CS1010E Lecture 11
Strings and Structures

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
`www.comp.nus.edu.sg/~joxan`
`cs1010e@comp.nus.edu.sg`

Semester II, 2016/2017

Lecture Outline

- Strings
- Structures
- Using Functions with Structures.
- Arrays of Structures.

String Definition and I/O

- Character string constants are enclosed in double quotes, as in `"sensor1.txt"`, `"r"`, and `"15762"`.

- A character string array can be initialized using string constants, or using character constant, as shown in the following equivalent statements:

  - `char filename[12]="sensor1.txt";`

  - `char filename[]="sensor1.txt";`

  -
    ```
    char filename[]={'s','e','n',
          's','o','r','1','.','t','x',
          't','\0'};
    ```

String Definition and I/O

- Note that in general, to store a string that has n characters, the character string array must have a size of at least $n+1$ to store the null character.

- We cannot split up the declaration and initialization like this:

  -
    ```
    char filename[12];
    filename="sensor1.txt";
    ```

- Instead, we must use the `strcpy` function as follows:

  -
    ```
    char filename[12];
    strcpy(filename,"sensor1.txt");
    ```

String Definition and I/O

- We may also use `scanf` and `fscanf` with the `%s` format specifier to read “words” from the keyboard or from a file. Words are separated by one or more whitespaces.

  -
    ```
    char word[101];
    printf("Enter one word: ");
    scanf("%s", word);
    ```

- Note that we do not need the address-of operator `&` in the call to `scanf` since `word` is a character array and it contains the address of the first character.

String Definition and I/O

- When reading strings from the keyboard, we always call `fflush` before `gets`, `fgets`, and `scanf` to ensure that stray characters are flushed from the standard input `stdin`.

- Even so, it is not advisable to mix `gets` / `fgets` with `scanf` as unexpected results can occur even if `fflush` is always used.

String Conversion Functions

- Often, it is useful to convert a number stored as a string to an `int` or a `double`.

  -
    ```
    char s[] = "4735";
    int i;
    i = atoi(s); /* Stores 4735 in i. */
    ```
- We can do the same for floating-point numbers:

  -
    ```
    char s[] = "24.43";
    double d;
    d = strtod(s, NULL);
    /* Stores 24.43 in d. */
    ```
- The function `strtof` works similarly for `float` variables.

String Conversion Functions

- The reverse conversion from an `int` or a `double` to a string is also possible using `sprintf`:

  -
    ```
    char s1[101], s2[101];
    int i = 4735;
    double d = 24.43;

    sprintf(s1,"%i",i);
    /* Stores "4735" in s1. */

    sprintf(s2,"%i and %.3f",i,d);
    /* Stores "4735 and 24.430" in s2. */
    ```

String Functions

```
/*-----------------------------------------------------*/
/*  This function determines the length of a string.  */
/*  Does not include the null character '\0'.         */

int strg_len_1(char s[])
{
    /*  Declare variables.  */
    int k=0;

    /*  Count characters.  */
    while (s[k] != '\0')
        k++;

    /*  Return string length.  */
    return k;
}
```

String Functions

- The Standard C library contains a number of functions for working with strings.
- In the next few slides, assume that `s` and `t` are character strings.
- The variable `n` is an unsigned integer, and `c` is an integer that is converted to a character.
- Usage of these functions require that you include `<string.h>`.

String Functions

(Examples)

`strlen(s)`	Returns the length of the string `s`.
`strcpy(s,t)`	Copies string `t` to string `s`. Returns a pointer to `s`.
`strcat(s,t)`	Concatenates string `t` to the end of string `s`. Returns a pointer to `s`.
`strcmp(s,t)`	Compares string `s` to string `t` in an element-by-element comparison, starting with `s[0]` and `t[0]`. A negative value is returned if `s<t`, zero is returned if `s` is identical to `t`, and a positive value is returned if `s>t`.

More on `strcmp(s, t)`

- Returns 0
  if the strings `s` and `t` are the same.
- Returns a positive number
  if `t` is a shorter substring of `s`
- Returns a negative number
  if `s` is a shorter substring of `t`
- Returns `s[i] - t[i]`
  where `i` denotes the first position of disagreement.

Example: `strcmp("123", "127")` returns -4.

Strcpmp

```
void strcpy2(char dest[], char src[]) {
    char *dp = &dest[0], *sp = &src[0];
    while(*sp != '\0')
       *dp++ = *sp++;
    *dp = '\0';
}

strcmp2(char *str1, char *str2) {
    char *p1 = &str1[0], *p2 = &str2[0];
    while(1) {
        if(*p1 != *p2)
            return *p1 - *p2;
        if(*p1 == '\0' || *p2 == '\0') return 0;
        p1++;
        p2++;
    }
}
```

NOTE: This precisely defines the meaning of the `strcmp` function in `string.h`

Summary of Strings

A string is an array of `char` terminated with the NULL char '\0'.

-
  ```
  char s[] = "abc";
  ```

-
  ```
  char *s = "abc";
  ```

-
  ```
  char s[5];
  s[0] = 'a'; s[1] = 'b'; s[2] = 'c'; s[3] = '\0';
  ```

-
  ```
  char s[5];
  strcpy(s, "abc");
  ```
- Not correct:
  ```
  char *s;
  strcpy(s, "abc");
  ```

Pitfalls:

- When a string pointer is used to modify a string, be aware that the memory location changed is valid.
- When declaring and initializing simultaneously eg: `char *s = "abc";`
  the memory locations for "abc" may NOT be valid for writing.
  (It is valid for `char s[] = "abc";` - why?)

Structures

- When solving engineering problems, it is often necessary to work with large amounts of data.
- We learned that arrays provide a convenient way to store and manipulate large data sets.
- But arrays work only if all the data are of the same type.
- In many cases, the data that represent an object or a set of information has multiple data types.

Structures

- For example, hurricanes are given names, and they are categorized by the intensity of their winds.
- Thus, to represent a hurricane, we might include the hurricane's name, the year in which it occurred, and its intensity category.
- A character string could represent the name, and integers could represent the year and intensity.
- A **structure** defines a set of data, but the individual parts of the data do not have to be of the same type.

Structures

- Thus, a structure can be defined to represent a hurricane as follows:

  -
    ```
    struct hurricane
    {
        char name[10];
        int year, category;
    };
    ```
- We can now defined variables, and even arrays, of type `struct hurricane`.
- Each variable or array element would contain three data values—a character string and two integers.

Structures

- Structures are often called aggregate data types, because they allow multiple data values to be collected into a single data type.

- Individual data values within a structure are called **data members**, and each data member is given a name. The name is also called an **attribute**.

- In our previous example, the names of the data members are `name`, `year`, and `category`.

Structures

- A data member is referenced using the structure variable name followed by a period (called the **structure member operator**) and a data member name.

- Note the difference between referencing a value in a structure and in an array: A value in an array is referenced with the array name and subscript.

Definition and Initialization

- To use a structure in a program, you must first define the structure.
- The keyword `struct` is used to define the name of the structure (also called the structure **tag**) and the data members that are included in the structure.
- After the structure has been defined, structure variables can be defined using declaration statements.

Definition and Initialization

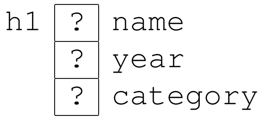
- Consider the previous definition for a structure representing a hurricane.
- The name of the structure is `hurricane`.
- The three data members are `name`, `year`, and `category`.
- Note that a semicolon is required after the structure definition.
- The statements to define a structure can appear before the `main` function or stored in a header file.

Definition and Initialization

- It is important to note that these statements do not reserve any memory—they only define the structure.
- To define a variable of this structure type, we use the following statement in the declaration section of our program:
  - `struct hurricane h1;`

Definition and Initialization

- The preceding statement defines a variable named `h1` that has three data members; this is shown in the following diagram:



- The declaration statement allocates memory for the three data members, but initial values have not been assigned; thus, their values are unknown.

Definition and Initialization

- The data members of a structure can be initialized in a declaration statement or with program statements.

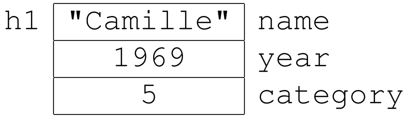
- To initialize a structure with a declaration statement, the values are specified in a sequence that is separated by commas and enclosed in braces.

- The following declaration statement defines and initializes the example structure `h1`:

  - `struct hurricane h1={"Camille",1969,5};`

Definition and Initialization

- The data members of `h1` are now initialized:



- To initialize the structure `h1` using program statements, we can use statements such as the following:

  -
    ```
    strcpy(h1.name,"Camille");
    h1.year = 1969;
    h1.category = 5;
    ```

Input and Output

- We can use `scanf` or `fscanf` statements to read values into the data members of a structure, and `printf` or `fprintf` to print the values of the data members.
- The structure member operator must be used to specify an individual data member.
- In the next program, the information for a group of hurricanes is read from a file named `storms2.txt`, and the information is printed to the screen.

Input and Output

- The contents of the file `storms2.txt` looks like this:

  -
    ```
    Hazel       1954    4
    Audrey      1957    4
    Donna       1960    4
    Carla       1961    4
    Camille     1969    5
    Celia       1970    3
    Frederic    1979    3
    Allen       1980    3
    Gloria      1985    3
    Hugo        1989    4
    Andrew      1992    5
    Opal        1995    3
    ```

Input and Output

```
/*-----------------------------------------------------*/
/*  Program chapter7_1                                 */
/*                                                     */
/*  This program reads the information for a hurricane */
/*  from a data file and then prints it.               */

#include <stdio.h>
#define FILENAME "storms2.txt"

/*  Define structure to represent a hurricane.  */
struct hurricane
{
    char name[10];
    int year, category;
};
```

Input and Output

```
int main(void)
{
    struct hurricane h1;
    FILE *storms;

    storms = fopen(FILENAME,"r");
    if (storms == NULL)
        printf("Error opening data file. \n");
    else {
        while (fscanf(storms,"%s %d %d",h1.name,&h1.year,
                    &h1.category) == 3) {
            printf("Hurricane: %s \n",h1.name);
            printf("Year: %d, Category: %d \n",h1.year,
                    h1.category);
        }
        fclose(storms);
    }
    return 0;
}
```

Input and Output

- Note that the reference in the `fscanf` function for the hurricane name is `h1.name` instead of `&h1.name`.

- Since `name` is a character string, the variable name represents an address or pointer and there is no need to put the address-of `&` operator.

- The first four lines of the sample run is:

  -
    ```
    Hurricane: Hazel
    Year: 1954, Category: 4
    Hurricane: Audrey
    Year: 1957, Category: 4
    ```

Computations

- We have seen that the structure member operator (`.`) is used with the name of the structure variable to access individual data members of the structure.

- When the name of the structure variable is used without the structure member operator, it refers to the entire structure.

- The assignment operator can be used with the structure variables of the same type to assign an entire structure to another structure, as shown in the next slide.

Computations

```
struct hurricane
{
    char name[10];
    int year, category;
};

int main(void)
{
    /*  Declare variables.  */
    struct hurricane h1={"Audrey",1957,4}, h2;

    h2 = h1;
}
```

Computations

- However, relational operators cannot be applied to an entire structure.
- To compare one structure with another, you must compare the individual data members.
- To illustrate computations with structures, consider a program that reads `storms2.txt` and then prints the names of all category five hurricanes.

Computations

```
/*-----------------------------------------------------------*/
/*  Program chapter7_2                                       */
/*                                                           */
/*  This program reads hurricane information from a data     */
/*  file and then prints the names of category 5 hurricanes. */

#include <stdio.h>
#define FILENAME "storms2.txt"

/*  Define structure to represent a hurricane.  */
struct hurricane
{
    char name[10];
    int year, category;
};
int main(void)
{
    struct hurricane h1;
    FILE *storms;
```

Code on Next Slide

```
   /* Exit program. */ return 0; }
```

Body Code for Program `chapter7_2`

```
storms = fopen(FILENAME,"r");
if (storms == NULL)
    printf("Error opening data file. \n");
else {
    printf("Category 5 Hurricanes: \n");
    while (fscanf(storms,"%s %d %d",h1.name,&h1.year,
                &h1.category) == 3)
        if (h1.category == 5)
            printf("%s \n",h1.name);
    fclose(storms);
}
```

Using Functions with Structures

- Structures can be used as arguments to functions, and functions can return structures.

- We will consider each of these cases separately.

Function Arguments

- Entire structures can be passed as arguments to functions.
- When a structure is used as a function argument, it is a call-by-value reference.
- When a function reference is made, the value of each data member of the actual parameter is passed to the function and is used as the value of the corresponding data member of the formal parameter.
- Thus, changing the value of a formal parameter does not change the corresponding actual parameter.

Function Arguments

- When functions are written to modify the value of a data member within a structure, we must use a pointer to the structure as the function argument.
- This allows the function direct access to the data members of the structure.
- When data members are accessed via a pointer to the structure, the **pointer operator** (−>) is used instead of the structure member operator.

Functions Returning Structures

- A function can be defined to return a value of type `struct`.
- After the function is called, an entire structure is returned to the calling function.
- This is illustrated by the function `get_info` in the next program.

Function Arguments

```
/*-------------------------------------------------------------------
/*  Program chapter7_3a
/*
/*  This program reads the information for a hurricane
/*  from the screen, downgrades it, then prints it.
/*  Demonstrates the use of functions with structures.
#include <stdio.h>

/*  Define structure to represent a hurricane.  */
struct hurricane
{
    char name[10];
    int year, category;
};

/*  Declare function prototypes.  */
void print_hurricane(struct hurricane h);
void downgrade_hurricane(struct hurricane *h_ptr);
struct hurricane get_info(void);
```

Function Arguments

```
int main(void)
{
    /*  Declare variables.  */
    struct hurricane h1;

    h1 = get_info();
    printf("\nHurricane information:\n");
    print_hurricane(h1);
    downgrade_hurricane(&h1);
    printf("\nAfter downgrading, hurricane information:\n");
    print_hurricane(h1);
    printf("\n");

    /*  Exit program.  */
    return 0;
}
```

Function Arguments

```
/*----------------------------------------------------*/
/*  This function prints the hurricane information.    */
void print_hurricane(struct hurricane h)
{
    printf("Hurricane: %s \n",h.name);
    printf("Year: %d, category: %d \n",h.year,h.category);

    return;
}
/*----------------------------------------------------*/

/*----------------------------------------------------*/
/*  This function downgrades the category by one.      */
void downgrade_hurricane(struct hurricane *h_ptr)
{
    if (h_ptr->category > 1)
        (h_ptr->category)--;
}
/*----------------------------------------------------*/
```

Function Arguments

```
/*-------------------------------------------------------------*/
/*  This function gets information from the user to enter       */
/*  into a hurricane structure.                                  */

struct hurricane get_info(void)
{
    /*  Declare variables.  */
    struct hurricane h;

    printf("Enter information for hurricane in this order:\n");
    printf("Enter name (no spaces), year, category:\n");
    scanf("%s %d %d", h.name, &h.year, &h.category);

    return h;
}
/*-------------------------------------------------------------*/
```

Array of Structures

- In engineering applications, it is often convenient to store information in arrays for analysis.
- However, an array can contain only a single type of information, such as an array of integers or an array of character strings.
- If we need to use an array to store information that contains different types of values, then we can use an array of structures.
- For example, if we want to store an array of information for hurricanes, we can use an array of the structure type `hurricane`.

Array of Structures

- We can write a statement to define an array of 25 elements, each of the type `hurricane`:
  - `struct hurricane h_arr[25];`
- Each element in the array is a structure containing the three variables, as illustrated in the following diagram:

`h_arr[0]`	?	?	?
`h_arr[1]`	?	?	?
...	...	...	...
`h_arr[24]`	?	?	?

Array of Structures

- To access an individual data member of a structure in the array, we must specify the array name, a subscript, and the data member name.

- As an example, we will assign values to the first hurricane in the array `h_arr`:

  -
    ```
    strcpy(h_arr[0].name,"Camille");
    h_arr[0].year = 1969;
    h_arr[0].category = 5;
    ```

Array of Structures

- To access an entire structure within the array, we must specify the name of the array and a subscript.

- As an example, we can call the output function `print_hurricane` to print the first hurricane in the array:

    - `print_hurricane(h_arr[0]);`

- We now present a program that reads all hurricane information from the file, downgrades all hurricanes, and prints out all hurricane information on the screen.

C Program Code

```
/*------------------------------------------------------------*/
/*  Program chapter7_4a                                        */
/*                                                             */
/*  This program reads the information for a hurricane         */
/*  from a data file and downgrades all the hurricanes.        */

#include <stdio.h>
#define FILENAME "storms2.txt"

struct hurricane
{
    char name[10];
    int year, category;
};

void print_hurricane(struct hurricane h);
void print_all_hurricanes(struct hurricane h_arr[], int npts);
void downgrade_hurricane(struct hurricane *h_ptr);
void downgrade_all_hurricanes(struct hurricane h_arr[], int npts);
```

Array of Structures

```
int main(void)
{
    int k=0, npts;
    struct hurricane h_arr[100];
    FILE *storms;
    /*  Read and print information from the file.  */
    storms = fopen(FILENAME,"r");
    if (storms == NULL)
        printf("Error opening data file. \n");
    else  {
        while (fscanf(
            storms,"%s %d %d",h_arr[k].name,&h_arr[k].year,
            &h_arr[k].category) == 3) {
            k++;
        }
        npts = k;
        printf("All Hurricanes Downgraded\n");
        downgrade_all_hurricanes(h_arr,npts);
        print_all_hurricanes(h_arr,npts);

        fclose(storms);
    }
    /*  Exit program.  */  return 0; }
```

Array of Structures

```
/*------------------------------------------------------------*/
/*  This function prints the hurricane information.          */
void print_hurricane(struct hurricane h)
{
    printf("Hurricane: %s \n",h.name);
    printf("Year: %d, category: %d \n",h.year, h.category);

    return;
}
/*------------------------------------------------------------*/

/*------------------------------------------------------------*/
/*  This function prints all hurricane information.          */
void print_all_hurricanes(struct hurricane h_arr[], int npts)
{
    int i;
    for (i=0; i<=npts-1; i++)
        print_hurricane(h_arr[i]);
}
/*------------------------------------------------------------*/
```

Array of Structures

```
/*-------------------------------------------------------*/
/*  This function downgrades the category by one.       */
void downgrade_hurricane(struct hurricane *h_ptr)
{
    if (h_ptr->category > 1)
        (h_ptr->category)--;
}
/*-------------------------------------------------------*/

/*-----------------------------------------------------------*/
/*  This function prints all hurricane information.          */
void downgrade_all_hurricanes(struct hurricane h_arr[], int npts)
{
    int i;
    for (i=0; i<=npts-1; i++)
        downgrade_hurricane(&h_arr[i]);
}
/*-----------------------------------------------------------*/
```

Summary of Structures

A structure is a construct that can group together variables of *different* types.
Eg. an "employee" record:

```
    struct emp {
        char lname[20];     /* last name */
        char fname[20];     /* first name */
        int age;            /* age */
        float rate;         /* e.g. $12.75 per hour */
    };
```

To obtain one struct variable:

```
struct emp my_struct;
int main(void)
{
    strcpy(my_struct.lname, "Jaffar");
    strcpy(my_struct.fname, "Joxan");
    my_struct.age = 21;
    my_struct.rate = 999;
}
```

Structures provide for Complex Data Structures

Structures can be arbitrarily complicated:

```
struct date { short day, month, year; }

struct emp {
    char lname[20];    /* last name */
    char fname[20];    /* first name */
    int age;           /* age */
    float rate;        /* e.g. $12.75 per hour */
    struct date starting_date;
    struct date previous_raise;
    int last_percent_increase;
    struct emp *manager; /* makes emp a SELF-REFERENTIAL struct */ }
```

NOTE: Wrong to use `struct emp manager;` to self-reference. Why?

Structure Assignment/Copy

```
struct emp { char name[20]; float rate; };
struct emp2 { char *name;  float rate; };

struct emp joxan;
struct emp2 joxan2;

int main(void) {
char str[] = "joxan2";
    strcpy(joxan.name, "joxan"); joxan.rate = 111;
    joxan2.name = str; joxan2.rate = 222;
    changename(joxan); changename2(joxan2);
    printf("%s %s\n", joxan.name, joxan2.name);
}
void changename(struct emp x) {
    strcpy(x.name, "wong");
}
void changename2(struct emp2 x) {
    strcpy(x.name, "wong2");
}
```

Output: joxan wong2

NOTE: copying a structure copies all the space allocated to the structure. In the structure `emp`, the attribute `name` is an array with allocated space 20 chars. In the structure `emp2`, the attribute `name` is allocated a *single* pointer.

Array of Structures

```
struct emp {
    char lname[20];    /* last name */
    char fname[20];    /* first name */
    int age;           /* age */
    float rate;        /* e.g. $12.75 per hour */
} database[1000];

main() {
    printf("%d %p %p\n",
        sizeof(struct emp), database, database + 1);
}
```

Output: 48 0x100001080 0x1000010b0

NOTE: pointer increment +1 translates into an address increment of 48.

Though arrays are intuitively the way to store a large collection of structures, in practice, such a collection is usually not stored this way. Instead, the structures are linked together using pointers. (Cf. next lecture on linked lists).

References

Etter Sections 7.1 to 7.4