C-9 Structures

Further information available
on Canvas under Reference
File = **structs.pdf**

Record: data *structure* that stores different types of data under a single variable name.

In C, we use the keyword ***struct*** to define records. They can contain *components* (also called *members* or *fields)* that can have different types.

Record

Buzz word. A *record* is one line in a data base.

Example of a Parts Warehouse:

For any given Part, there might be information stored about it in a **record** :

- part number
- its cost
- amount in stock
- location in warehouse
- name of supplier

Reminder: what do we mean by “type”?

Examples of types we have used:

- int
- unsigned
- float
- double
- char

So now we are moving on to creating our **own** types!

(and then, on to multi-part types.)

typedef - a mechanism which allows the programmer to explicitly associate a **type** with an **identifier**.

Example 1:

```
typedef  int  Length_t;

Length_t len, maxlen;
```

Example 2:

```
typedef int Inches_t, Feet_t;

Inches_t  box_length, box_width;

Feet_t lot_width, lot_length;
```

Defining Structure Types

A new **type definition** can be defined for the structure, which can then be used to declare variables:

```
typedef struct
{
    int month;
    int day;
    int year;
} date_t;

date_t birth, current;    /* This line declares two type
                             date_t variables */
```

The declaration should list each member on its own line, properly indented.

```
typedef struct
{
    int month;
    int day;
    int year;
} date_t;
```

Common Industry Practice:

The **"_t"** suffix is not required by C
but certainly makes it easier to keep a program readable.

It is very common in industry & that's what we will use in this class.

(Some companies use all caps.)

Initialization of structs:

Structures can be initialized when declared by putting the values, in the correct order, inside brackets { }.

```
date_t  birth     = {3, 13, 1989};

date_t  current = {9, 23, 2017};
```

A structure can contain data items of different types:

```
/* First set up the structure */
        typedef struct
        {
              char     id[20];
              double  price;
              int       current_inv;
        }auto_part_t;

/* This declares variable part1 of type auto_part_t */

        auto_part_t   part1;

/*These 3 lines initialize the parts of part1 */

        strcpy (part1.id , "A45X");      // string copy function
        part1.price = 10.60;
        part1.current_inv = 23;
```

Structures within Structures

```
typedef struct {
   int month;
   int day;
   int year;
} date_t;              /* This sets up the structure date_t */

typedef struct {
    char name[20];
    date_t birth;
} person_t;            /* This sets up the structure person_t  */

person_t person;   /* Initialize a variable person of type person_t */
```

To reference the data items in the structure:

person.name
person.birth.month
person.birth.day
person.birth.year

/* Or similarly create a person type */

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

/* Declare 2 variables of type person_t */

```
person_t  pers1, pers2;
```

/* To reference the data items *pers1* & *pers2* of type person_t */

```
pers1.name                pers2.name
pers1.birth.month         pers2.birth.month
pers1.birth.day           pers2.birth.day
pers1.birth.year          pers2.birth.year
```

Arrays of Structures

Declaring an array of structures is the same as declaring an array of any other type of variable.

```
typedef struct
{
    int idnum;          /* employee id */
    char name[20];      /* employee name */
    float rate;         /* employee pay rate */
} pay_rec_t;            /* structure for one employee */
```

Define an array of 10 employees of type pay_rec_t:

```
pay_rec_t  employee[10];
int        counter;
double     average;
```

```
/* A program using struct & typedef */

#include <stdio.h>
#include <stdlib.h>
#define MAXNAME 30
#define NUMRECS 5

typedef struct                  /* a global type definition */
{
   long id;                     /* employee id */
   char name[MAXNAME];          /* employee name */
   double rate;                 /* employee pay rate */
} pay_rec_t;
…
```

```
int main(void)
{
    int j;
    pay_rec_t employee[NUMRECS] = {
                        { 32479, “Abrams, B.”, 6.72 },
                        { 33623, “Bohm, P.”, 7.54 },
                        { 34145, “Donaldson, S.”, 5.56 },
                        { 35987, “Ernst, T”, 5.43 },
                        { 36203, “Gooding, K.”, 8.73 }};

    for (j = 0; j < NUMRECS; j++)
    {
        printf(“%li %s  %4.2f \n”, employee[j].id,
                employee[j].name, employee[j].rate);
    }
    return EXIT_SUCCESS;
}
```

The output would be:

32479 Abrams, B. 6.72
33623 Bohm, P. 7.54
34145 Donaldson, S. 5.56
35987 Ernst, T. 5.43
36203 Gooding, K. 8.73

Structures and Functions

Structures and Functions

```
typedef struct
{
        int hour, minute, second;
} time_t;
```

- -

function prototype:

```
time_t  new_time (time_t time_of_day, int elapsed_secs);
```

- -

(example next slide)

First, an example:

Suppose the **current** time is 21:58:32 and **elapsed** time is 97 seconds.
What is the sum of the two times?
Call the function from **main,** which has the following declarations:

```
time_t time_now = {21, 58, 32};
int secs = 97;
```

Similarly could assign values as follows:

```
time_t time_now;
int secs;

time_now.hour = 21;
time_now.minute = 58;
time_now.second = 32;
secs = 97;

time_now = new_time(time_now, secs);
```

new_time would return a value of 22:00:09

/*Here is the **function code** that would work as in the previous example. */

```
time_t  new_time (time_t time_of_day, int elapsed_secs)
{
    int new_hr,  new_min,  new_sec;

    new_sec                 = time_of_day.sec + elapsed_secs;
    time_of_day.sec = new_sec % 60;

    new_min                 = time_of_day.min + new_sec / 60;
    time_of_day.min = new_min % 60;

    new_hr                = time_of_day.hr + new_min / 60;
    time_of_day.hr = new_hr % 24;

    return(time_of_day);
}
```

Structures as Function Arguments

Individual structure members may be passed to a function in the same manner as any scalar (or non-array) variable.

For example, given the structure definition:

```
typedef struct
{
    int id_num;
    double pay_rate;
    double hours;
} emp_t;
```

```
emp_t emp;    /* declaration of a variable emp */
```

the statement

```
display(emp.id_num);
```

passes a copy of the structure member emp.id_num to a function called **display( );**

Similarly,

calc_pay(emp.pay_rate, emp.hours);

passes copies of *emp.pay_rate*
and *emp.hours*
to a function to calculate the amount of pay owed the employee.

A copy of the complete structure can also be passed to a function:

```
calc_net(emp);
```

passes a copy of the entire *emp* structure
to the function calc_net( ).

```
/* another example */

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

typedef struct                  /* global type definition */
{
    int id_num;
    double pay_rate;
    double hours;
} employee_t;

double calc_net(employee_t temp);  /* function prototype*/

…
```

```
…
int main(void)
{
    employee_t emp = {6782, 8.93, 40.5};
    double net_pay;
    net_pay = calc_net(emp);
    printf("The net pay for employee %i is $%6.2f \n\n",
                    emp.id_num, net_pay);
    return EXIT_SUCCESS;
}
*/-------------------------------------------------------*/
double calc_net(employee_t temp)
{
    return (temp.pay_rate * temp.hours);
}
*/-------------------------------------------------------*/
```

The output is:

The net pay for employee 6782 is $361.66

POINTERS AND STRUCTS

When we use *pointers* and *structs* together we use:

the *address operator* (&)

the *indirection operator* (*)

We also add a **new operator**, the *structure pointer operator* **(->)**

(a minus sign followed by a greater-than sign).

In general practice we use this new pointer operator
when we are using a pointer to values not in an array,
in place of the dot notation (.) that we have been using to get
into a *struct*.

The structure pointer operator is illustrated in the code that follows.

Passing a Structure to a Function as a Pointer

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

typedef struct                    /* global type definition */
{
    int id_num;
    double pay_rate;
    double hours;
} employee_t;

double calc_net(employee_t *e);  /* function prototype*/

…
```

```
int main(void)
{
    employee_t emp = {6782, 8.93, 40.5};
    double net_pay;

    net_pay = calc_net(&emp);
    printf("The net pay for employee %i is $%6.2f \n\n",
            emp.id_num, net_pay);
    return EXIT_SUCCESS;
}
/*-------------------------------------------------*/
double calc_net(employee_t *e)
{
    return (e→pay_rate * e→hours);
}
/*-------------------------------------------------*/
```

The output is:

```
        The net pay for employee 6782 is $361.66
```

Struct Return Types for Functions

```
#include <stdio.h>
#include <stdlib.h>
typedef struct              /* global type definition */
{
    int id_num;
    double pay_rate;
    double hours;
} employee_t;

// More on next slide
```

```
int main(void)
{
    employee_t emp;
    employee_t get_vals(void);   /* function */
    emp = get_vals( );
    printf("The employee id number is %i \n",
        emp.id_num);
    printf("The employee pay rate is $%6.2f \n",
        emp.pay_rate);
    printf("The employee hours are %4.1f \n",
        emp.hours);
    return EXIT_SUCCESS;
}
```

```
/*-------------------------------------------------------*/
/* This function returns an employee structure */
employee_t  get_vals (void)
{
    employee_t  one;
    one.id_num   = 6789;
    one.pay_rate = 16.25;
    one.hours    = 40.0;
    return (one);
}
/*-------------------------------------------------------*/
```

Output:

```
The employee id number is 6789
The employee pay rate is $16.25
The employee hours are 40.0
```

You can read more helpful material in **structs Information** Which is on Canvas under Reference Materials.

It is a chapter that is eleven pages, and has very good examples of *structs* and *typedef*.

Rule of Thumb:

If the struct comes into a function as an **array**,
use the *dot notation* **(.)**.

If the struct comes into a function as a **pointer** with an *****,
use the *points-into notation* **(->)**.

Extra material on Enumeration Types

enum - keyword - used to declare enumeration types.
provides a means of naming a finite set,
and declaring identifiers as elements of the set.

Declare a type named *day*

```
enum day {sun, mon, tue, wed, thu, fri, sat };
```

Declare variables *d1* and *d2* of type enum day

```
enum  day  d1, d2;

d1 = fri;                    /* allowed */

if (d1 == d2)…               /* allowed */
```

```
/* Compute the next day using function find_next_day */

enum day {sun, mon, tue, wed, thu, fri, sat };

typedef enum day  day_t:

day_t find_next_day (day_t d)
{
        day_t  next_day;
        switch (d) {
        case sun:
                next_day = mon;
                break;
        case mon:
                next_day = tue;
                break;
        …… /* and so on */
        return next_day;
}
```

```
/* A Second Version of the same function                    */
/* Compute the next day using function find_next_day */

enum day {sun, mon, tue, wed, thu, fri, sat };

typedef enum day  day_t:

day_t find_next_day (day d)
{
        return (day) (((int) d + 1) % 7);
}
```

The values in ***day*** are constants of type *int.*
By default, the first item is assigned a zero,
and each succeeding one has the next integer value.

/*The default value of zero can be changed. */

enum day {sun=1, mon, tue, wed, thu, fri, sat };

/* Now **sun** starts as 1, and the group go from 1 to 7. */

Another example:

```
enum trees
        { oak, maple, cherry, spruce, pine};

enum trees tree1;
```

Or combine the previous two lines into one line:

```
enum trees
        { oak, maple, cherry, spruce, pine
} tree1;
```

Same example, using typedef:

```
typedef enum
        { oak, maple, cherry, spruce, pine }
trees_t;

trees_t tree1;
```

C-9 Structures

The End