Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 8)

❖ Memory Allocation

In many cases, the exact sizes of arrays to be used by the program cannot be determined before the compilation of the program. This brings inflexibility to the program if a static array is used. An alternative is allocating the memory required for a variable (or for an array) dynamically.

The following functions, which are defined in standard library (stdlib.h) can be used for this purpose:

- Malloc Function:

malloc() function allocates a specified size of memory space and returns the beginning address of the newly allocated memory space. For example,

```
int *ptrk;
ptrk = malloc(10 * sizeof(int));
```

reserves space in memory that is able to hold 10 integers. Then, it is possible to use this reserved space by using the pointer ”ptrk”. Examine the below example;

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

int main()
{
   int *pk = malloc(10 * sizeof(int));  // Reserve memory space
   *pk = 6;         // Let the first element be 6
   *(pk+1) = 20;    // Let the second element be 20
   *(pk+9) = -35;   // Let the ninth element be -35

   printf("%d", *(pk+1));  // Prints 20 on the screen

   return 0;
}
```

- Calloc Function:

calloc() function works like malloc() but two arguments are used the of which determines the number of elements in the memory area and the second of which determines the number of bytes required by each element. All elements are initialized to 0 in the beginning. For example;

```
int *ptrk;

ptrk = calloc(10, sizeof(int));
```

The above code reserves space in memory (10 integers) and then initialized this memory space by zeros.

- Free Function:

Every reserved memory space must be released when it is no more necessary. The function **free()** can be used for this purpose. For example,

```
free(ptrk);
```

frees the memory space occupied (pointed) by the pointer “ptrk”.

Consider the function named “MatrixAdd” considered in the previous lecture notes. Instead of returning “result”, we can return the memory address directly. Please check the below code segment carefully (You can find the definition of the other functions, such as “PrintTable” in the previous lecture notes).

```
double* MatrixAdd(double *A, double *B, int n, int m)
{
	int i,j;
	double *C=malloc(n*m*sizeof(double));
	for(i=0;i<n;i++)
		for(j=0;j<m;j++)
			*(C+i*m+j)=*(A+i*m+j)+*(B+i*m+j);
	return C; // Return the memory space that holds the result
}

int main()
{
   double matA[3][2]={{1.0, 2.4},{3.1, 0.2},{1.3, -2.0}};
   double matB[3][2]={{0.4, -3.2},{-4.0, 1.4},{5.3, 0.2}};
   double *result;  // Keeps the result of the matrix sum

   result = MatrixAdd((double*)matA,(double*)matB, 3, 2);

   PrintTable((double*)result, 3, 2);

   free(result);    // Clear (free) the memory space pointed by “result”

   return 0;
}
```

❖ Strings

A string is a character array, with a null character ('\0') used to mark the end of the string. For instance,

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

defines a character string. It is possible to use double quotes to enclose a string constant. For example,

```
char str[6] = "Hello";
```

The size of the array is always one more than the length of the string due to the fact that the null character is added at the end. the compiler automatically adds the null character at the end of the string constant.

- Reading and Writing Strings:

gets() and **puts()** functions can be used to read character strings from the standard input stream and write character strings to standard output stream, respectively. Check the below C program:

```
#include <stdio.h>
#define MAX_CHARS 50

int main()
{
   char str[MAX_CHARS + 1];

   printf("Enter a string (at most %d characters):\n", MAX_CHARS);
   gets(str);  // Get the string from user
   puts("You entered: ");
   puts(str);  // Put the string on the screen
   return 0;
}
```

It is also possible to use **%s** specifier with printf and scanf functions to read and write strings. The above code is written by the help of **printf()** and **scanf()** functions below:

Please note that **scanf()** function does NOT read anything after a space character. So in order to read strings that can contain space characters you should use **gets()** function!!

```c
#include <stdio.h>
#define MAX_CHARS 50

int main()
{
   char str[MAX_CHARS + 1];

   printf("Enter a string (at most %d characters):\n", MAX_CHARS);
   scanf("%s", str);  // Get the string from user (& not used! Array!)
   printf("You entered: %s", str);  // Put the string on the screen
   return 0;
}
```

- Strings as Arguments to Functions:

Since strings are arrays of characters they can be passed to functions as arguments by the help of pointers. For example,

```c
#include <stdio.h>
#define MAX_CHARS 50

void ToUpperCase(char *);

int main()
{
   char str[MAX_CHARS + 1];

   printf("Enter a string (at most %d characters):\n", MAX_CHARS);
   gets(str);
   ToUpperCase(str);  // Convert all letters to capital letters
   printf("You entered: %s", str);
   return 0;
}

void ToUpperCase(char *str)
{
   char *ps;
   for(ps=str; *ps != '\0'; ps++)
      if(*ps >= 'a' && *ps <= 'z')
         *ps = *ps + 'A' - 'a';
}
```

In the **ToUpperCase()** function, all (lower case) letterss are converted to the corresponding capital characters. If the corresponding character is a lower case letter, then we subtracted **32** (97-65) from the value of the memory location.

- String.h Library:

There are several readily defined functions to work with strings in the string library. These include;

strlen(str): Finds the length of the null terminated string “str” (returns an integer).

strcpy(deststr, str): Copies the contents of a null terminated string ”str” to “deststr”. It is programmer’s responsibility to make sure that “deststr” can contain all the characters in “str”.

strcmp(str1, str2): Compares the null terminated strings “str1” and “str2”. If both are equal **0** is returned. If, otherwise, a value less than or greater than 0 is returned depending on the lexicographical order of “str1” and “str2”.

strcat(deststr, str): Appends a copy of “str” to “deststr”. This is used for concatenation of character strings.

Check the below example:

```
#include <stdio.h>
#define MAX_CHARS 100
int main()
{
   char str[MAX_CHARS + 1], strC[MAX_CHARS + 1];
   char *pch=str;
   printf("Enter a string (at most %d characters):\n", MAX_CHARS);
   gets(str);
   printf("Length of the string : %d\n", strlen(str));
   strcpy(strC, str);
   printf("strC becomes after strcpy: %s\n", strC);
   printf("The same memory can be reached by pch: %s\n", pch);
   return 0;
}
```

It is also possible to specify the maximum operation size by using the following functions:

strncpy(deststr, str, n): Similar to strcpy() except that at most n characters are copied. Note that strncpy() does not necessarily put a null character at the end of the copied part of the string. This is programmer's responsibility.

strncmp(str1, str2, n): Similar to strcmp() except that at most n characters are compared.

strncat(deststr, str, n): Similar to strcat() except that at most n characters (not counting the null character) from "str" are appended to "deststr".

❖ The Enum Data Type

The enumerated (enum) data type can be used to declare named integer constants. It makes the C program more readable and easier to maintain. Syntax:

```
enum tagName {item1, item2, ...} var1, var2, ...;
```

Here "tagName" is the name of the enumeration, "item1, item2, ..." are the names to represent integer constants, and "var1, var2 ..." are the variables of this newly defined type. For instance,

```
enum gender {male, female} studentG;
```

Declares an enumarated type called "gender", which can have values **male (0)** or **female (1)**. And a variable named "studentG" is also defined. Now, we can do the following:

```
studentG = male; // Same as doing studentG = 0
printf("%d\n", studentG); // Prints 0 on the screen
```

Please also check the below C program:

```
#include <stdio.h>
enum gender{male,female};
void PrintGender(enum gender g)
{
   if(g == male) printf("male");
   else if(g == female) printf("female");
   else printf("unknown");
}
```

```
int main()
{
   enum gender guser;

   printf("Enter 0 for male, 1 for female\n");
   scanf("%d", &guser);
   printf("Your gender: ");
   PrintGender(guser);
   return 0;
}
```

❖ Type Definitions

It is possible to define an alias for a given data type using the "typedef" statement. For example,

```
typedef int INTEGER;
```

defines an alias for the data type int called INTEGER.

Later in the program, INTEGER can be used wherever int is used as illustrated below:

```
INTEGER k, *pk;
```

The above code declares two variables one of which is an integer and the other is a pointer to integers. Check the below example for the usage of "typedef":

```
#include <stdio.h>

typedef char *string;
typedef string strArray[2];

int main()
{
   string str = "Hello World!";
   strArray name = {"Emre", "Dincel"};

   printf("%s", str);  // Prints Hello World! on the screen.
   printf("%s", name[0]);  // Prints Emre on the screen.
   printf("%s", name[1]);  // Prints Dincel on the screen.
   return 0;
}
```

There are two usages of typedef:

- It can provide a shorthand alias for data types that are long and difficult to comprehend.
- It brings flexibility to the program in the sense that the types of all variables in a certain data type can be changed easily. For example, if we want all variables of type INTEGER to become long integers in the program all we have to do is to change the “typedef” line in the program.