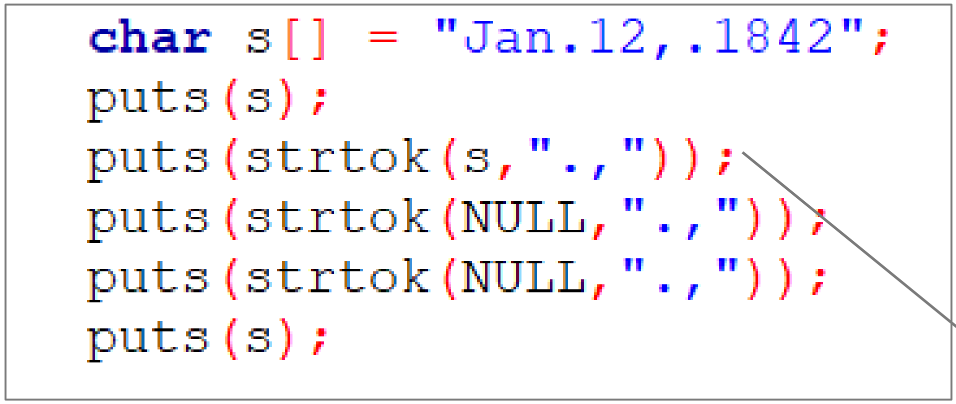
string.h library

Function	Purpose	Parameters	Result Type
strtok	*Breaks parameter string source into tokens by using* *any of the delimiter characters* *«series of calls to strtok are performed to split all tokens»*	char* source const char* delim *delim* argument is a string that specifies a set of delimiters that may surround the token being extracted.	char*

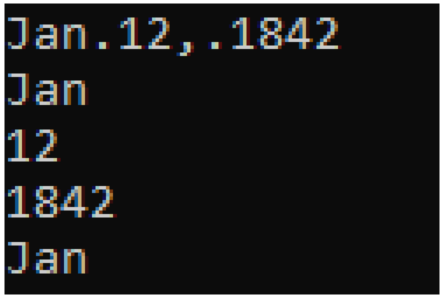
- The string to be split up is passed as the source <u>argument on the first call only</u>. The strtok function uses this to set up some internal state information.
- Subsequent calls to get additional tokens from the same string are indicated by passing a null pointer as the newstring argument.
- Calling strtok with another non-null source argument reinitializes the state information. It is guaranteed that no other library function ever calls strtok behind your back (which would mess up this internal state information).

strtok

s:	J	a	n	.	1	2	,	.	1	8	4	2	\0



```c
char s[] = "Jan.12,.1842";
puts(s);
puts(strtok(s,".,"));
puts(strtok(NULL,".,"));
puts(strtok(NULL,".,"));
puts(s);
```



```
Jan.12,.1842
Jan
12
1842
Jan
```

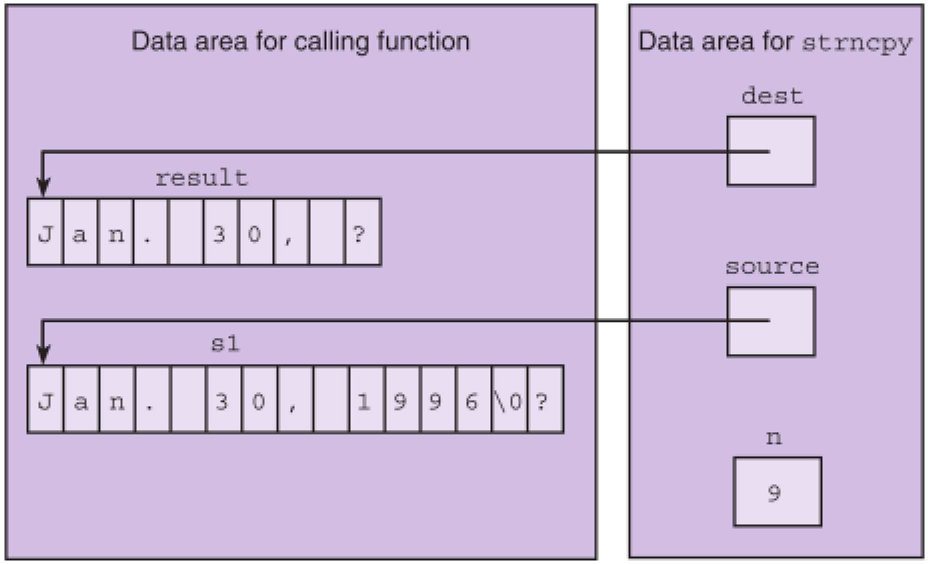
First call MUST provide both source and delim,
Others with NULL and delim

The first byte that is *not* a member of this set of delimiters marks the beginning of the next token. The end of the token is found by looking for the next byte that is a member of the delimiter set. This byte in the original string *source* is overwritten by a null byte, and the pointer to the beginning of the token in *source* is returned.

Substrings

- a fragment of a longer string

```
char result[10], s1[15] = "Jan. 30, 1996";
strncpy(result, s1, 9);
result[9] = '\0';
```



Substrings

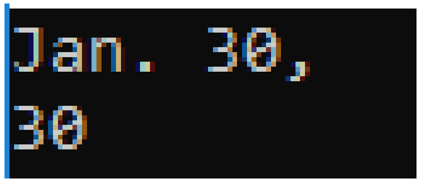
How to use strncpy to extract a middle substring

- Use the address of the first character to copy

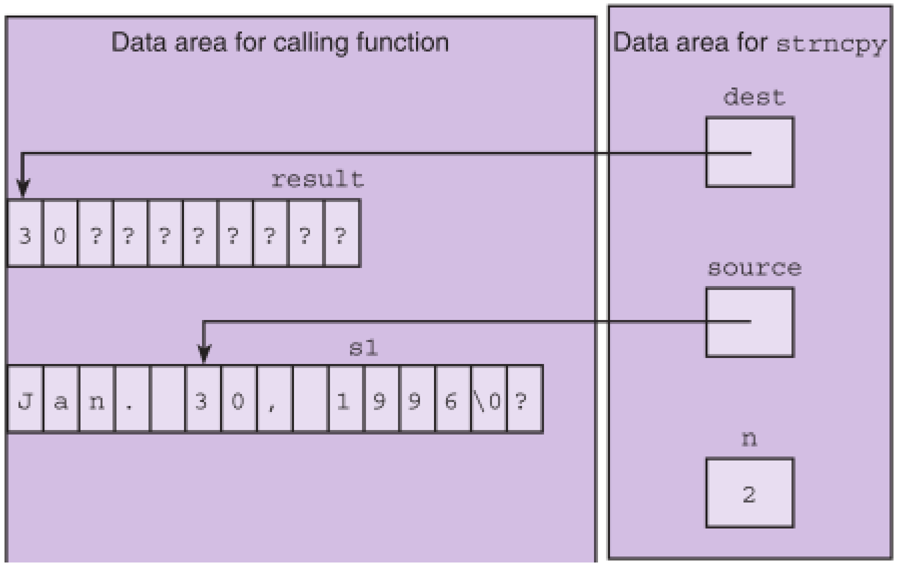
```
char result[10], s1[15] = "Jan. 30, 1996", sub[3];
strncpy(result, s1, 9);
result[9] = '\0';
puts(result);

strncpy(sub, &s1[5], 2);
sub[2] = '\0';
puts(sub);
```

Substrings

```
char result[10], s1[15] = "Jan. 30, 1996", sub[3];
strncpy(result, s1, 9);
result[9] = '\0';
puts(result);

strncpy(sub, &s1[5], 2);
sub[2] = '\0';
puts(sub);
```



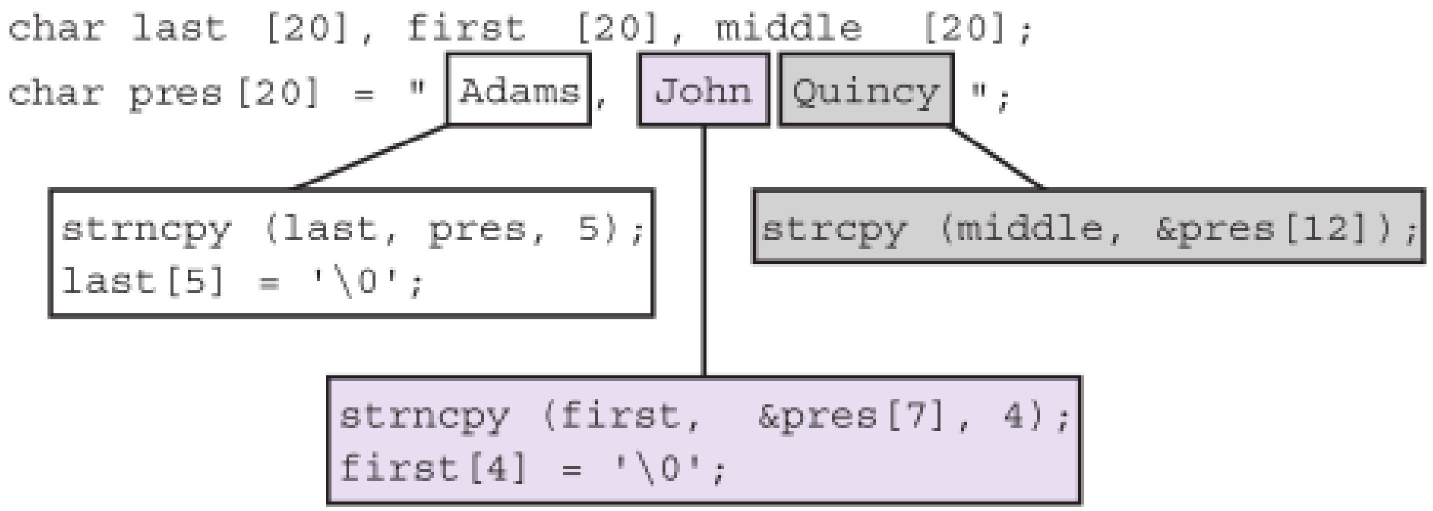
```
Jan. 30,
30
```



What if I write:
strcpy(result,&s1[9])?

Copies until '\0':
1996

Substrings



```
char last  [20], first  [20], middle  [20];
char pres[20] = " Adams, John Quincy ";
```

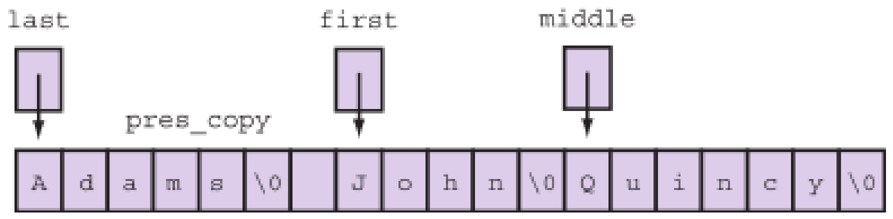
```
strncpy (last, pres, 5);
last[5] = '\0';
```

```
strcpy (middle, &pres[12]);
```

```
strncpy (first,  &pres[7], 4);
first[4] = '\0';
```

Substrings

```
char *last, *first, *middle;
char pres[20] = "Adams, John Quincy";
char pres_copy[20];
strcpy(pres_copy, pres);
```



```
last = strtok(pres_copy, ", ");
first = strtok(NULL, ", ");
middle = strtok(NULL, ", ");
```

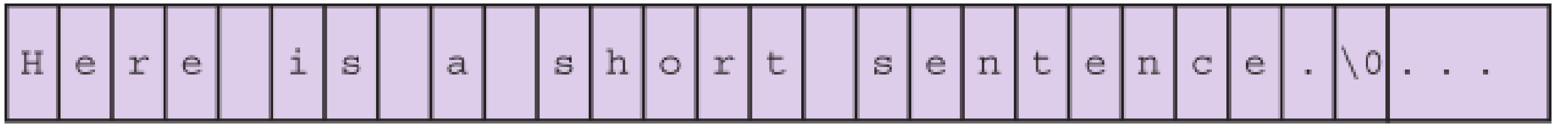
Scanning a Full Line

- For interactive input of one complete line of data, use the `gets` function.
- The `\n` character representing the <return> or <enter> key pressed at the end of the line is not stored.

Scanning a Full Line

```
char line[80];
printf("Type in a line of data.\n> ");
gets(line);
```

```
Type in a line of data.
> Here is a short sentence.
```

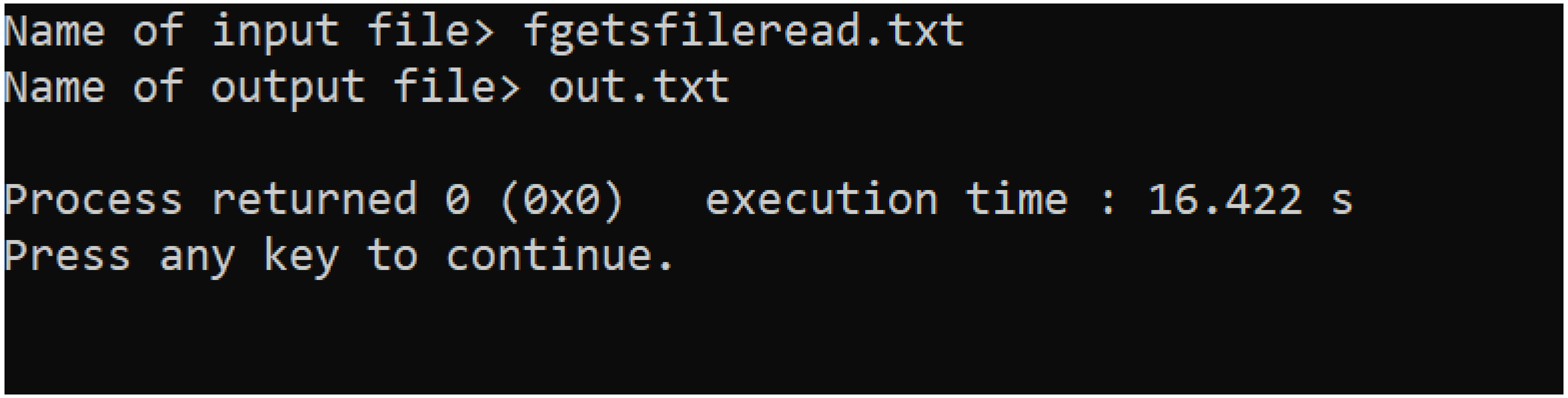


Get Line From File: fgets

The C library function **char *fgets(char *str, int n, FILE *stream)** reads a line from the specified stream and stores it into the string pointed to by **str**.

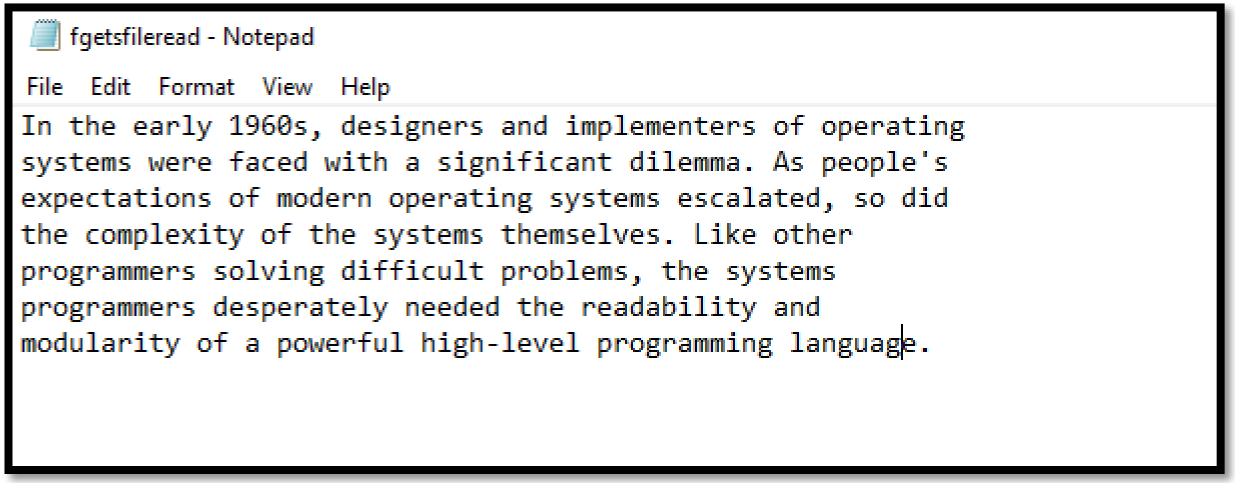
It stops when either **(n-1)**characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned. If an error occurs, a null pointer is returned.

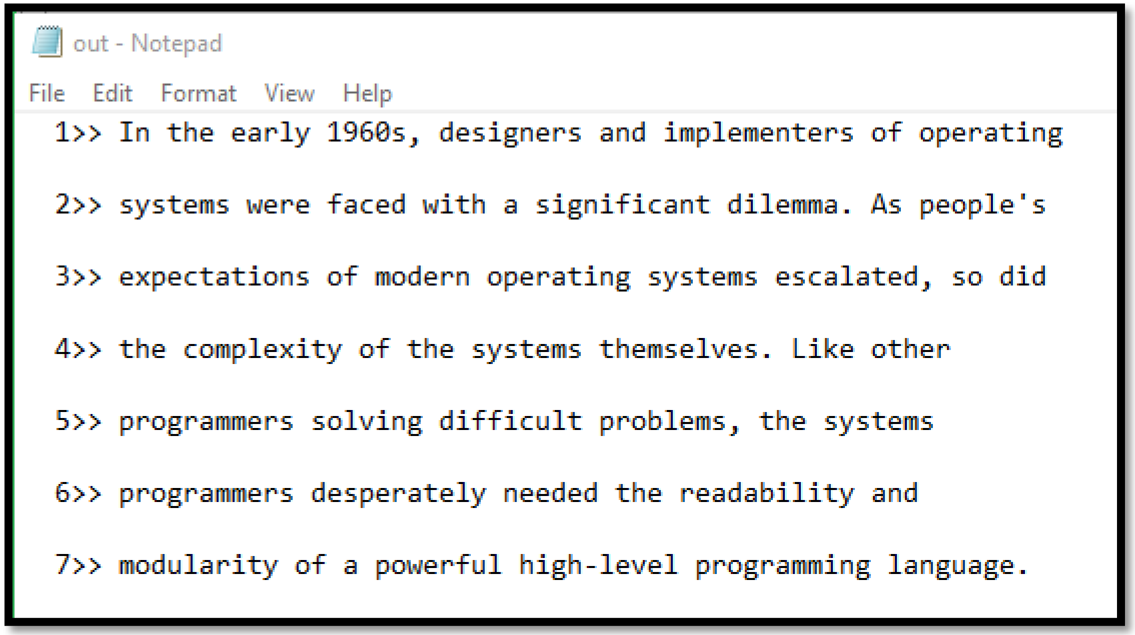


```
Name of input file> fgetsfileread.txt
Name of output file> out.txt

Process returned 0 (0x0)   execution time : 16.422 s
Press any key to continue.
```

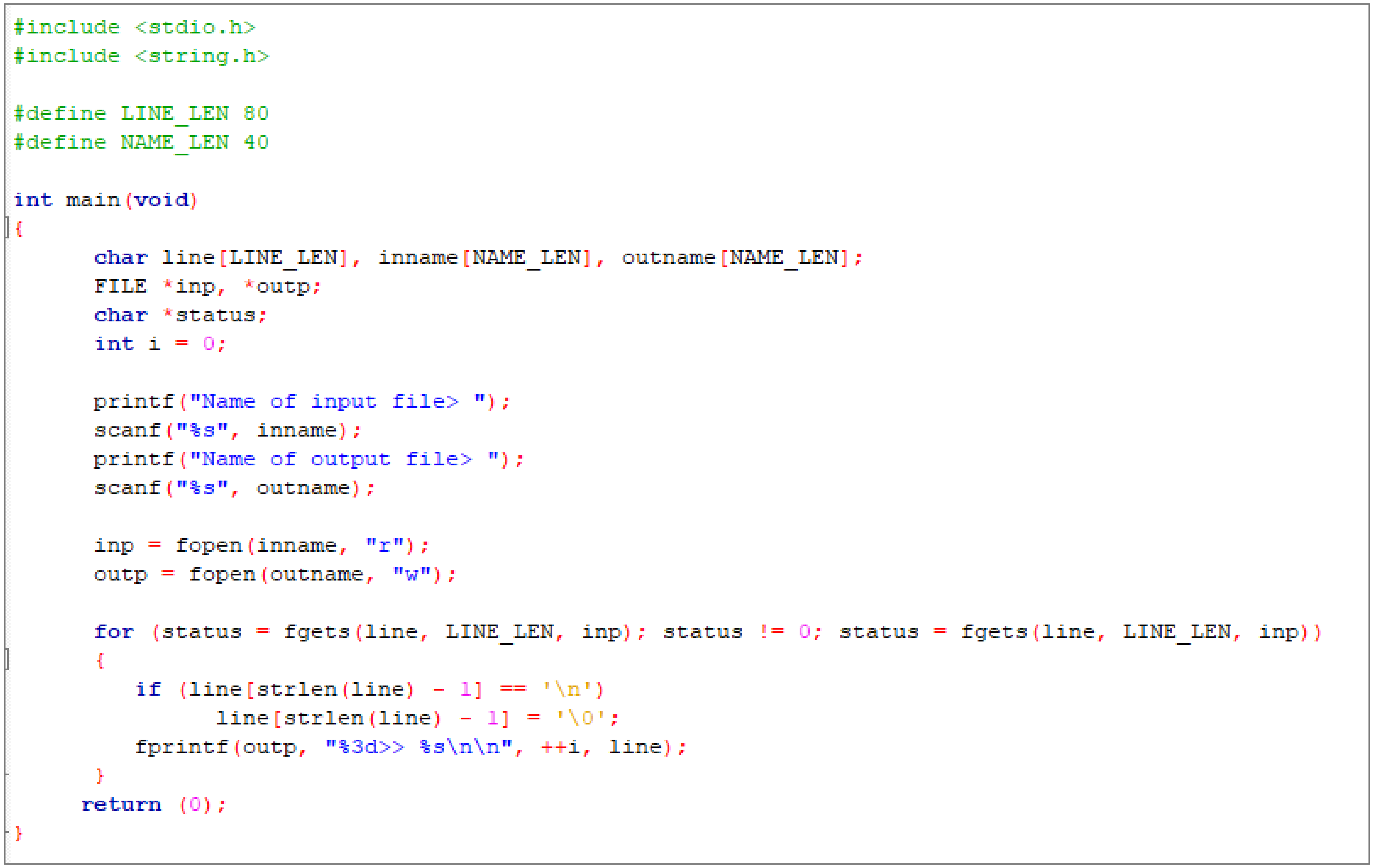
Get Line From File: fgets



```
In the early 1960s, designers and implementers of operating
systems were faced with a significant dilemma. As people's
expectations of modern operating systems escalated, so did
the complexity of the systems themselves. Like other
programmers solving difficult problems, the systems
programmers desperately needed the readability and
modularity of a powerful high-level programming language.
```



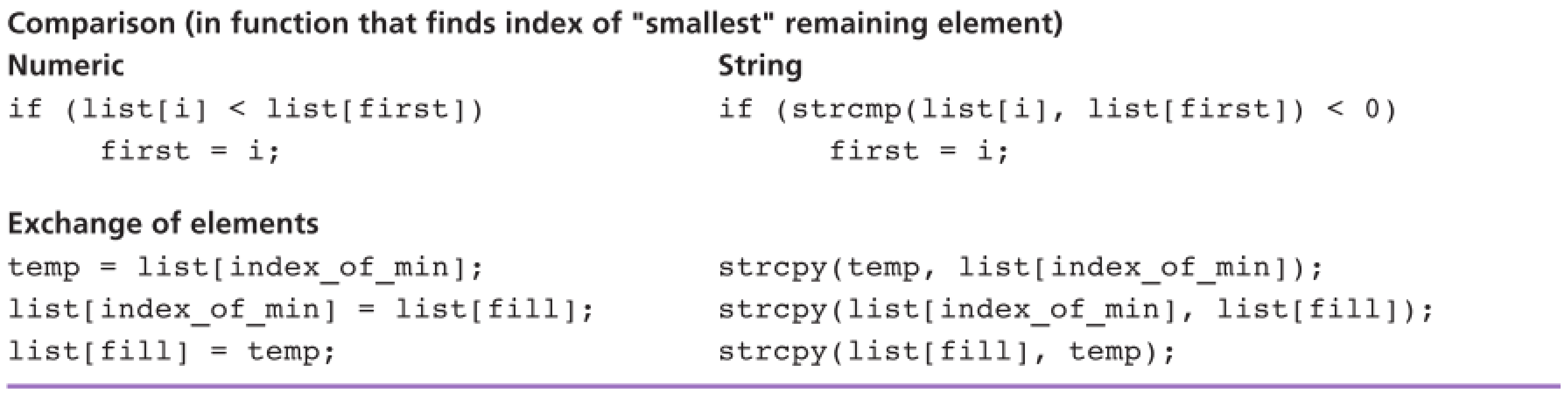
```
 1>> In the early 1960s, designers and implementers of operating

 2>> systems were faced with a significant dilemma. As people's

 3>> expectations of modern operating systems escalated, so did

 4>> the complexity of the systems themselves. Like other

 5>> programmers solving difficult problems, the systems

 6>> programmers desperately needed the readability and

 7>> modularity of a powerful high-level programming language.
```



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

#define LINE_LEN 80
#define NAME_LEN 40

int main(void)
{
      char line[LINE_LEN], inname[NAME_LEN], outname[NAME_LEN];
      FILE *inp, *outp;
      char *status;
      int i = 0;

      printf("Name of input file> ");
      scanf("%s", inname);
      printf("Name of output file> ");
      scanf("%s", outname);

      inp = fopen(inname, "r");
      outp = fopen(outname, "w");

      for (status = fgets(line, LINE_LEN, inp); status != 0; status = fgets(line, LINE_LEN, inp))
      {
         if (line[strlen(line) - 1] == '\n')
               line[strlen(line) - 1] = '\0';
         fprintf(outp, "%3d>> %s\n\n", ++i, line);
      }
     return (0);
}
```

HOA

- Write the string selection sort function



Comparison (in function that finds index of "smallest" remaining element)

Numeric

```
if (list[i] < list[first])
     first = i;
```

String

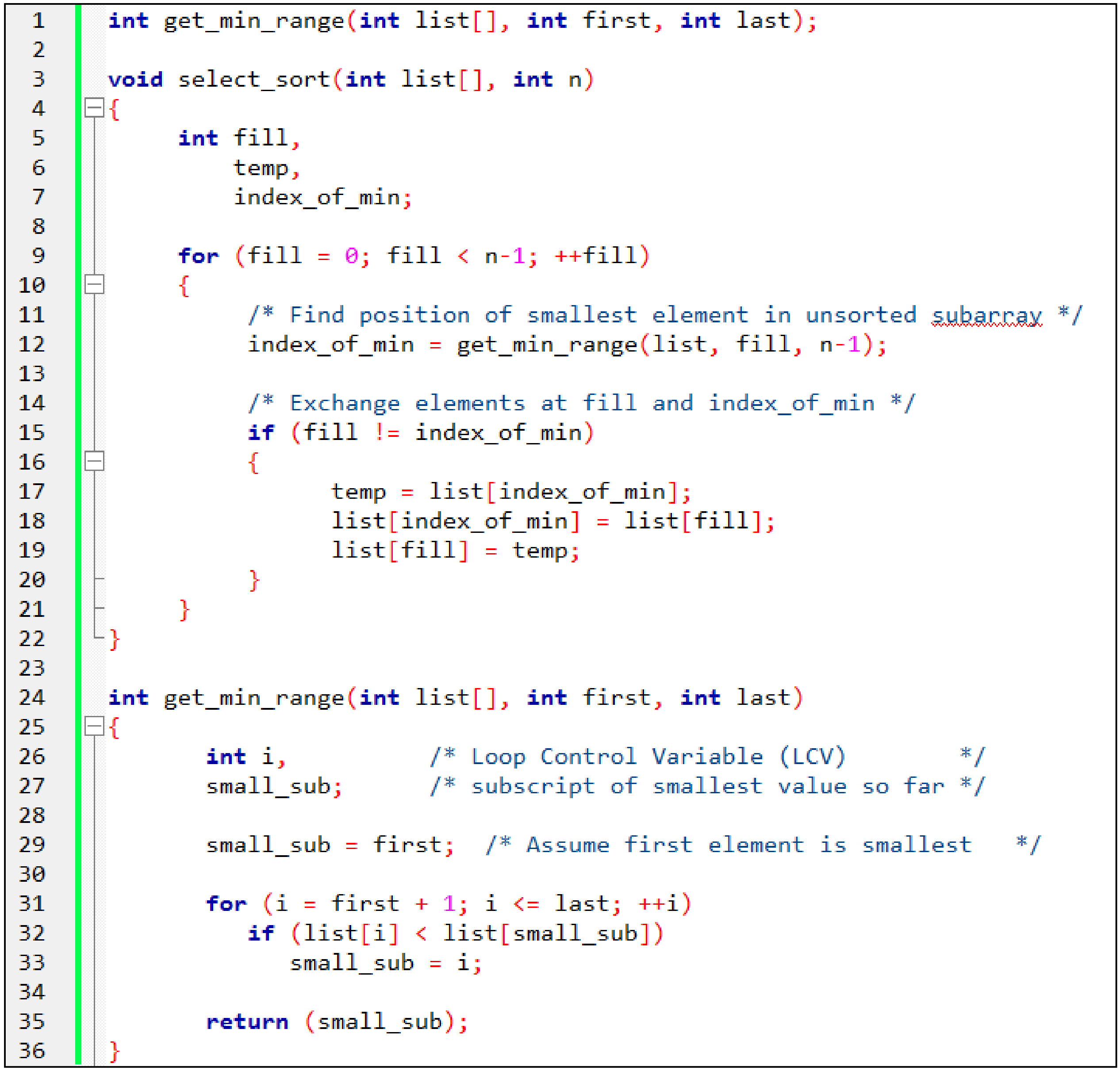
```
if (strcmp(list[i], list[first]) < 0)
      first = i;
```

Exchange of elements

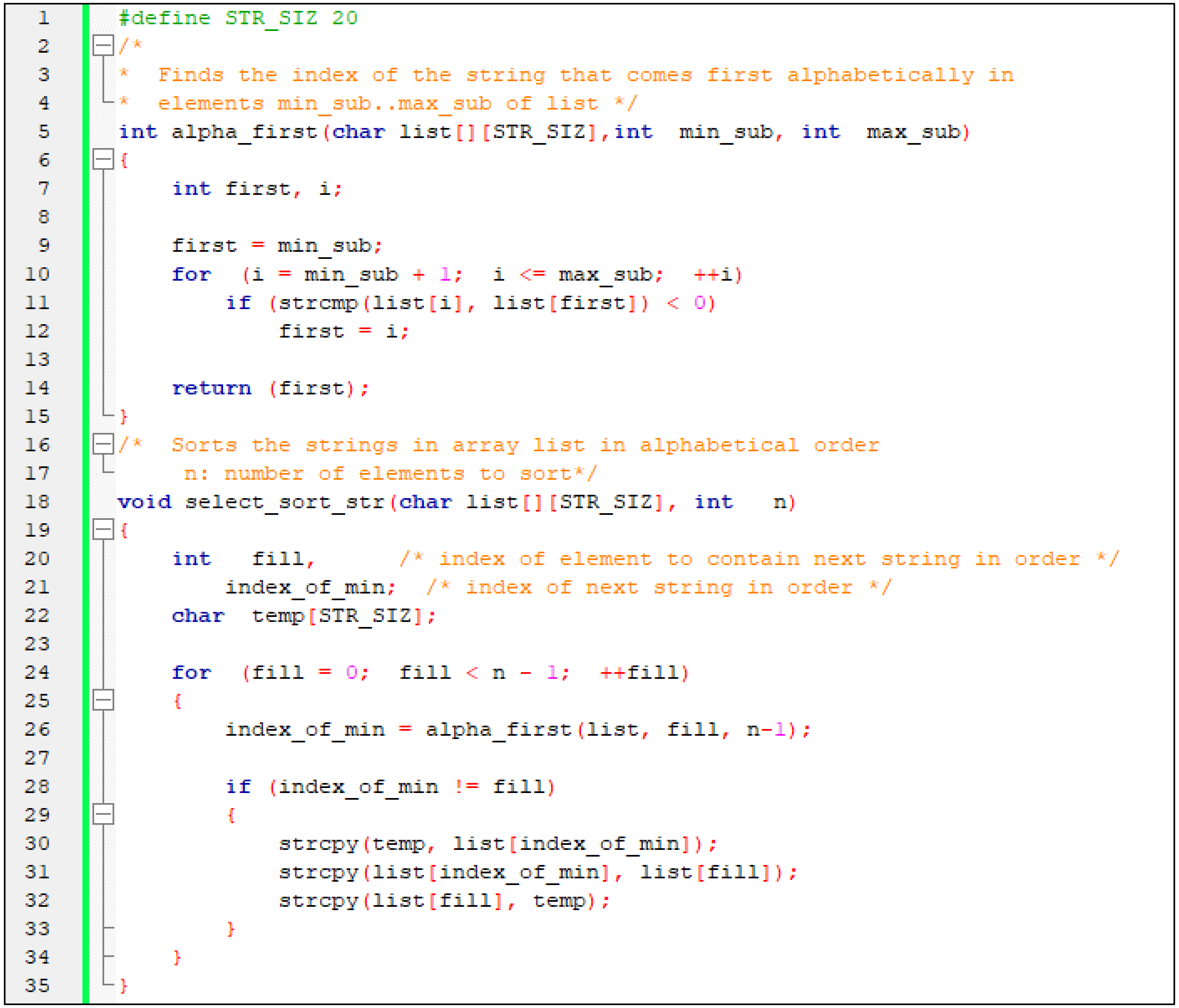
```
temp = list[index_of_min];
list[index_of_min] = list[fill];
list[fill] = temp;
```

```
strcpy(temp, list[index_of_min]);
strcpy(list[index_of_min], list[fill]);
strcpy(list[fill], temp);
```

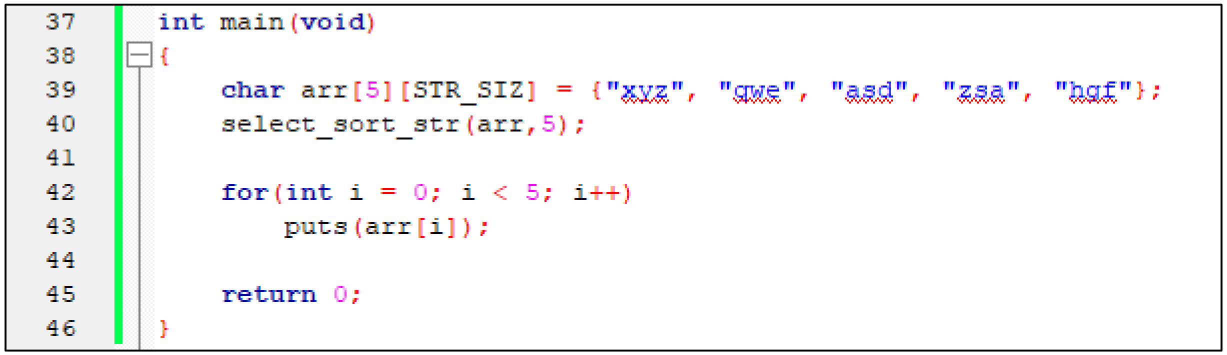
Numeric Version Reminder



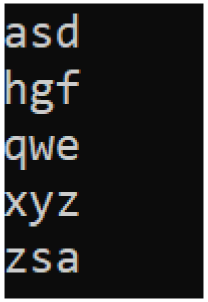
```c
int get_min_range(int list[], int first, int last);

void select_sort(int list[], int n)
{
     int fill,
         temp,
         index_of_min;

     for (fill = 0; fill < n-1; ++fill)
     {
          /* Find position of smallest element in unsorted subarray */
          index_of_min = get_min_range(list, fill, n-1);

          /* Exchange elements at fill and index_of_min */
          if (fill != index_of_min)
          {
                temp = list[index_of_min];
                list[index_of_min] = list[fill];
                list[fill] = temp;
          }
     }
}

int get_min_range(int list[], int first, int last)
{
       int i,          /* Loop Control Variable (LCV)        */
       small_sub;      /* subscript of smallest value so far */

       small_sub = first;  /* Assume first element is smallest   */

       for (i = first + 1; i <= last; ++i)
          if (list[i] < list[small_sub])
             small_sub = i;

       return (small_sub);
}
```



```
#define STR_SIZ 20
/*
 *  Finds the index of the string that comes first alphabetically in
 *  elements min_sub..max_sub of list */
int alpha_first(char list[][STR_SIZ],int  min_sub, int  max_sub)
{
    int first, i;

    first = min_sub;
    for  (i = min_sub + 1;  i <= max_sub;  ++i)
        if (strcmp(list[i], list[first]) < 0)
            first = i;

    return (first);
}
/*  Sorts the strings in array list in alphabetical order
     n: number of elements to sort*/
void select_sort_str(char list[][STR_SIZ], int   n)
{
    int   fill,      /* index of element to contain next string in order */
        index_of_min;  /* index of next string in order */
    char  temp[STR_SIZ];

    for  (fill = 0;  fill < n - 1;  ++fill)
    {
        index_of_min = alpha_first(list, fill, n-1);

        if (index_of_min != fill)
        {
            strcpy(temp, list[index_of_min]);
            strcpy(list[index_of_min], list[fill]);
            strcpy(list[fill], temp);
        }
    }
}
```



```c
int main(void)
{
    char arr[5][STR_SIZ] = {"xyz", "qwe", "asd", "zsa", "hgf"};
    select_sort_str(arr,5);

    for(int i = 0; i < 5; i++)
        puts(arr[i]);

    return 0;
}
```



```
asd
hgf
qwe
xyz
zsa
```

Sentinel Controlled Loop

- If we do not know how much data will be entered, SENTINEL is a good choice to use

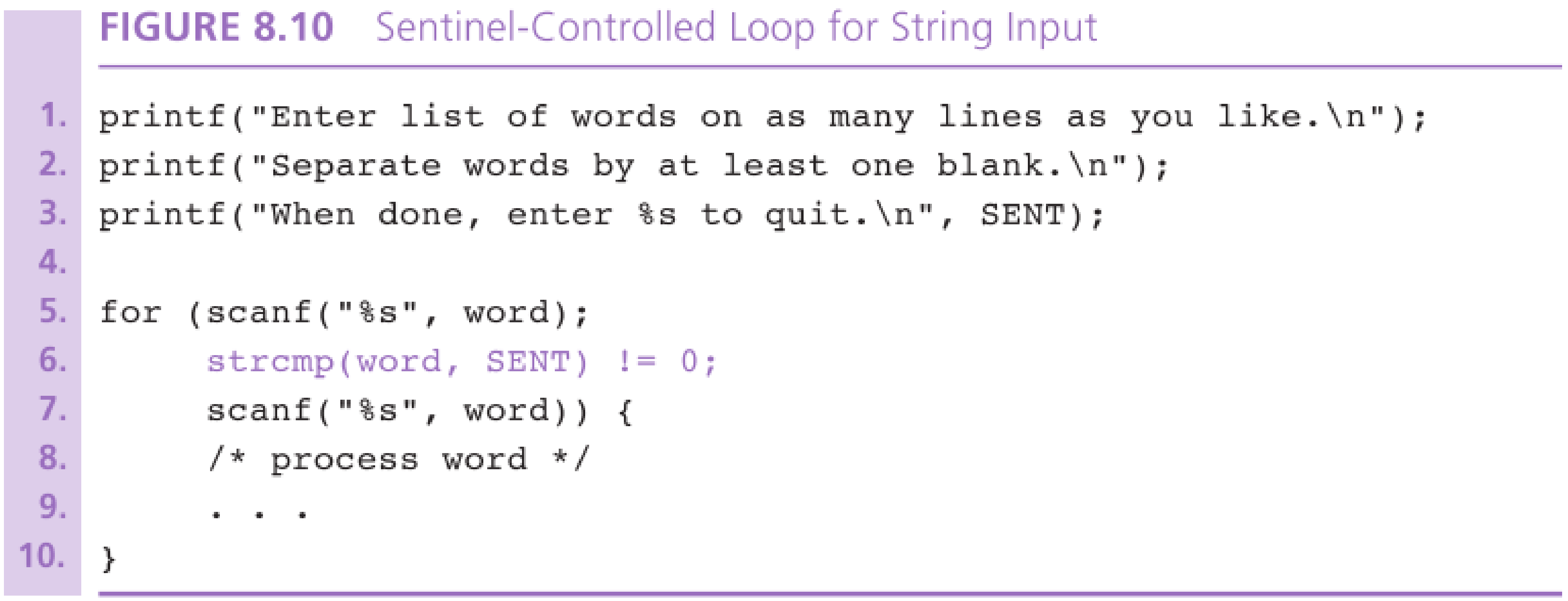


FIGURE 8.10 Sentinel-Controlled Loop for String Input

```
printf("Enter list of words on as many lines as you like.\n");
printf("Separate words by at least one blank.\n");
printf("When done, enter %s to quit.\n", SENT);

for (scanf("%s", word);
     strcmp(word, SENT) != 0;
     scanf("%s", word)) {
     /* process word */
     . . .
}
```

Arrays of Pointers

- When sorting a list of strings, there is <u>a lot of copying</u> of characters from one memory cell to another.
  - **3 operations for every exchange**

```
    strcpy(temp, list[index_of_min]);
    strcpy(list[index_of_min], list[fill]);
    strcpy(list[fill], temp);
```

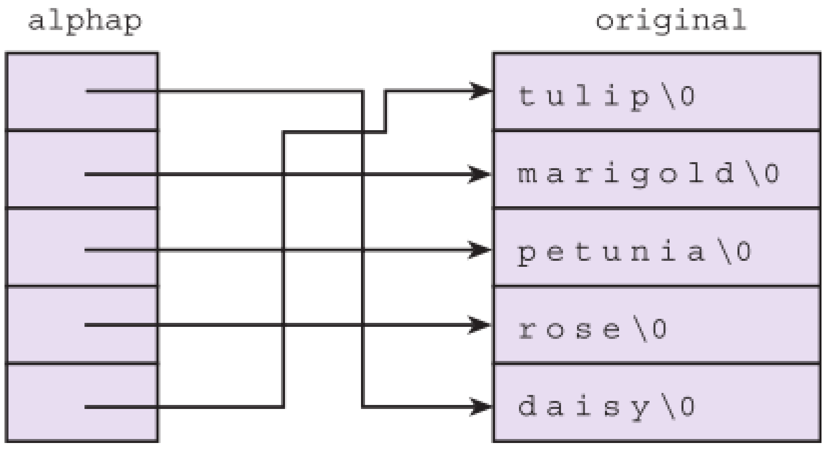
list[any_index] is actually an array
of characters, therefore it is passed
to a function as pointer: address of list[0]

  - Original list is lost since arrays are passed by their ADDRESS automatically
- C represents every array by its starting address.

Arrays of Pointers

- Alternative sorting:
  - Consider an array of pointers, each element the address of a character string.

alphab[0] address of "daisy"
alphab[1] address of "marigold"
alphab[2] address of "petunia"
alphab[3] address of "rose"
alphab[4] address of "tulip"



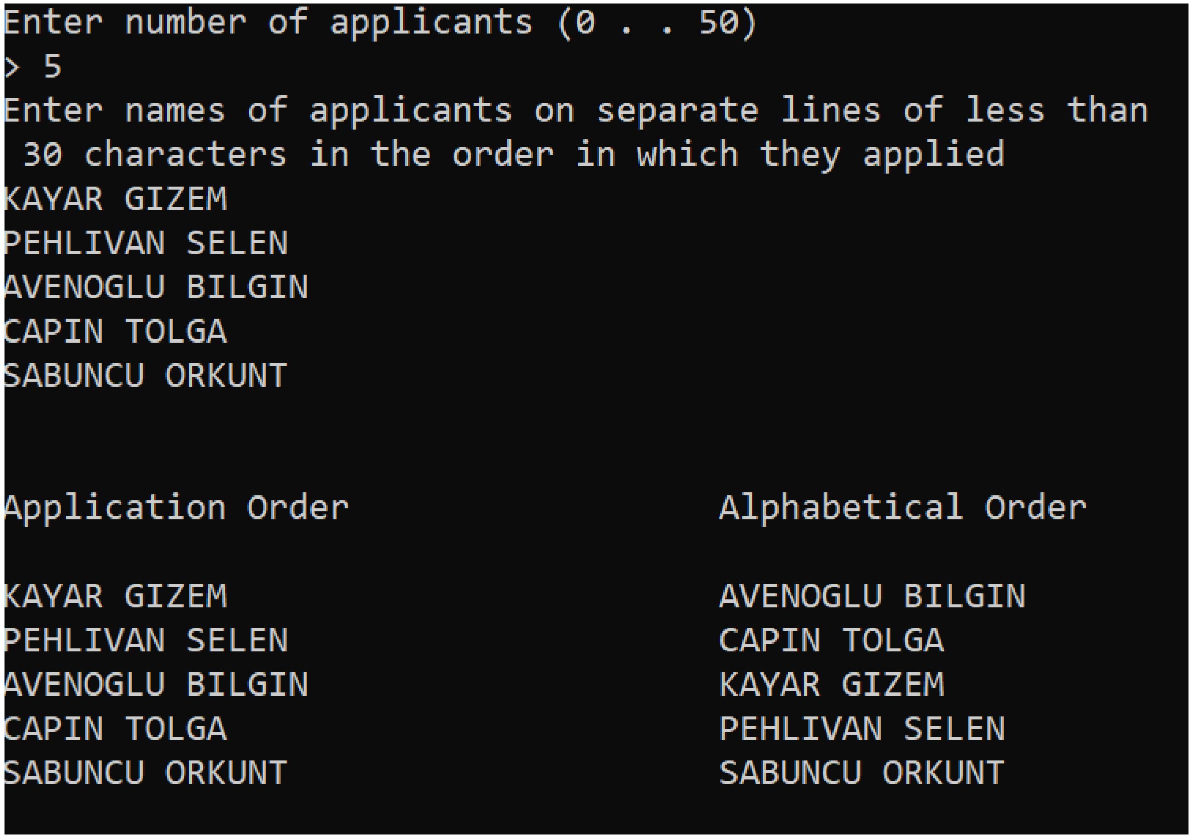
```
for(int i = 0; i < 5; i++)
    puts(alphab[i]);
```

Declaration:
char* alphab[5];

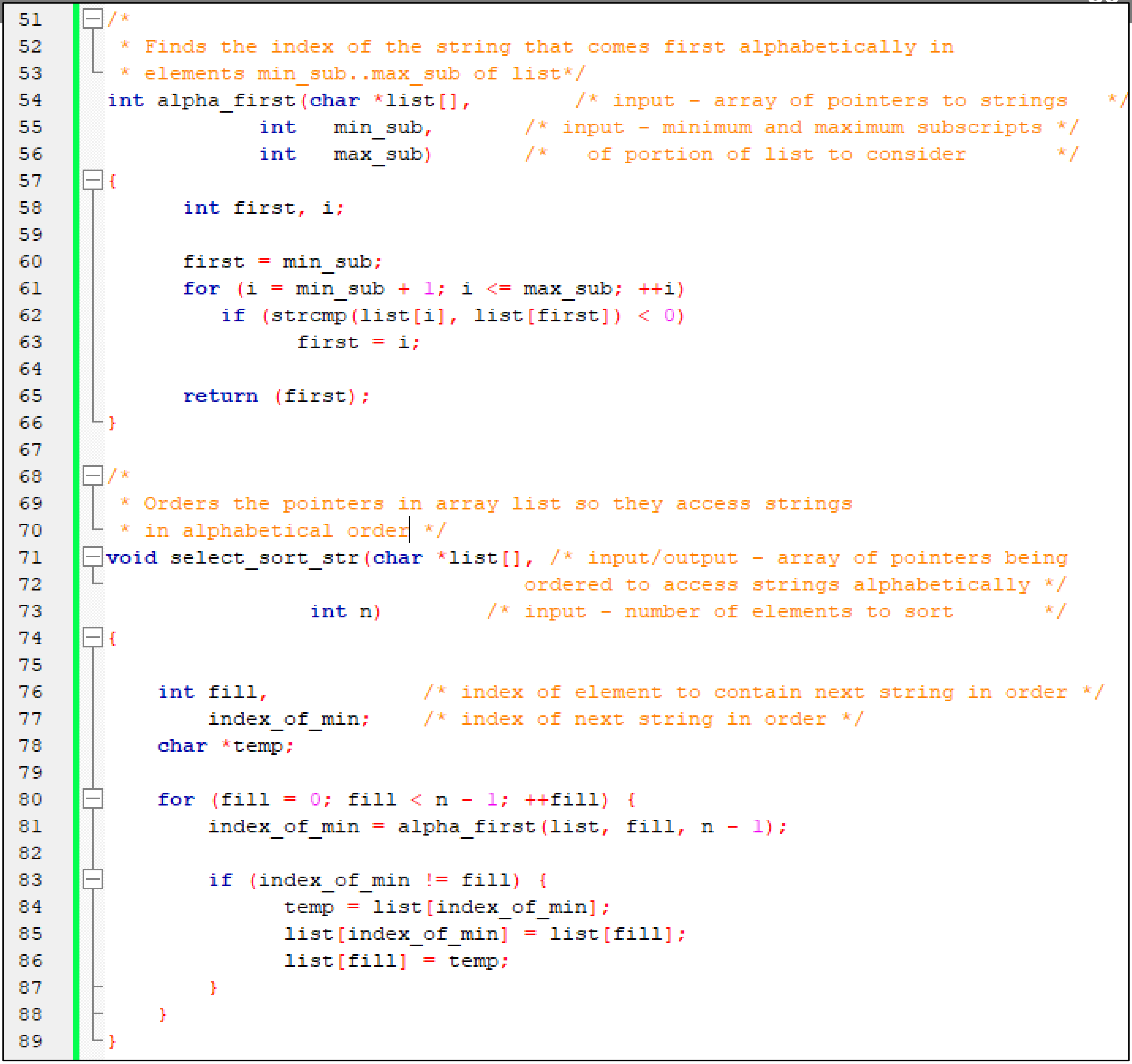
prints *original* in alphabetical order. *original* is **not lost.**

HOA

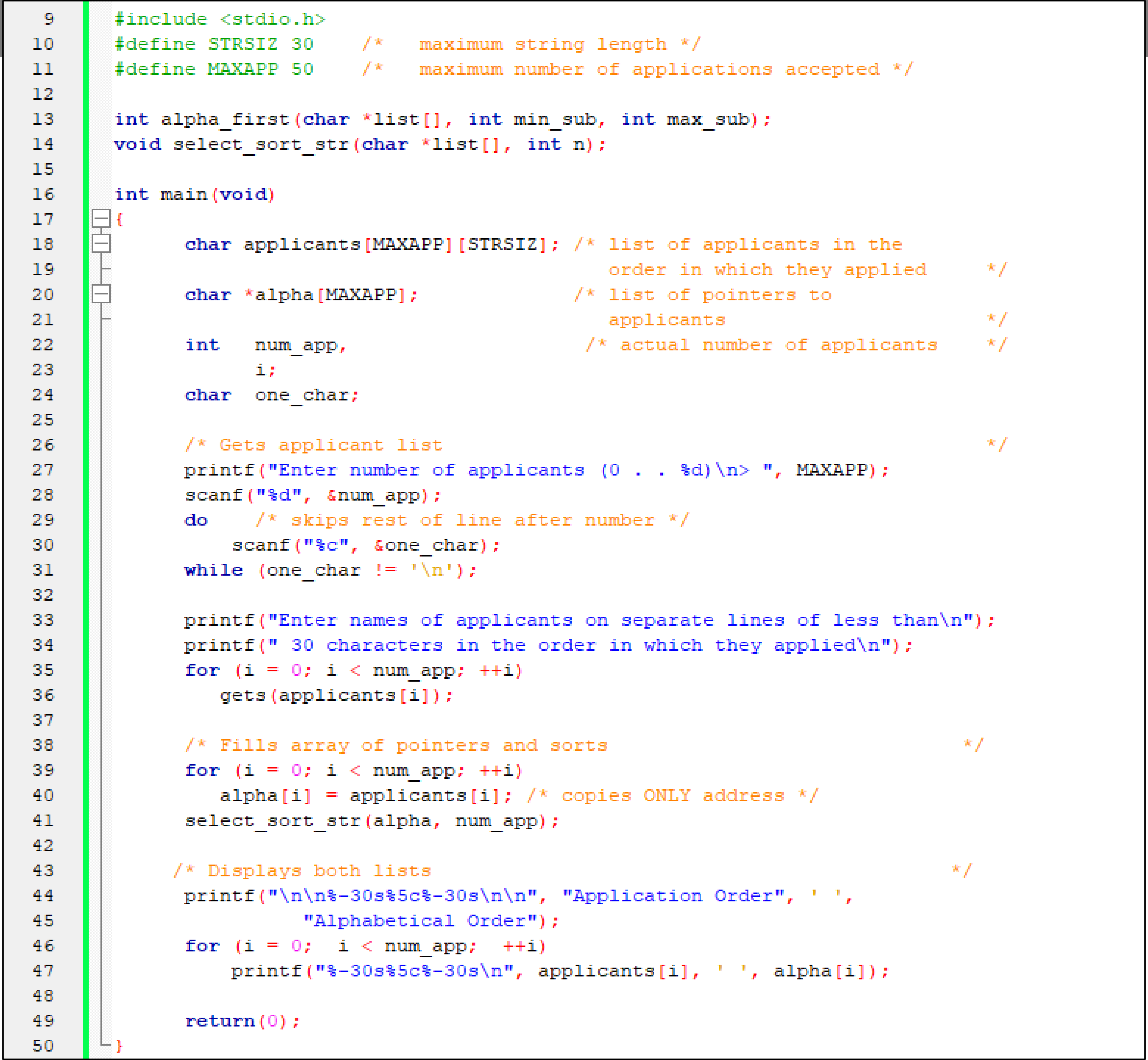
- Order the name of applicants to a school as you also keep the original list



```
Enter number of applicants (0 . . 50)
> 5
Enter names of applicants on separate lines of less than
 30 characters in the order in which they applied
KAYAR GIZEM
PEHLIVAN SELEN
AVENOGLU BILGIN
CAPIN TOLGA
SABUNCU ORKUNT


Application Order                  Alphabetical Order

KAYAR GIZEM                        AVENOGLU BILGIN
PEHLIVAN SELEN                     CAPIN TOLGA
AVENOGLU BILGIN                    KAYAR GIZEM
CAPIN TOLGA                        PEHLIVAN SELEN
SABUNCU ORKUNT                     SABUNCU ORKUNT
```



```
/*
 * Finds the index of the string that comes first alphabetically in
 * elements min_sub..max_sub of list*/
int alpha_first(char *list[],        /* input - array of pointers to strings   */
            int   min_sub,       /* input - minimum and maximum subscripts */
            int   max_sub)       /*   of portion of list to consider       */
{
      int first, i;

      first = min_sub;
      for (i = min_sub + 1; i <= max_sub; ++i)
         if (strcmp(list[i], list[first]) < 0)
               first = i;

      return (first);
}

/*
 * Orders the pointers in array list so they access strings
 * in alphabetical order */
void select_sort_str(char *list[], /* input/output - array of pointers being
                                 ordered to access strings alphabetically */
                int n)        /* input - number of elements to sort       */
{

    int fill,               /* index of element to contain next string in order */
        index_of_min;       /* index of next string in order */
    char *temp;

    for (fill = 0; fill < n - 1; ++fill) {
        index_of_min = alpha_first(list, fill, n - 1);

        if (index_of_min != fill) {
              temp = list[index_of_min];
              list[index_of_min] = list[fill];
              list[fill] = temp;
        }
    }
}
```



```c
#include <stdio.h>
#define STRSIZ 30    /*   maximum string length */
#define MAXAPP 50    /*   maximum number of applications accepted */

int alpha_first(char *list[], int min_sub, int max_sub);
void select_sort_str(char *list[], int n);

int main(void)
{
      char applicants[MAXAPP][STRSIZ]; /* list of applicants in the
                                          order in which they applied     */
      char *alpha[MAXAPP];             /* list of pointers to
                                          applicants                      */
      int   num_app,                    /* actual number of applicants    */
            i;
      char  one_char;

      /* Gets applicant list                                               */
      printf("Enter number of applicants (0 . . %d)\n> ", MAXAPP);
      scanf("%d", &num_app);
      do    /* skips rest of line after number */
          scanf("%c", &one_char);
      while (one_char != '\n');

      printf("Enter names of applicants on separate lines of less than\n");
      printf(" 30 characters in the order in which they applied\n");
      for (i = 0; i < num_app; ++i)
         gets(applicants[i]);

      /* Fills array of pointers and sorts                               */
      for (i = 0; i < num_app; ++i)
         alpha[i] = applicants[i]; /* copies ONLY address */
      select_sort_str(alpha, num_app);

     /* Displays both lists                                              */
      printf("\n\n%-30s%5c%-30s\n\n", "Application Order", ' ',
                "Alphabetical Order");
      for (i = 0;  i < num_app;  ++i)
          printf("%-30s%5c%-30s\n", applicants[i], ' ', alpha[i]);

      return(0);
}
```

Advantages

- A pointer requires less storage space than a full copy of character string
- Sorting array of pointers by copying them is faster than copying complete array of characters
- Any spelling correction made in the original list will be reflected in other orderings

Character Input/Output

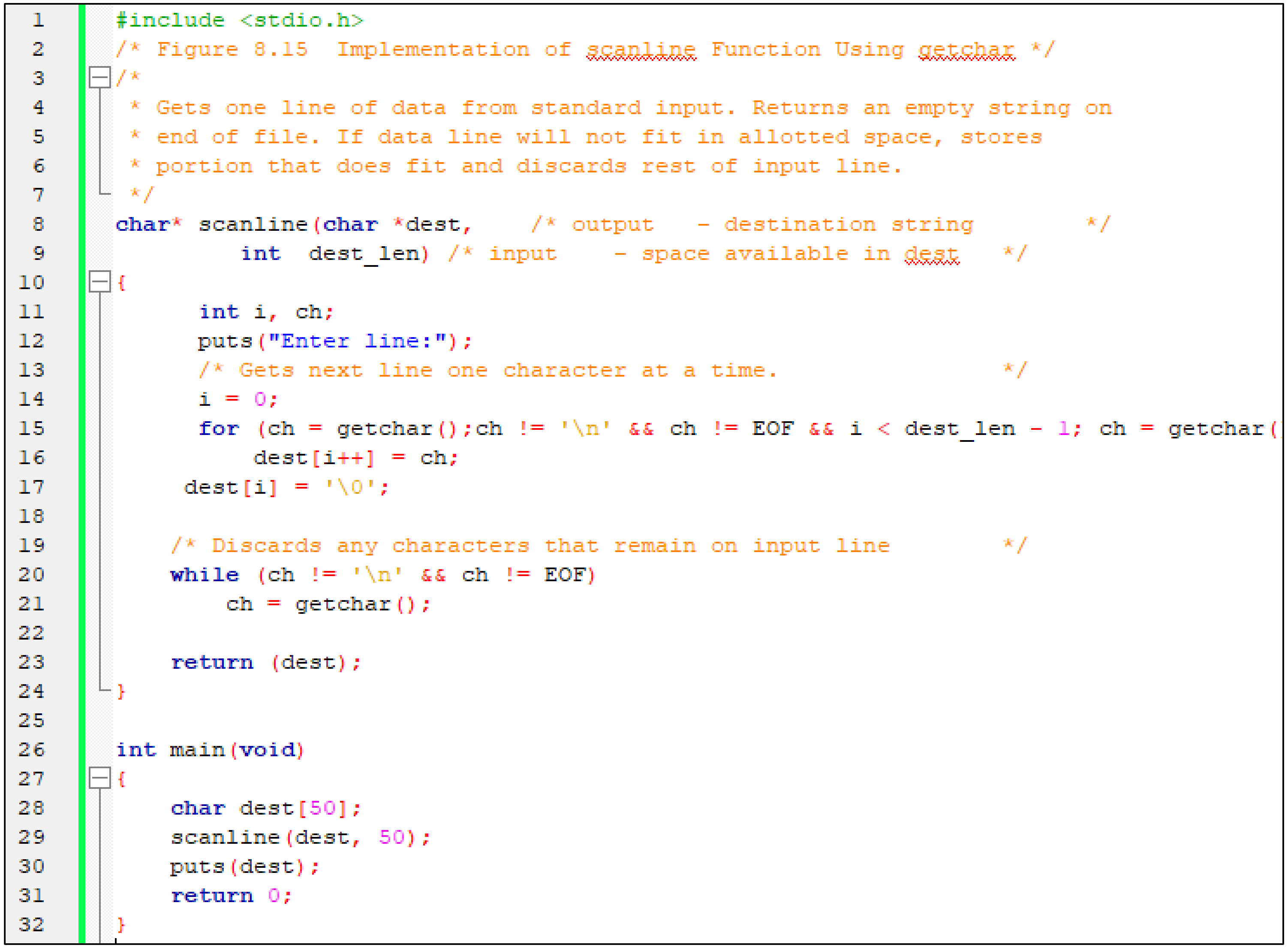
- getchar
  - get the next character from the standard input source (that scanf uses)
  - does not expect the calling module to pass the address of a variable to store the input character
  - takes no arguments, returns the character as its result

```
ch = getchar()
```

defined in <stdio.h>

HOA

- Write a scanline function which scans a line using getchar



```c
#include <stdio.h>
/* Figure 8.15  Implementation of scanline Function Using getchar */
/*
 * Gets one line of data from standard input. Returns an empty string on
 * end of file. If data line will not fit in allotted space, stores
 * portion that does fit and discards rest of input line.
 */
char* scanline(char *dest,    /* output   - destination string        */
         int  dest_len) /* input     - space available in dest    */
{
      int i, ch;
      puts("Enter line:");
      /* Gets next line one character at a time.                  */
      i = 0;
      for (ch = getchar();ch != '\n' && ch != EOF && i < dest_len - 1; ch = getchar(
          dest[i++] = ch;
     dest[i] = '\0';

    /* Discards any characters that remain on input line         */
    while (ch != '\n' && ch != EOF)
        ch = getchar();

    return (dest);
}

int main(void)
{
    char dest[50];
    scanline(dest, 50);
    puts(dest);
    return 0;
}
```

Character Input/Output

- getc
  - used to get a single character from a file
  - comparable to getchar except that the character returned is obtained from the file accessed by a file pointer (ex., inp)

```
getc(inp)
```

defined in <stdio.h>

Character Input/Output

- putchar
  - single-character output
  - first argument is a type int character code
  - recall that type char can always be converted to type in with no loss of information

```
putchar('a');
```

defined in <stdio.h>

Character Input/Output

- putc
  - identical to putchar except it sends the single character/int to a file, ex., outp

```
putc('a', outp);
```

defined in <stdio.h>

ctype.h

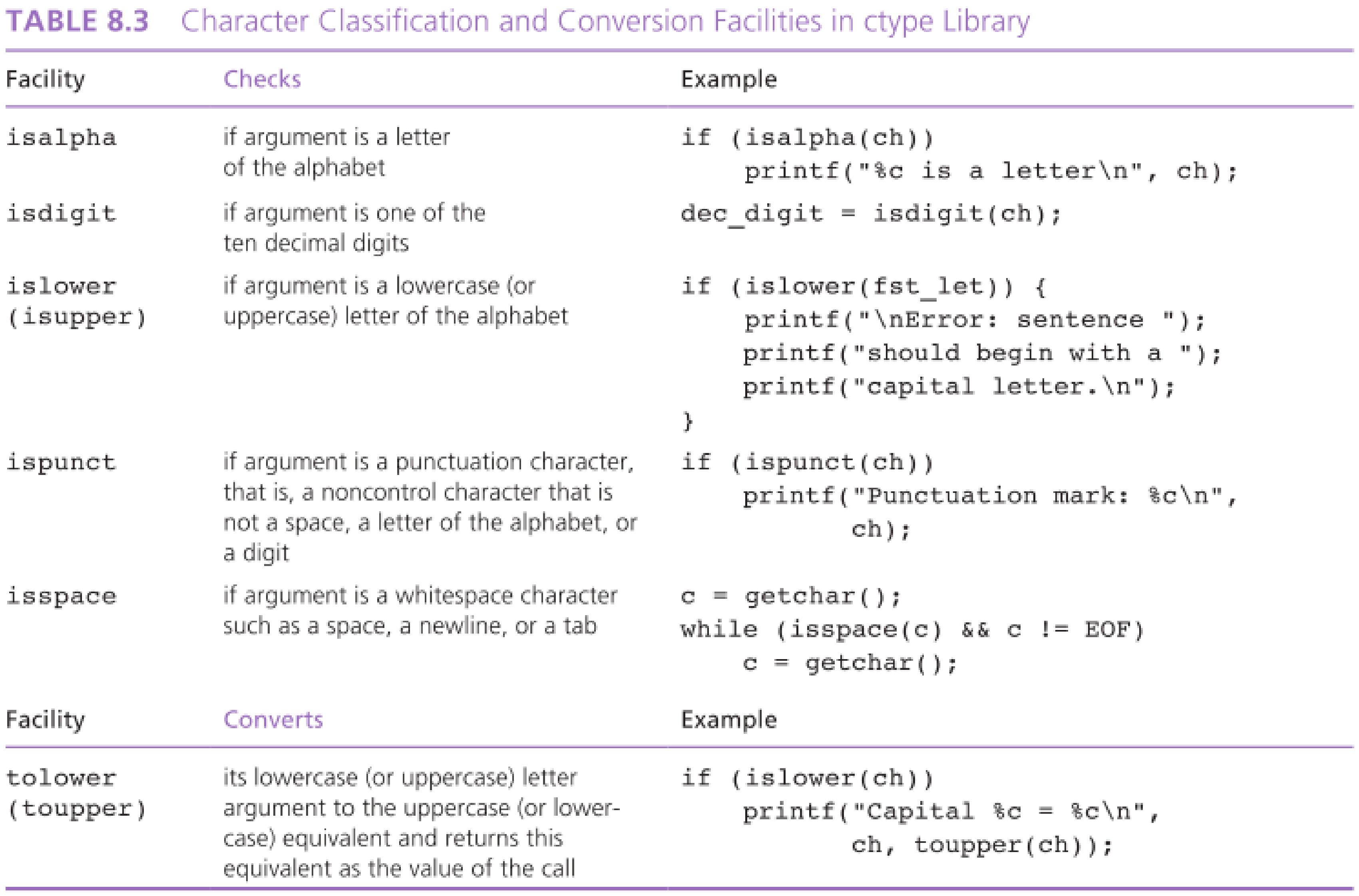


TABLE 8.3 Character Classification and Conversion Facilities in ctype Library

Facility	Checks	Example

`isalpha`	if argument is a letter of the alphabet	`if (isalpha(ch))`
		`    printf("%c is a letter\n", ch);`

`isdigit`	if argument is one of the ten decimal digits	`dec_digit = isdigit(ch);`

`islower` (`isupper`)	if argument is a lowercase (or uppercase) letter of the alphabet	`if (islower(fst_let)) {`
		`    printf("\nError: sentence ");`
		`    printf("should begin with a ");`
		`    printf("capital letter.\n");`
		`}`

`ispunct`	if argument is a punctuation character, that is, a noncontrol character that is not a space, a letter of the alphabet, or a digit	`if (ispunct(ch))`
		`    printf("Punctuation mark: %c\n",`
		`           ch);`

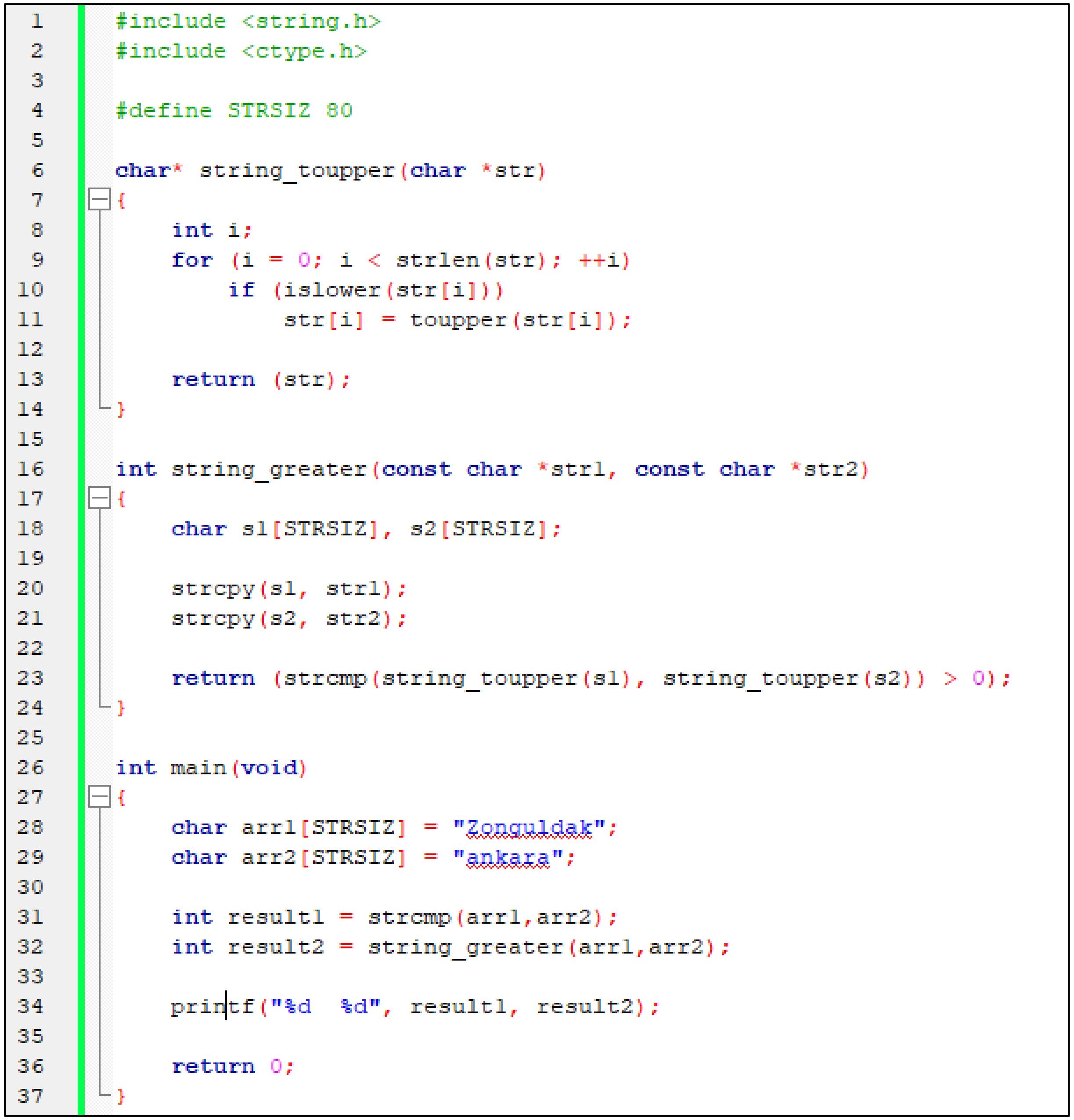
`isspace`	if argument is a whitespace character such as a space, a newline, or a tab	`c = getchar();`
		`while (isspace(c) && c != EOF)`
		`    c = getchar();`

Facility	**Converts**	**Example**

`tolower` (`toupper`)	its lowercase (or uppercase) letter argument to the uppercase (or lower-case) equivalent and returns this equivalent as the value of the call	`if (islower(ch))`
		`    printf("Capital %c = %c\n",`
		`           ch, toupper(ch));`

Example – Upper/Lower Cases

- What is the problem with strcmp("Zen","asd")?
  - Returns negative even Z comes after a alphabetically due to ASCII character codes
  - Capital letters come first!!
- What to do?
  - Convert all strings to upper or lower case
  - toupper function modifies the original therefore keep a copy of the original



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

#define STRSIZ 80

char* string_toupper(char *str)
{
    int i;
    for (i = 0; i < strlen(str); ++i)
        if (islower(str[i]))
            str[i] = toupper(str[i]);

    return (str);
}

int string_greater(const char *str1, const char *str2)
{
    char s1[STRSIZ], s2[STRSIZ];

    strcpy(s1, str1);
    strcpy(s2, str2);

    return (strcmp(string_toupper(s1), string_toupper(s2)) > 0);
}

int main(void)
{
    char arr1[STRSIZ] = "Zonguldak";
    char arr2[STRSIZ] = "ankara";

    int result1 = strcmp(arr1,arr2);
    int result2 = string_greater(arr1,arr2);

    printf("%d  %d", result1, result2);

    return 0;
}
```

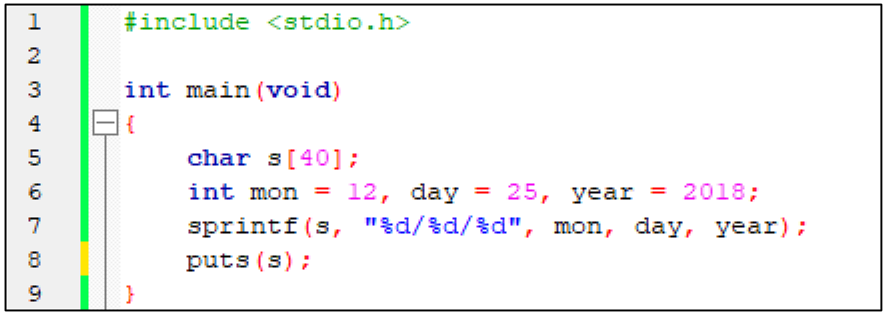
Output:

```
-1   1
```

sprintf

defined in <stdio.h>

- **printf**("format", args) is used to print the data onto the standard output, e.g. computer monitor.
- **fprintf**(FILE *fp, "format", args) is like printf however, instead of displaying the data on the monitor, the formated data is saved on a file which is pointed to by the file pointer.
- **sprintf**(char *, "format", args) is like printf. Instead of displaying the formated string on the standard output, it stores the formated data in a string pointed to by the char pointer (the very first parameter).
  - Risk of overflowing destination string

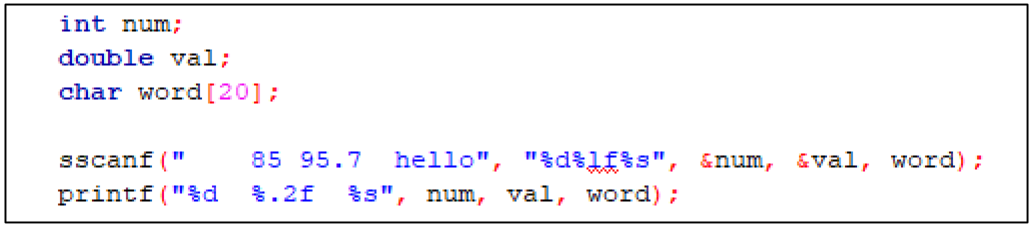


```c
#include <stdio.h>

int main(void)
{
    char s[40];
    int mon = 12, day = 25, year = 2018;
    sprintf(s, "%d/%d/%d", mon, day, year);
    puts(s);
}
```

Output:
12/25/2018

sscanf

- Similar to scanf and sprintf
- Does not scan from the input device

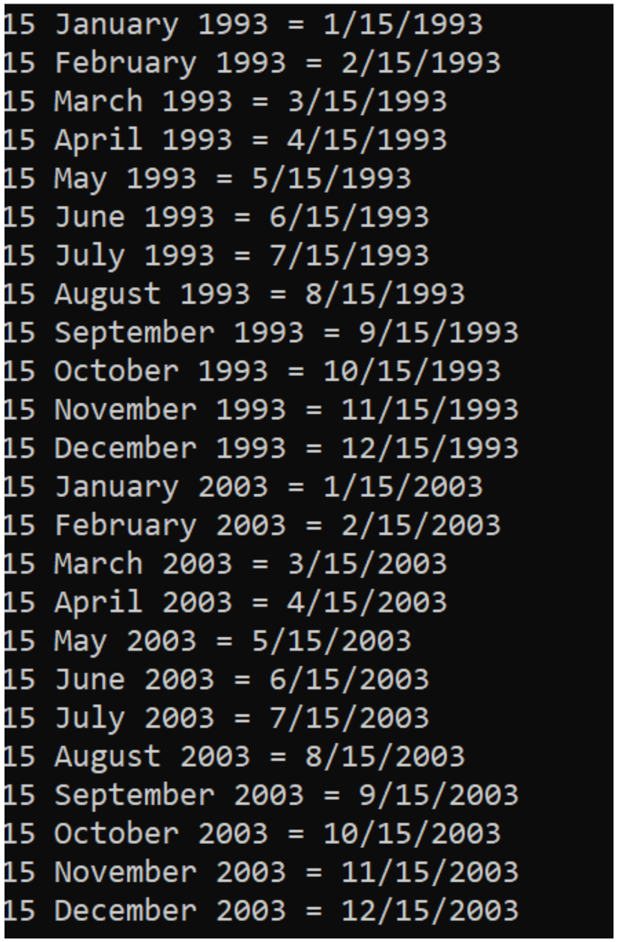


```
int num;
double val;
char word[20];

sscanf("    85 95.7  hello", "%d%lf%s", &num, &val, word);
printf("%d  %.2f  %s", num, val, word);
```

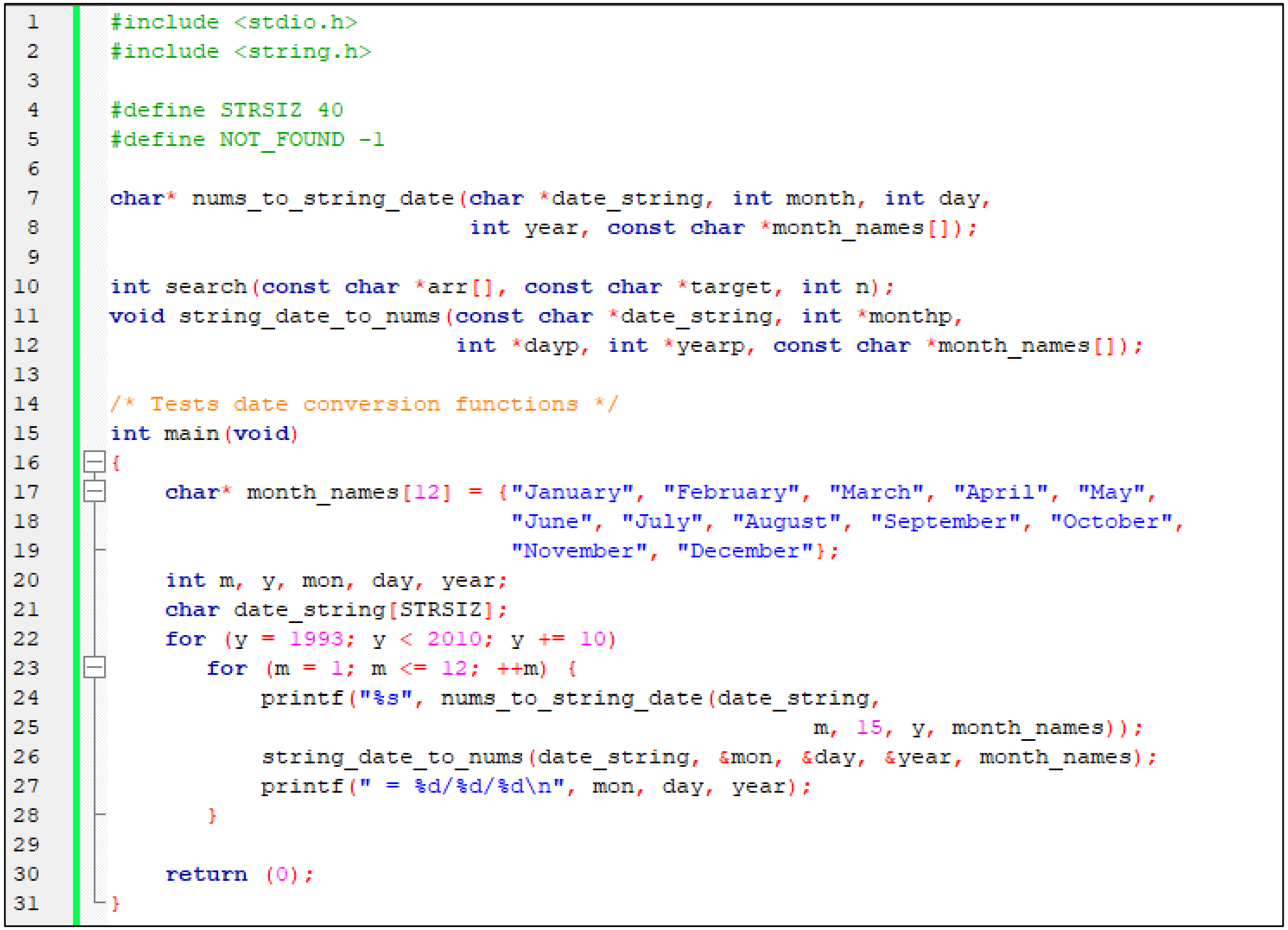
Output:
85 95.70 hello

Example

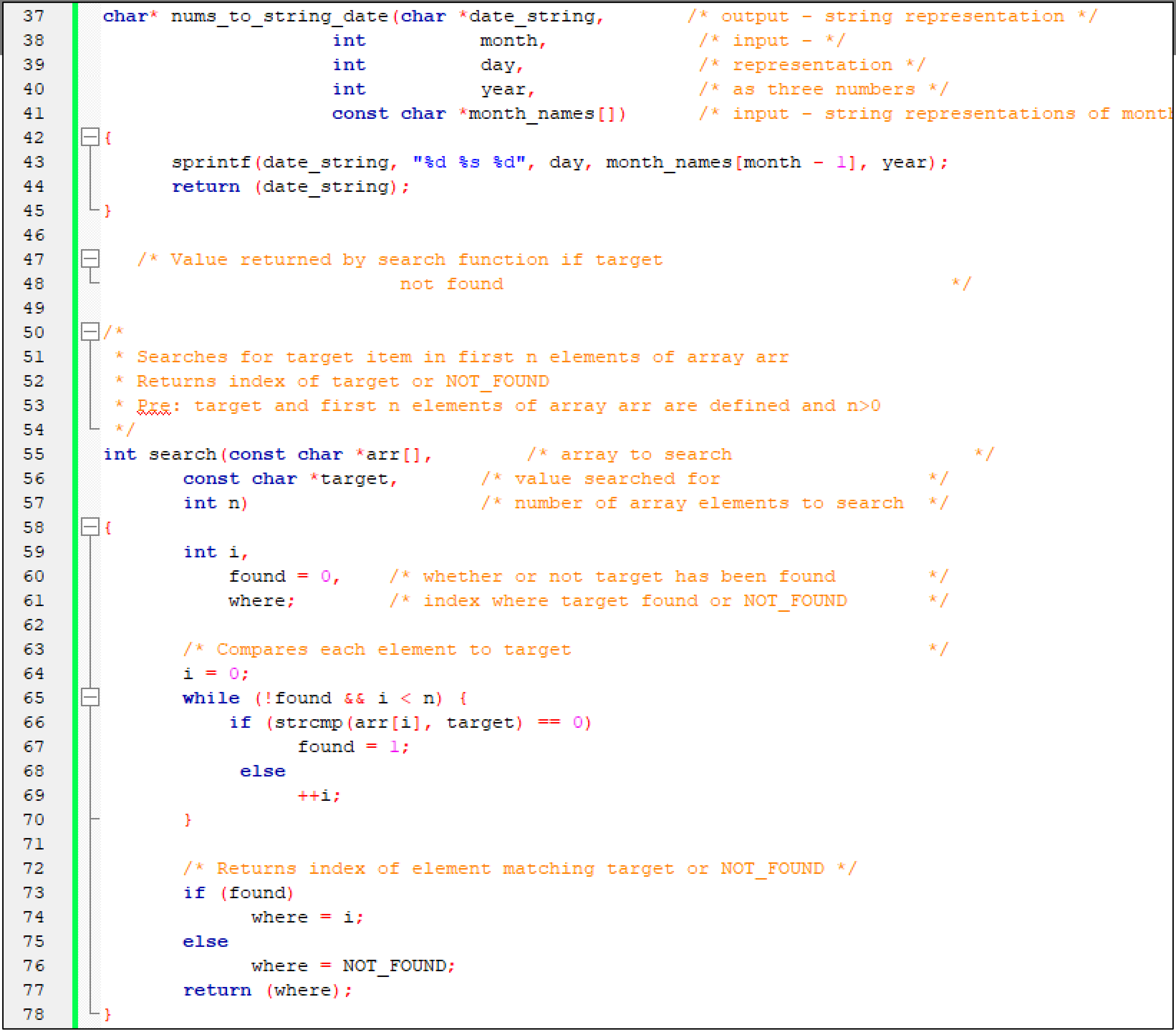
- Write a function which gives the below output:



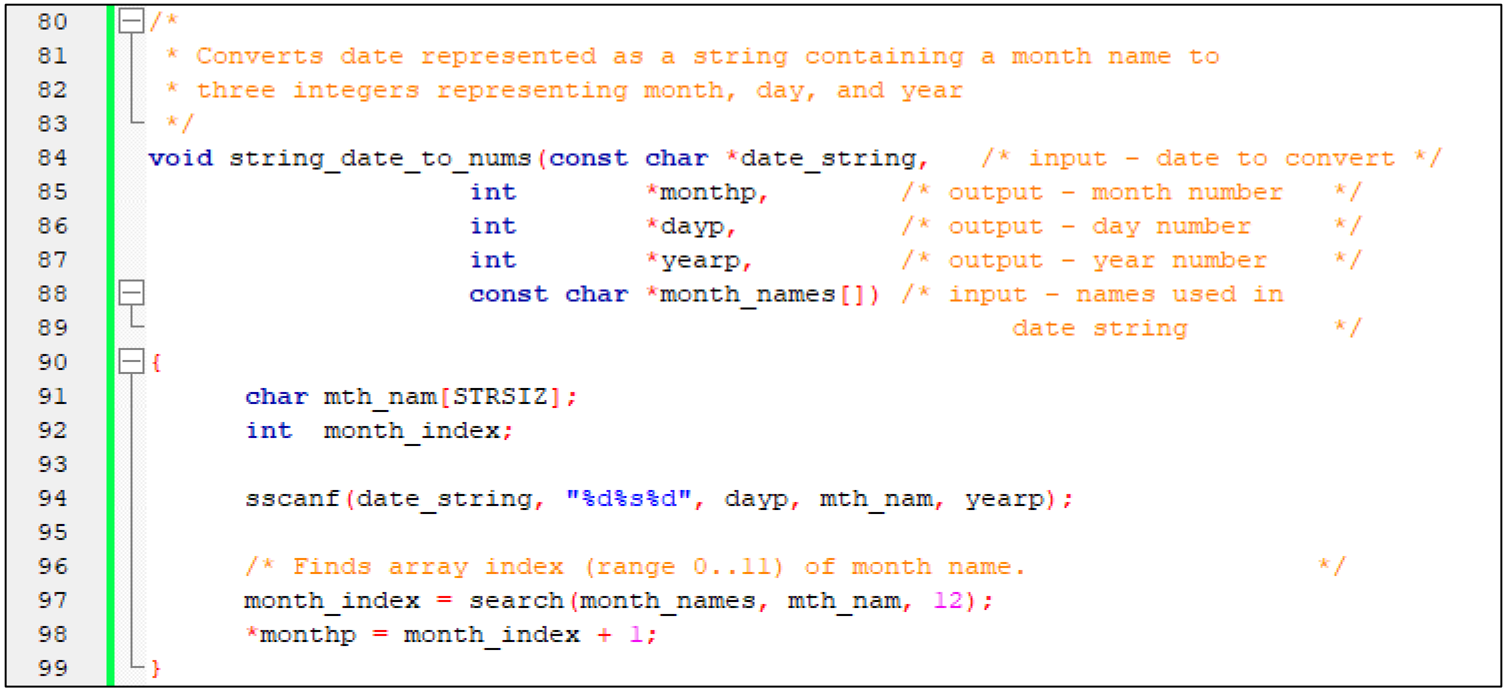
```
15 January 1993 = 1/15/1993
15 February 1993 = 2/15/1993
15 March 1993 = 3/15/1993
15 April 1993 = 4/15/1993
15 May 1993 = 5/15/1993
15 June 1993 = 6/15/1993
15 July 1993 = 7/15/1993
15 August 1993 = 8/15/1993
15 September 1993 = 9/15/1993
15 October 1993 = 10/15/1993
15 November 1993 = 11/15/1993
15 December 1993 = 12/15/1993
15 January 2003 = 1/15/2003
15 February 2003 = 2/15/2003
15 March 2003 = 3/15/2003
15 April 2003 = 4/15/2003
15 May 2003 = 5/15/2003
15 June 2003 = 6/15/2003
15 July 2003 = 7/15/2003
15 August 2003 = 8/15/2003
15 September 2003 = 9/15/2003
15 October 2003 = 10/15/2003
15 November 2003 = 11/15/2003
15 December 2003 = 12/15/2003
```



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

#define STRSIZ 40
#define NOT_FOUND -1

char* nums_to_string_date(char *date_string, int month, int day,
                          int year, const char *month_names[]);

int search(const char *arr[], const char *target, int n);
void string_date_to_nums(const char *date_string, int *monthp,
                         int *dayp, int *yearp, const char *month_names[]);

/* Tests date conversion functions */
int main(void)
{
    char* month_names[12] = {"January", "February", "March", "April", "May",
                             "June", "July", "August", "September", "October",
                             "November", "December"};
    int m, y, mon, day, year;
    char date_string[STRSIZ];
    for (y = 1993; y < 2010; y += 10)
       for (m = 1; m <= 12; ++m) {
           printf("%s", nums_to_string_date(date_string,
                                            m, 15, y, month_names));
           string_date_to_nums(date_string, &mon, &day, &year, month_names);
           printf(" = %d/%d/%d\n", mon, day, year);
       }

    return (0);
}
```



```
char* nums_to_string_date(char *date_string,          /* output - string representation */
                    int          month,               /* input - */
                    int          day,                 /* representation */
                    int          year,                /* as three numbers */
                    const char *month_names[])        /* input - string representations of mont
{
      sprintf(date_string, "%d %s %d", day, month_names[month - 1], year);
      return (date_string);
}

   /* Value returned by search function if target
                          not found                                                */

/*
 * Searches for target item in first n elements of array arr
 * Returns index of target or NOT_FOUND
 * Pre: target and first n elements of array arr are defined and n>0
 */
int search(const char *arr[],         /* array to search                         */
       const char *target,        /* value searched for                       */
       int n)                     /* number of array elements to search   */
{
       int i,
           found = 0,    /* whether or not target has been found         */
           where;        /* index where target found or NOT_FOUND        */

       /* Compares each element to target                                 */
       i = 0;
       while (!found && i < n) {
           if (strcmp(arr[i], target) == 0)
                 found = 1;
            else
                 ++i;
       }

       /* Returns index of element matching target or NOT_FOUND */
       if (found)
             where = i;
       else
             where = NOT_FOUND;
       return (where);
}
```



```
/*
 * Converts date represented as a string containing a month name to
 * three integers representing month, day, and year
 */
void string_date_to_nums(const char *date_string,   /* input - date to convert */
                         int        *monthp,        /* output - month number   */
                         int        *dayp,          /* output - day number     */
                         int        *yearp,         /* output - year number    */
                         const char *month_names[]) /* input - names used in
                                                       date string             */
{
      char mth_nam[STRSIZ];
      int  month_index;

      sscanf(date_string, "%d%s%d", dayp, mth_nam, yearp);

      /* Finds array index (range 0..11) of month name.                       */
      month_index = search(month_names, mth_nam, 12);
      *monthp = month_index + 1;
}
```

Other way to copy Strings or Arrays

memcpy

defined in <string.h>

*void * **memcpy** (void * to, const void * from, size_t size)*

- Function memcpy copies a specified number of characters (*bytes*) from the object pointed to by its second argument into the object pointed to by its first argument.
- The function can receive a pointer to <u>any</u> type of object.
- The result of this function is *undefined* if the two objects overlap in memory (i.e., if they are parts of the same object)—in such cases, use **memmove**.
- Figure 8.31 uses memcpy to copy the string in array s2 to array s1.

```c
// Fig. 8.28: fig08_28.c
// Using function memcpy
#include <stdio.h>
#include <string.h>

int main(void)
{
   char s1[17]; // create char array s1
   char s2[] = "Copy this string"; // initialize char array s2

   memcpy(s1, s2, 17);
   printf("%s\n%s\"%s\"\n",
      "After s2 is copied into s1 with memcpy,",
      "s1 contains ", s1);
}
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

Fig. 8.28 | Using function `memcpy`.

Strcpy vs Memcpy [3]

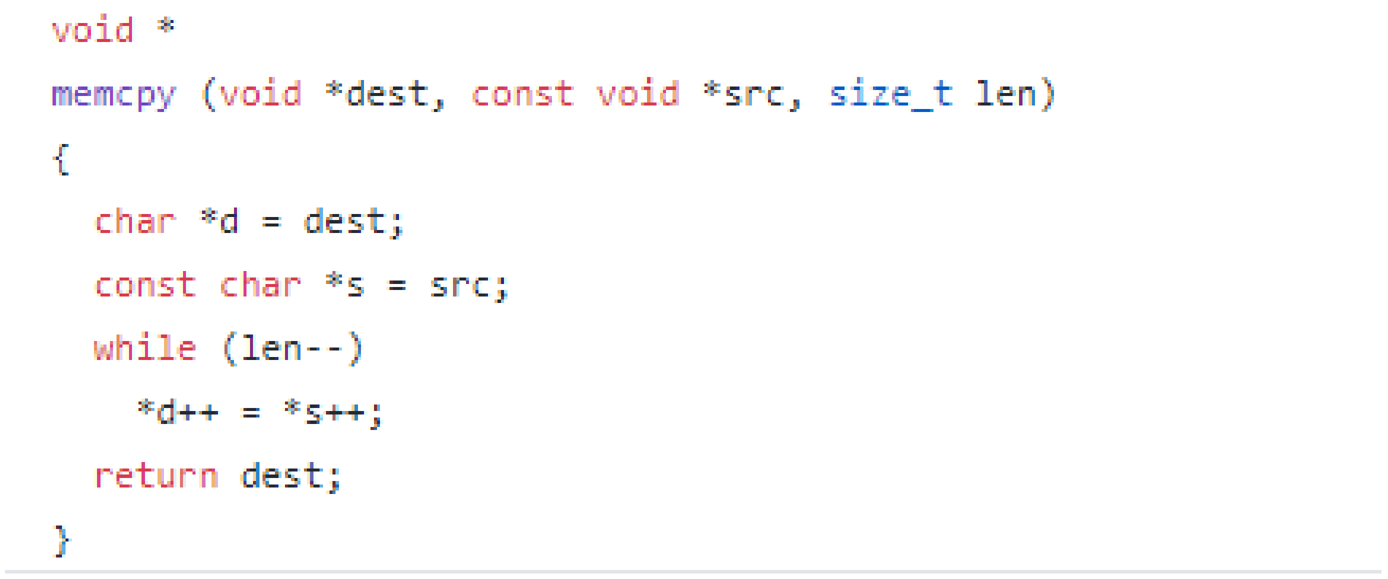
- strcpy() copies a string until it comes across the termination character '\0'. With memcopy(), the programmer needs to specify the size of data to be copied.
- memcpy() copies specific number of bytes from source to destination in RAM, where as strcpy() copies a constant / string into another string.
- memcpy() works on fixed length of arbitrary data, where as strcpy() works on null-terminated strings and it has no length limitations.
- memcpy() is used to copy the exact amount of data, whereas strcpy() is used of copy variable-length null terminated strings.

memcpy

- Memcpy copies memory areas and returns a pointer to destination
- <u>Can also be used with other data types</u>, e.g.

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
int b[10]={0};
memcpy(b, a, sizeof(int)* 10);
```

Example Implementation of memcpy



```
void *
memcpy (void *dest, const void *src, size_t len)
{
  char *d = dest;
  const char *s = src;
  while (len--)
    *d++ = *s++;
  return dest;
}
```

memmove

*void ** **memmove** *(void *to, const void *from, size_t size)*

- memmove copies the *size* bytes at *from* into the size bytes at *to*, even if those two blocks of space overlap.
- In the case of overlap, memmove is careful to copy the original values of the bytes in the block at from, including those bytes which also belong to the block at to.

- The value returned by memmove is the value of to.

```
/* memmove example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "memmove can be very useful......";
  memmove (str+20,str+15,11);
  puts (str);
  return 0;
}
```

Output: memmove can be very very useful.

A Brief Intro to Dynamic Memory Allocation

A Brief Intro to Dynamic Memory Allocation

- Manual memory management in C
- Functions:
  - malloc
  - calloc
  - realloc
  - free
- Sometimes, you do not know the actual size of an array until run time.A simple example:
  - Assume that the string you entered as a user does not fit into the character array you declared.
  - What do you do in such a case?

malloc

defined in <stdlib.h>

*void ** **malloc** *(size_t size)*

- char *p;
- p = malloc(5);
  - area of 5 bytes is reserved
  - addres of this memory area’s beginning is now assigned to p
  - this example reserves an area for 5 elements since the type of the pointer is char (1 bytes)
- Write correctly:
  - p = malloc(sizeof(char)*5);
- Be sure that return type is correct:
  - p = (char*)malloc(sizeof(char)*5);

malloc

- int *p;
- p = malloc(20);

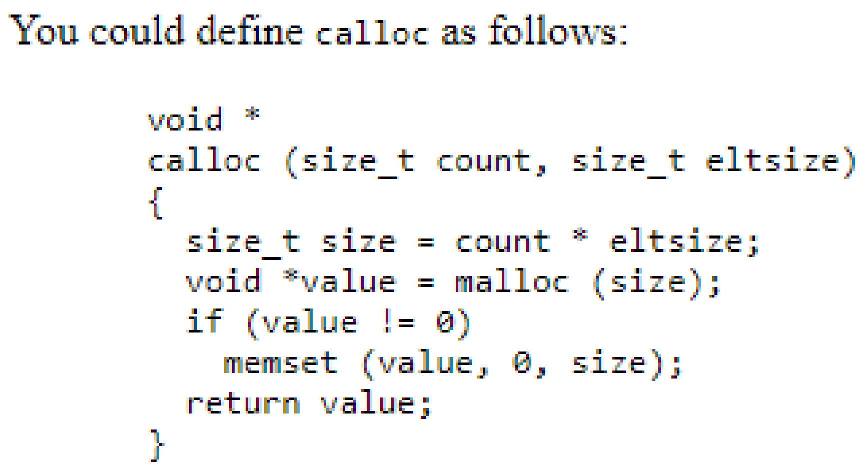
- == ??
- int *p; p = (int *)malloc(sizeof(int)*5);

calloc

defined in <stdlib.h>

*void ** **calloc** *(size_t count, size_t eltsize)*

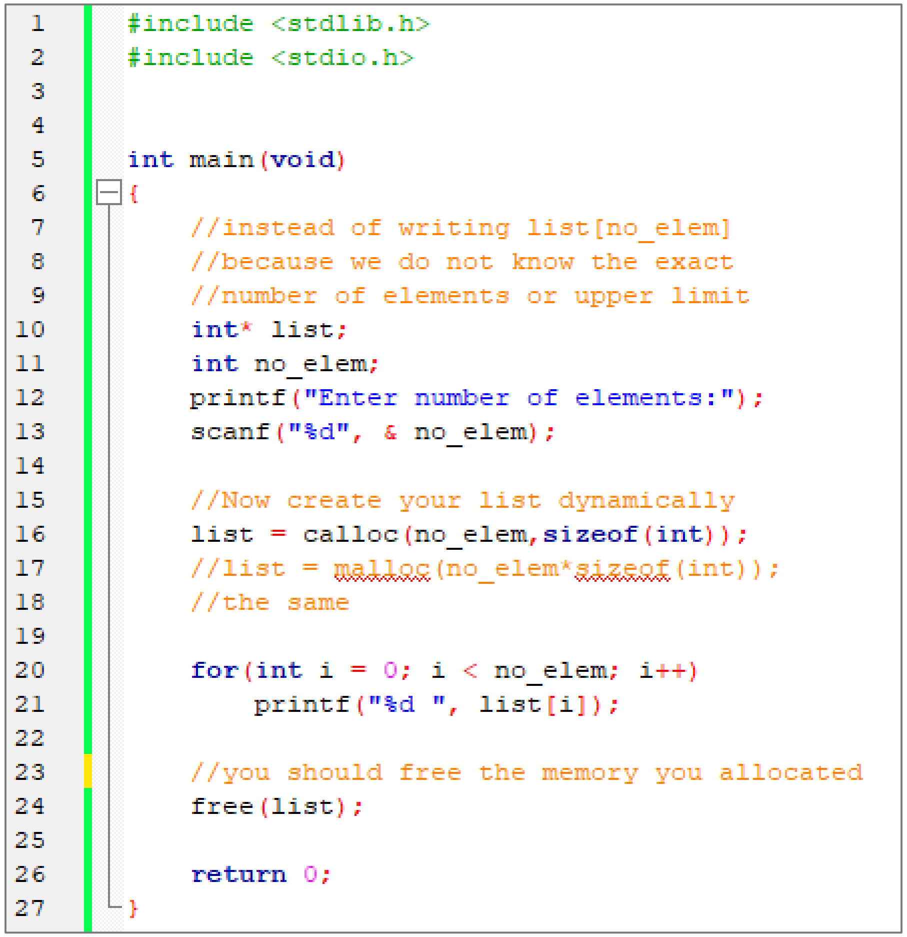
- This function allocates a block long enough to contain a vector of count elements, each of size eltsize.
- Its contents are cleared to zero before calloc returns.



You could define `calloc` as follows:

```
void *
calloc (size_t count, size_t eltsize)
{
  size_t size = count * eltsize;
  void *value = malloc (size);
  if (value != 0)
    memset (value, 0, size);
  return value;
}
```

Example



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


int main(void)
{
    //instead of writing list[no_elem]
    //because we do not know the exact
    //number of elements or upper limit
    int* list;
    int no_elem;
    printf("Enter number of elements:");
    scanf("%d", & no_elem);

    //Now create your list dynamically
    list = calloc(no_elem,sizeof(int));
    //list = malloc(no_elem*sizeof(int));
    //the same

    for(int i = 0; i < no_elem; i++)
        printf("%d ", list[i]);

    //you should free the memory you allocated
    free(list);

    return 0;
}
```

Example cont.

- Guarantee that memory is allocated:

```
        list = calloc(no_elem,sizeof(int))
        if(list == NULL)
                printf("not enough storage");
```

Realloc

- Widens or narrows down the space allocated before using malloc or calloc
- Gets 2 parameters: the starting address of the previous block of data and the new size
- int *p;
- p = calloc(15,sizeof(int));
- p = realloc(p,sizeof(int)*5);
- returns the new beginning
- What if there is not enough space next to the current block in case of extending? Carries all data together to another appropriate space

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *ptr = (int *)malloc(sizeof(int)*2);
int i;
int *ptr_new;

*ptr = 10;
*(ptr + 1) = 20;

ptr_new = (int *)realloc(ptr, sizeof(int)*3);
*(ptr_new + 2) = 30;
for(i = 0; i < 3; i++)
            printf("%d ", *(ptr_new + i));

getchar();
return 0;
}
```

Wrap Up

- Strings in C are arrays of characters terminated by the null character ‘\0’.
- String input is done using
  - scanf and fscanf for strings separated by whitespace
  - gets and fgets for input of while lines
  - getchar and getc for single character input

Wrap Up

- The string library provides functions for
  - assignment and extraction
  - string length
  - concatenation
  - alphabetic comparison
- The standard I/O library includes functions for
  - string-to-number conversion
  - number-to-string conversion

References

1. Problem Solving & Program Design in C, Jeri R. Hanly & Elliot B. Koffman, Pearson 8. Edition, Global Edition
2. C How to Program, Paul Deitel, Harvey Deitel. Pearson 8th Edition, Global Edition.
3. http://www.careerride.com/C-strcpy()-and-memcpy().aspx