STRUCTURES

(This text borrowed from Turbo C: The Essentials of C Programming, by Al Kelley & Ira Pohl, 1988; updated by Ruthann Biel, 2000.)

The structure type allows the programmer to aggregate components into a single, named variable. A structure has components that are individually named. These components are called members. The members of a structure can be various types, which allow the programmer to create aggregates of data that are suitable for a problem.

In this chapter we show how to declare structures and how to use them to represent a variety of familiar examples, such as a playing card or a student record. Critical to processing structures is the accessing of their members. This is done with either the member operator "." (We use a period, or as we call it, the “dot”) or the structure operator "->" (which we call “points into”). These operators, along with ( ) and { }, have the highest precedence. Please refer to your precedence chart.

Many examples are given in the chapter to show how structures are processed. An example that implements a student record system is given to show the use of structures and the accessing of its members.

S.1 DECLARING STRUCTURES

Structures are a means of aggregating a collection of data items of possible different types. As a simple example let us define a structure type that will describe a playing card. The spots on a card that represent its numeric value are called "pips." A playing card, such as the three of spades, has a pip value of 3, and a suit value of spades. We can declare the structure type

```
typedef struct
{    int   pips;
     char  suit;
} card_t;
```

to capture the information needed to represent a playing card.

In this declaration, *typedef* (short for type definition and on the keyword or reserved-word chart) is setting up a unique user-defined type, instead of system-defined types like *int* and *double*.

struct (also a keyword) is setting up an outline or a template for our data collection.
card_t is the user-defined identifier or name for the new type.

The "_t" suffix is common engineering practice to indicate to the reader that a user-defined type is being referred to. The variables pips and suit are members of the structure. The variable pips will

take values from 1 to 13 representing ace to king, and the variable suit will take values from 'c', 'd', 'h', 's', representing the suits clubs, diamonds, hearts, and spades.

This declaration creates the user-defined data type struct *card_t*. The declaration can be thought of as a template; it creates the type *card_t*, but no storage is allocated. The name can now be used to declare variables of this new type. The declaration

```
card_t c1,c2;
```

allocates storage for the identifier-variables *c1* and *c2,* which are of type *card_t*. To access the members of *c1* and *c2* we use the structure member operator "." (dot). Suppose that we want to assign *c1* the values representing the five of diamonds and to *c2* the values representing the queen of spades. To do this we can write

```
c1.pips = 5;
c1.suit = 'd';
c2.pips = 12;
c2.suit = 's';
```

A construct of the form

```
structure_variable.member_name
```

is used as a variable in the same way a simple variable or an element of an array is used. The member name must be unique within the specified structure. Since the member must always be prefaced or accessed through a unique structure variable identifier, there is no confusion between two members having the same name in two or more different structures. An example is

```
typedef struct
{    char   name[15];
     int    calories;
} fruit_t;

typedef struct
{    char   name[15];
    int    calories;
} vegetable_t;

fruit_t      a;
vegetable_t   b;
```

Having made these declarations, we can access *a.calories* and *b.calories* without ambiguity. With the last two lines, the storage is actually allocated for the variables *a* and *b*. The template of the structure type by itself does not cause storage to be allocated.

It is possible to declare structures without doing a type definition but it is no longer a usual engineering practice.

S.2 ACCESSING A MEMBER

We have already seen the use of the member operator ".". In this section we give further examples of its use and introduce the structure pointer operator ->.

Suppose that we are writing a program called *class_info*, which generates information about a class of 100 students. We can begin creating the start of this program.

```
#define CLASS_SIZE  100

typedef struct
{   char  *last_name;
    int   student_id;
    char  grade:
} student_t;

int main (void)
{
    student_t temp;
    student_t class[CLASS_SIZE];
```

We can assign values to the members of the structure variable *temp* by using the statements such as

```
temp.grade = 'A';
temp.last_name = "Wootten";
temp.student_id = 590017;
```

Now suppose that we want to count the number of failing students in a given class. To do this, we can write a function that accesses the grade member. Here is a function *fail* that counts the number of F grades in the array *class*[].

```
int fail(student_t class[CLASS_SIZE])
{
    int i, count = 0;

    for (i = 0; i < CLASS_SIZE;  i++)
        if (class[i].grade == 'F')
            count ++;

    return count;
}
```

DISSECTION OF THE *fail* FUNCTION:

int fail(student_t class[CLASS_SIZE])

The parameter class is of type *student_t*. It is a one dimensional array of structures. Parameters of any type, including structure types, can be used in function definitions.

for (i = 0; i < CLASS_SIZE; i++)

We are assuming that when this function is called, an array of the size of *CLASS_SIZE* will be passed as an argument.

```
if (class[i].grade == 'F')
   count ++;
```

The test is made to see if *grade* of the i-th element (counting from zero) of the array of structures *class* is equal to 'F'. If equality exists, then the value of the *count* is incremented by one.

return count;

The number of failing grades is returned to the calling environment.

C provides the structure pointer operator -> to access the members of a structure via a pointer. This operator is typed on the keyboard as a minus sign followed by a greater than sign. If a pointer variable is assigned the address of a structure, then a member of the structure can be accessed by a construct of the form

```
pointer_to_structure -> member_name
```

An equivalent construct is given by

```
(*pointer_to_structure).member_name
```

The parentheses are necessary here. The operators -> and ".", along with ( ) and [ ], have the highest precedence, and they associate "left to right". Because of this, the above construct without parentheses would be equivalent to

```
*(pointer_to_structure.member_name)
```

In complicated situations the two accessing modes can be combined in complicated ways. The following table illustrates their use in a straightforward manner.

Declarations and assignments		
`student_t temp, *p = &temp;`		
`temp.grade = 'A';`		
`temp.last_name = "Wootten";`		
`temp.student_id = 590017;`		
EXPRESSION	**EQUIVALENT EXPRESSION**	**CONCEPTUAL VALUE**
`temp.grade`	`p -> grade`	`A`
`temp.last_name`	`p -> last_name`	`Wootten`
`temp.student_id`	`p -> student_id`	`590017`
`(*p).student_id`	`p -> student_id`	`590017`

S.4 STRUCTURES, FUNCTIONS, AND ASSIGNMENT

To illustrate the use of structures with functions, we will use the structure type *card_t.* For the remainder of this section assume this structure.

```
typedef struct
{   int   pips;
    char  suit;
} card_t;

card_t card;
```

Let us write functions that will assign values to a card, extract the member values of a card, and print the values of a card. We will assume that the definitions of *card_t* and *card* have been included, perhaps in a header file, wherever needed.

```
void assign_values(card_t *c_ptr, int p, char s)
{
  c_ptr -> pips = p;
  c_ptr -> suit = s;
  return;
}
```

```
void extract_values(card_t *c_ptr, int *p_ptr, char *s_ptr)
{
  *p_ptr = c_ptr -> pips;
  *s_ptr = c_ptr -> suit;
  return;
}
```

These functions access a *card* by using a pointer to a variable of type *card_t*. The structure pointer operator -> is used throughout to access the required member. Next, let us write a card printing routine that takes a pointer to *card* and prints its values using function *extract_values*.

```
void prn_values(card_t *c_ptr)
{
   int   p;          /* pips value */
   int   s;          /* suit value  */
   char *suit_name;
   void extract_values(card_t *c_ptr,
                       int *p_ptr,
                       char *s_ptr);

   extract_values(c_ptr, &p, &s);
   if (s == 'c')
      suit_name = "clubs";
   else if (s == 'd')
      suit_name = "diamonds";
   else if (s == 'h')
      suit_name = "hearts";
   else if (s == 's')
      suit_name = "spades";
   else
      suit_name = "error";

   printf("card:  %d of %s \n", p, suit_name);
}
```

| PS: An alternate way to code the if-else-if using the conditional operator follows.
|

```
|   suit_name = (s == 'c') ? "clubs" :
|                (s == 'd') ? "diamonds" :
|                (s == 'h') ? "hearts" :
|                (s == 's') ? "spades" :
|               "error";
```

Finally, we want to illustrate how these functions can be used. First, we assign values to a deck of cards, and then as a test, we print out the heart suit.

```
int main (void)
{
    int     i;
    card_t  deck[52];
    void  assign_values(card_t *c_ptr, int p, char s);
    void  prn_values(card_t *c_ptr);

    for (i = 0; i < 13;  i++)
    {
        assign_values(deck + i, i + 1, 'c');
        assign_values(deck + i + 13, i + 1, 'd');
        assign_values(deck + i + 26, i + 1, 'h');
        assign_values(deck + i + 39, i + 1, 's');
    }
    for (i = 0; i < 13; i++)   /* print out the hearts */
        prn_values(deck + i+26);
}
```

Functions can be designed to work with structures as parameters, rather than with pointers to structures. To illustrate this, let us rewrite the functions *assign_values* and *extract_values*.

```
card_t assign_values(int p, char s)
{
    card_t c;

    c.pips = p;
    c.suit = s;
    return (c);      /* now the function returns a struct */
}

void extract_values(card_t c, int *p_ptr, char *s_ptr)
{
    *p_ptr = c.pips;
    *s_ptr = c.suit;
     return;
}
```

In C the value of an argument that is passed to a function is copied when the function is invoked. This call-by-value mechanism has been discussed before. (In CSC25 we have been calling it "original copy passing".) Because of this, when a structure is passed as an argument to a function,

the structure is copied when the function is invoked. For this reason, passing the address of the structure is more efficient that passing the structure itself.

S.5 AN EXAMPLE: STUDENT RECORDS

The variations available in C to define complicated data structures involve all meaningful combinations of structure, pointer, and array. We will start with our previous example of *student_t* and develop it into a more comprehensive data structure for a student record. We begin by defining the various types needed, putting them in the file "*student.h*", as follows:

```
#define  CLASS_SIZE  50
#define  NCOURSES    10   /* number of courses */

typedef struct
{    char *last_name;
     int    student_id:
    char grade;
}student_t;

typedef struct
{   int    day;
    char  month[10];
    int     year;
}date_t;

typedef struct
{   char      name[20];
    date_t   birthday;
}person_t;

typedef struct
{   person_t  p;
    int           student_in;
    char         grade[NCOURSES];
}student_data_t;
```

Notice that *student_data_t* is constructed with nested structures. One of its members is the structure *p*, which has as one of its members the structure *birthday*. After the declaration

```
student_data_t  temp;
```

has been made, the expression

```
temp.p.birthday.month[0]
```

has as its value the first letter of the month of the birthday of the student whose data is in temp. Structures such as date and person are used in data base applications.

Let us write the function *read_date* to enter data into a variable of type *date_t*. When the function is called, the address of the variable must be passed as an argument to the function.

```
#include  "student.h"

void read_date(date_t d)
{
    printf("Enter  day(int)  month(string)  year(int): ");
    scanf("%d%s%d", &d -> day, d -> month, &d -> year);
    return;
}
```

DISSECTION OF THE read_date FUNCTION

```
void read_date(date_t d)
{
    printf("Enter  day(int)  month(string)  year(int): ");
```

The parameter d has type "pointer to *date_t*". The *printf( )* statement prompts the user for information.

&d -> day

This is an address. Because **&** is of lower precedence than **->,** this expression is equivalent to &(day -> day)

First the pointer *d* is used to access the member day. Then the address operator **&** is applied to this member to obtain its address.

d -> month

This is an address. The pointer *d* is being used to access a member that is an array. An array name by itself is a pointer, or an address. It points to the base address of the array.

scanf("%d%s%d", &d -> day, d -> month, &d -> year);

The function *scanf( )* is used to read in three values and to store them at appropriate addresses. Recall that in the header file *student.h* the two members *day* and *year* of *date_t* were declared to be of type *int*.

The function *read_date* can be used to read information into a variable of type *student_data_t*. For example, the code

```
student_data_t  temp;
read_date(&temp.p.birthday);
```

can be used to place information into the appropriate member of *temp*.

Here is a function to enter grades.

```
void read_grades(char g[])
{
    int  i;

   printf("Enter %d grades:  ", NCOURSES);
    for (i = 0; i < NCOURSES; i++)
        scanf(" %c", &g[i];

    return:
}
```

The control string " **%c**" is being used to read in a single nonwhite space character. The blank just before the % matches optional white in the input stream. This function could be called to read a list of grades into *temp* as follows:

```
read_grades(temp.grade);
```

The argument *temp.grade* is an address (pointer) because it refers to a member of a structure that is an array, and an array name by itself is the base address of the array. Thus when the function is invoked, it causes the values of *temp.grade* in the calling environment to be changed.

Basically, understanding structures comes down to understanding how to access their members. As a further example let us now write a function that takes data stored in the long form in *student_date_t* and coverts it to the short form stored in *student_t*.

```
#include  "student.h"
void extract (student_data_t *s_data,
              int n,              /* course number */
              student_t *undergrad)
{
    undergrad -> student = s_data -> student_id;
    undergrad -> last_name = s_data -> p.name;
    undergrad -> grade = s_data -> grade[n];
```

```
    return;
}
```

S.6 INITIALIZATION OF STRUCTURES

A structure variable can be followed by an equal sign "=" and a list of constants contained with braces. If not enough values are used to assign all the members of the structure, the remaining members are assigned the value zero by default. Some examples are

```
card_t c = {12,'s'};  /* the queen of spades */

static fruit_t  frt = {"plum", 150};

typedef struct
{     double  real;
      double  imaginary;
{ complex_t;

complex_t m[3][3] = {{{1.0, -0.5}. {2.5, 1.0}, {0.7, 0.7}},
                     {{7.0, -6.5}, {-0.5, 1.0}, {45.7, 8.0};          /* m[2][ ] is assigned zeroes */
```