Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 10)

❖ Structures

The data types we have seen so far were simple in the sense that they represent only a single data unit such as an integer or real number. Sometimes the data is of a more complex type. For example, the data to hold the information about a person may consist of the name, surname, gender and date of birth of that person. In order to "give reference to such data by using a single variable", it is possible to define new data types called **structures**.

A structure can be defined using the following syntax:

```
struct structTag {
   [component_definition_1]
   [component_definition_ 2]
   ...
};
```

Here, definitions for components are in the same form with the definitions for variables. Consider the following example;

```
struct Complex {
   double Re, Im;
}
```

The above code shows a data type that consists of two components with names "Re" and "Im" both of which are of type double. Structure variables can be defined by using the following syntax:

```
struct structTag var1, var2, ...;
```

For example;

```
struct Complex cVar;
```

defines a variable named "cVar" of **Complex** type.

It is possible to give reference to components of a structure variable by using the dot (**.**) operator. For instance,

```
cVar.Re = -2.0;
cVar.Im = 1.5;
```

causes the **Re** component of the variable "cVar" to become -2.0 and, the **Im** component of the variable "cVar" to become 1.5, respectively.

Components referenced in this way can be used as if they were variables (in printf() and scanf() statements, as arguments to functions etc.). For example,

```
printf("Complex number: %lf+j%lf", cVar.Re, cVar.Im);
```

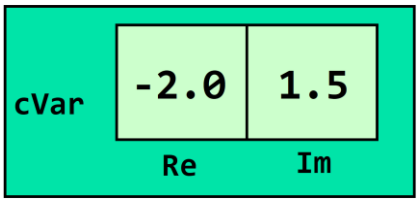
prints "$-2.0 + j1.5$" on the screen.

- Initializing Structures:

A structure can be initialized by a list of data called initializers. The types of corresponding items in component list and initializers should match. For example,

```
struct Complex cVar = {-2.0, 1.5};
```

declares a complex variable "cVar" and assigns -3.0 and 1.5 to **Re** and **Im** components, respectively.



Remarks;

- It is possible to have arrays and other structures as components of structures.
- It is also possible to have array variables of structures.
- It is possible to pass structures as arguments to functions and functions can return structures as their output value.

- Pointers to Structures:

It is possible to declare pointers that can point to structure variables. The arrow (**->**) operator can be used with the pointer to directly reach the members of the structure. For example,

```
#include <stdio.h>

struct automobile {
   int year;
   char make[8], model[8];
   int engPower;
   float weight;
};
```

```
int main()
{
   struct automobile myCar = {1982, "BMW", "3.16i", 1600, 1543.5};
   struct automobile *ptrCar = &myCar;

   printf("myCar.year \t= %d\n", myCar.year);
   printf("(*ptrCar).year \t= %d\n",(*ptrCar).year);
   printf("ptrCar->year \t= %d\n",ptrCar->year);
   return 0;
}
```

In the above program, we have used 3 different ways to reach the same memory location. On the other hand, note how a member of the structure pointed by the pointer “ptrCar” is referenced by -> operator.

- Structures as Arguments of Functions:

It is possible to pass an entire structure to a function. In addition, a function can return a structure back to its caller. For example, the below program changes the year of the defined car which is sent to the corresponding function.

```
#include <stdio.h>

struct automobile {
   int year;
   char make[8], model[8];
   int engPower;
   float weight;
};
// Instead of writing “struct automobile” each time, we can use “sauto”
typedef struct automobile sauto;

sauto ChangeModel(sauto car)
{
   sauto temp = car;
   printf("Current model is %s\n", temp.model);
   printf("Enter new model: \n");
   gets(temp.model);
   return temp;
}
```

```
int main()
{
   sauto myCar = {1982, "BMW", "3.16i", 1600, 1543.5};
   sauto *ptrCar = &myCar;

   myCar = ChangeModel(myCar);
   printf("myCar.model = %s\n", myCar.model); // Or alternatively;
   printf("ptrCar->model = %s\n", ptrCar->model);

   return 0;
}
```

Please note that a struct may include another struct as a member. Check the below code segments:

```
struct structPerson {
   string name, surname;
   int age;
   gender genderP;
};
typedef struct structPerson person;

struct structStudent {
   person individual;
   string department;
   int quiz[6];
   int homework[6];
};
typedef struct structStudent student;
student classroomStudents[50];
```

❖ File Operations

In our programs so far, we have only used the standard input, which is normally the keyboard, for receiving data from the outer world, and the standard output, which is normally the screen, for sending data to the outer world. Therefore, when we exit the program all the information contained in the program (values of variables) is lost. This means that every time we start our computer we have to re-enter any necessary input data, re-

calculate any necessary calculations, and print out the results. However, one of the most important characteristics of computers is their ability to store data for later use!

The data, in general, is stored in files on hard-disks, floppy-disks, magnetic-tapes, zip-drives, CD-ROMS etc. Files that contain data to be processed by programs are called data files.

- The FILE Structure:

The FILE structure is defined in stdio.h and is used to access files. In order to reach files we first define a pointer to FILE as follows:

```
FILE *fptr;
```

To be processed, each file should be opened first. **fopen()** function is used to open the file. The syntax is given as below:

```
fptr = fopen(fileName, mode);
```

Here, fptr is a pointer to FILE structure, "fileName" is a string consisting of the path and name of the file to be opened, and "mode" is another string that determines the opening mode of the file. After the file is opened it can be reached via "fptr". If **fopen()** is not successful for any reason (for example file does not exist or the disk is full etc.) then it returns NULL value.

The "mode" parameter is made by a combination of the characters r (read), w (write), b (binary), a (append), and + (update). Please check the below table:

r	Opens existing text file for reading
w	creates a text file for writing
a	opens an existing text file for appending
r+	opens an existing text file for reading or writing
w+	creates a text file for reading and writing
a+	opens or creates a text file for appending
rb	opens an existing binary file for reading
wb	creates a binary file for writing
ab	opens an existing binary file for appending
r+b	opens an existing binary file for reading or writing
w+b	creates a binary file for reading and writing
a+b	opens or creates a binary file for appending

It is a good programming practice to close any files opened before the termination of the program. It is also a requirement that a file has to be closed before it can be opened again

(perhaps in another mode). This is done by using the **fclose()** function. The syntax is given as follows:

`fclose(`*`fptr`*`);`

Here, “fptr” is a pointer to a FILE that has been opened.

- Reading and Writing Text Files:

fgetc(*filePtr*) and **fputc**(ch, *filePtr*) functions can be used to read and write characters to text files. Here, “ch” is the character to be written to the file and “filePtr” is a pointer to the opened FILE. Also, **fgetc**() returns the character read from the file.

fgets(*str*, *n*, *filePtr*) and **fputs**(*str*, *filePtr*) functions can be used to read and write strings to text files. Here, ”str” is a pointer to a string (or a character array), “filePtr” is a pointer to an opened FILE and “n” is an integer that shows the maximum number of characters to be read from the file.

It is also possible to use **fprintf()** and **fscanf()** for formatted input/output. These are very similar to printf() and scanf() functions, except that they require a FILE pointer as the first argument.

Examine the below example;

```
#include <stdio.h>

int main()
{
   FILE *fptr = fopen("test.txt", "w");  // Open in write-only mode

   if (fptr == NULL)
   {
       printf("Error: File could not be opened!");
       return -1;
   }

   // Write a string to the file
   fprintf(fptr, "This text is written to a file...");
   fclose(fptr);

   return 0;
}
```

The below program reads the data previously written in test.txt file.

```c
#include <stdio.h>

int main()
{
   char buffer[255];
   FILE *fptr = fopen("test.txt", "r");  // Open in read-only mode

   if (fptr == NULL)
   {
       printf("Error: File could not be opened!");
       return -1;
   }

   fgets(buffer, 250, fptr);  // Read at most 250 chars from the file
   printf("%s\n", buffer);    // This text is written to a file...
   fclose(fptr);

   return 0;
}
```

- End of file:

When reading data from files, it is necessary to test whether the end of the file has been reached or not. If fgetc() or fscanf() functions are used they return an EOF character when the end of the file has been reached. On the other hand, if fgets() is used it returns a NULL pointer. It is also possible to use the **feof()** (end of file) function to test whether the end of the file has been reached or not.

An Example Program:

In an experiment, values of the temperature in a container are observed at several times during a day and the results of observations are entered from the keyboard to be saved in a file, whose name is entered by the user.

The program asks the user the time and the corresponding temperature repetitively until the temperature entered is –99. The time should be read as a four character string in the “hhmm” format. The program should allow integer temperatures between –99 and 250. As the information is being entered from the keyboard, it should be written to the output file in a suitable tabular format.

Please carefully examine the below program:

```c
#include <stdio.h>

int main()
{
   FILE *ftemp;
   char filename[250], hourMinute[5];
   int temp;

   printf("Enter the name of the file:\n");
   scanf("%250s", filename);

   ftemp = fopen(filename, "w");

   if (ftemp == NULL)
   {
       printf("Problem in opening the file!");
       return -1;
   }
   fprintf(ftemp, "Time \t Temp \n ---- \t ---- \n");

   do
   {
      printf("Please enter time:\n");
      scanf("%4s", hourMinute);

      do {
         printf("Please enter temperature:\n");
         scanf("%d", &temp);
      } while(temp<-99 || temp>250);

      if (temp != -99)
         fprintf(ftemp, "%4s \t %4d\n", hourMinute, temp);

   } while(temp != -99);

   fclose(ftemp);

   return 0;
}
```

```
Time Temp
---- ----
0910   45
0930   55
0950   64
```

An example file output after execution can be seen in the above figure.

The below program, on the other hand, prints the contents of the text file:

```
#include <stdio.h>

int main()
{
   FILE *ftemp;
   char filename[250], buffer[80];
   int flag = 0;

   printf("Enter the name of the file:\n");
   scanf("%250s", filename);

   ftemp = fopen(filename, "w");

   if (ftemp == NULL)
   {
       printf("Problem in opening the file!");
       return -1;
   }
   while (!feof(ftemp))
   {
      flag = fgets(buffer, 80, ftemp);

      if (flag != NULL)
         printf("%s", buffer);
   }

   fclose(ftemp);

   return 0;
}
```

Note: Please also check the below functions which are important when you work with files:

fread() fwrite() fgetpos() fsetpos() fseek() ftell() rewind()
rename() sscanf() etc...