ITC213: STRUCTURED PROGRAMMING

Bhaskar Shrestha
National College of Computer Studies
Tribhuvan University

Lecture 12: Structures

Readings: Chapter 11

Structures (1/2)

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling
- Structures are called “records” in some languages, notably Pascal
- Structures help to organize complicated data, particularly in large programs
- They permit a group of related variables to be treated as a unit instead of as separate entities

Structures (2/2)

- A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together
- Structures are one of the ways to create a custom data type in C
- Variables that make up the structure are called ***members***
  - Also commonly referred to as ***elements*** or ***fields***
- Individual members of a structure can be any of C’s data types
  - including arrays, pointers and other structure variables

Structure Declarations

- The first thing you need to do while using structures is to declare a structure type
- A ***structure declaration*** describes a template or shape of a structure
  - It defines the type and name of all of its members it is going include
- A structure declaration is identified by the `struct` keyword, followed by an identifier known as ***structure tag***
  - This ***structure tag*** is later used to define structure variables

Example of structure declaration

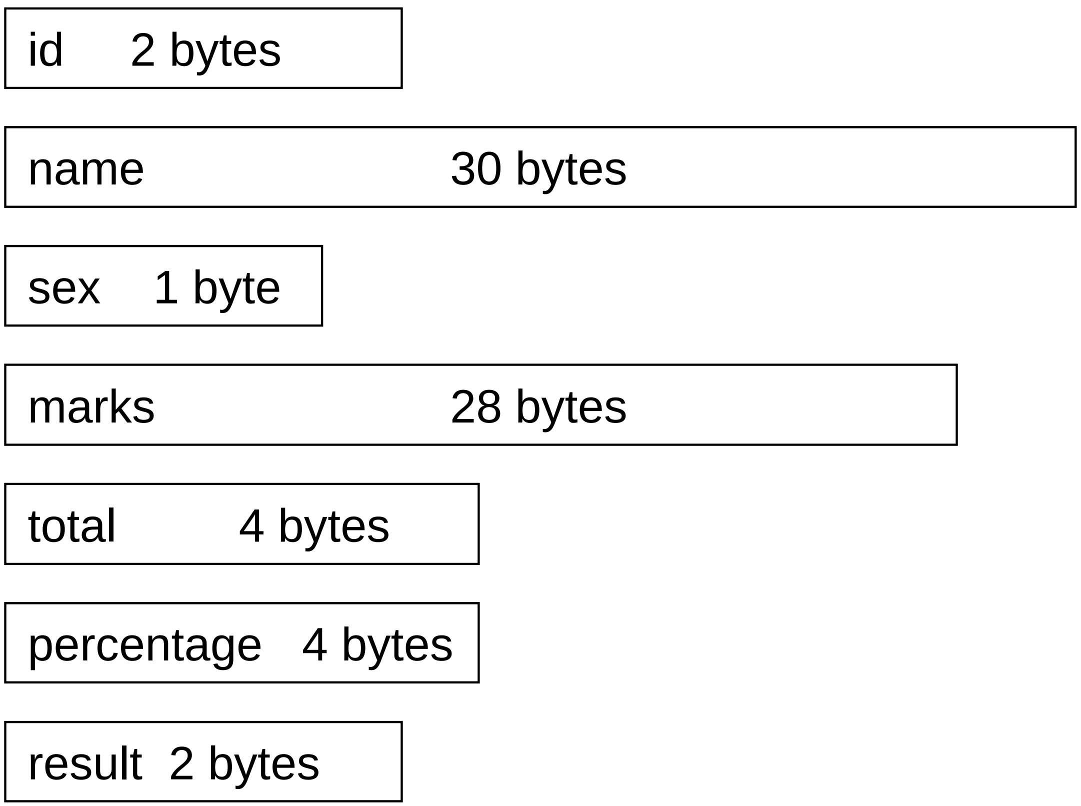
- A structure declaration to represent information about a student's name and other details
- ```
  struct student
  {
    int id;
    char name[30];
    char sex;
    float marks[7];
    float total;
    float per;
    int result;
  };
  ```
- Here, `struct` is the required keyword and `student` is the structure tag used to identify this structure
  ```

Structure Variables

- A structure declaration only defines the form of the data; it does not allocate memory
- A ***structure variable*** is a variable of a structure type
- To declare a variable of type `student`, defined earlier, write
  - `struct student st;`
  - This declares a variable of type `struct student` called `st`. Thus, `student` describes the form of a structure (its type), and `st` is an instance (a variable) of the structure

Structure Variable in Memory

- When a structure variable (such as `st`) is declared, the compiler automatically allocates sufficient memory to accommodate all of its members



The structure variable `st` in memory. Although not shown, all the members are stored contiguously in memory

More on Structure Declarations (1/2)

- You can also declare one or more variable when you declare a structure. For example,
  - ```
    struct student
    {
      int id;
      char name[30];
      char sex;
      float marks[7];
      float total;
      float per;
      int result;
    } a, b, c;
    ```
  - defines a structure type called `student` and declares three variables `a`, `b` and `c` of that type
- Each structure variable (`a`, `b`, and `c`) contains its own copies of the structure's members
    ```

More on Structure Declarations (2/2)

- If you only need one structure variable, the structure tag can be omitted. For example,
  - ```
    struct
    {
      int id;
      char name[30];
      char sex;
      float marks[7];
      float total;
      float per;
      int result;
    } st;
    ```
  - declares one variable named `st` as defined by the structure preceding it
    ```

General Form of Structure Declaration

- The general form of a structure declaration is
  - ```
    struct tag
    {
        type member-name;
        type member-name;
        type member-name;
        .
        .
        .
    } structure-variables;
    ```
  - where either `tag` or `structure-variables` may be omitted, but not both
- ***A structure member or tag and an ordinary variable can have the same name without conflict***
- The same member names may occur in different structures
    ```

Structure Declaration Examples

```
/* declare a structure template to represent a bank account */
struct bankaccount
{
    int acct_no; /* account no. */
    char acct_type; /* type of account:
            'C' for current, 'S' for savings, 'F' for fixed */
    char name[80]; /* account holder's name */
    float balance; /* the current balance */
};
/* now declare variables of type struct bankaccount */
struct bankaccount myaccount, friendsaccount;
/* myaccount and friendsaccount are structure variables */
```

```
/* declare a structure and instance together */
struct date
{
    int m, d, y;
} current_date; /* current_date is a structure variable */
```

Operations on Structure Variables

- The only legal operations on structure variables are
  - copying it or assigning to it as a unit
  - taking its address with &, and
  - accessing its members
- Besides these, no other operations on structure variables are defined
  - For e.g., structures variables may not be compared

Accessing Structure Members

- The members of a structure are usually processed individually, as separate entities
- A structure member can be accessed by writing
  - `variable.member`
  - where *`variable`* refers to the name of structure-type variable, and *`member`* refers to the name of a member within a structure
- The ***period*(.)** that separates the structure variable name from the member name is called the ***dot*** or ***structure-member operator***

Example on Accessing Structure Members

- The following statement assigns the id `1001` to the `id` member of the structure variable `st` declared earlier:
  - `st.id = 1001;`
- Therefore, to print the `id` of `st` on the screen, write
  - `printf("%d", st.id);`
- The character array `st.name` can be used in a call to `gets()`, as shown here:
  - `gets(st.name)`
  - This passes a character pointer to the start of `name` member of the structure variable `st`

More Examples

- Since `name` is a character array, you can access the individual characters of `st.name` by indexing `name`
- For example, you can print the contents of `st.name` one character at a time by using the following code:
  - ```
    for(i = 0; st.name[i] != '\0'; ++i)
     putchar(st.name[i]);
    ```
  - Notice that it is `name` (not `st`) that is indexed
- The code fragment calculates the total marks obtained by `st`:
  - ```
    st.total = 0;
    for (i = 0; i < 7; i++)
     st.total += st.marks[i];
    ```

The dot operator

- The **dot operator** is a member of the highest precedence group, thus it will take precedence over the unary operators as well as the various arithmetic, relational, logical and assignment operators
- Thus, an expression of the form
  - **++*variable.member*** is equivalent to **++(*variable.member*)**
  - **&*variable.member*** is equivalent to **&(*variable.member*)**
    - accesses the address of the structure member, not the starting address of the structure variable

More Examples

Expression	**Interpretation**
`++st.id`	Increment the value of `st.id`
`st.id++`	Increment the value of `st.id` after accessing its value
`--st.id`	Decrement the value of `st.id`
`&st`	Access the beginning address of `st`
`&st.id`	Access the address of `st.id`
`&st.marks[2]`	Access the address of third element of `st.marks`

Structure Assignments (1/2)

- Members contained in one structure variable can be assigned to another structure variable of the same type using a single assignment
- You do not need to assign the value of each member separately
- For example, consider the following structure declaration:
- ```
  struct bankaccount
  {
      int acct_no;
      char acct_type;
      char name[80];
      float balance;
  };
  ```
- Now if you declare two variables of type `struct bankaccount` as shown:
  - `struct bankaccount newaccount, oldaccount;`
  ```

Structure Assignments (2/2)

- The following statement causes each member of `oldaccount` to be assigned to the corresponding member of `newaccount`:
  - `newaccount = oldaccount; /* assign one structure to another */`
- This has the effect of copying each member individually, as shown:
  - ```
    newaccount.acct_no = oldaccount.acct_no;
    newaccount.acct_type = oldaccount.acct_type;
    strcpy(newaccount.name, oldaccount.name);
    newaccount.balance = oldaccount.balance;
    ```
  - Note that `name` is an array of characters representing a string. You must use `strcpy()` to copy strings
    ```

Nested Structures

- Recall, members of a structure can be variables of any of the valid data types
- Since structure is a custom type, you can define structure variable as a member of another structure

```
struct time {
    int hrs, mins;
};
struct date {
    int m,d,y;
};
struct flightschedule {
    int flightno;
    struct time departuretime;
    struct time arrivaltime;
    struct date scheduledate;
};
```

Here the structure variables `departuretime`, `arrivaltime`, and `scheduletime` are members of the structure `flightschedule`. These are said to be nested within the structure `flightschedule`. The declaration of `time` and `date` must precede the declaration of `flightschedule`.

Accessing Nested Structure Members

- Now suppose, you declare a structure variable named `myflight` of type `flightschedule` as shown:
  - `struct flightschedule myflight;`
- To access the `hrs` of `departuretime` member of `myflight`, you must apply the dot operator twice
- For example, the following statement assigns `9` to `hrs` member of `departuretime`:
  - `myflight.departuretime.hrs = 9;`
- Moreover, this value can be incremented by writing
  - `++myflight.departuretime.hrs`

Initializing Structure Variables

- To initialize structure variables, list the values for the individual members separated by commas and enclosed in braces
- The initial values must appear in the order in which they will be assigned to their corresponding structure members

```
struct bankaccount
{
   int acct_no;
   char acct_type;
   char name[80];
   float balance;
} myaccount = {1001, 'C', "Binod Chapagain", 12000.0};
```

`myaccount` **is a structure variable of type** `bankaccount`**, whose members are assigned initial values.** `acct_no` **is assigned the integer value** `1001`**,** `acct_type` **is assigned the character** `'C'`**,** `name[80]` **is assigned the string** `"Binod Chapagain"`**, and** `balance` **is assigned** `12000.011`

Array of Structures

- Since you can create an array of any valid type, it is possible to define an array of structures; i.e., an array in which each element is a structure
- To declare an array of structures, you must declare a structure and then declare an array variable of that type

```
struct student
{
    int id;
    char name[30];
    float marks[7];
    float total;
    float per;
    int result;
};

/* array of structures */
struct student studlist[100];
```

This creates 100 sets of variables that are organized as defined in the structure `student`. Each element in the array `studlist` is a structure of type `student` and is identified by subscript like other array element types

To access a specific structure, you index the array name, `studlist`. For example to print the `id` of 3rd student, write:

```
printf("%d", studlist[2].id);
```

typedef

- The `typedef` keyword is used to create a synonym or alias of an existing data type
- This process can help make machine-dependent programs more portable
  - If you define your own type name of each machine-dependent data type used by you program, then only the `typedef` statements have to be changed when compiling for new environment
- The general form of the `typedef` statement is
  - `typedef` *`type newname`*`;`
  - where *`type`* is any valid type, and *`newname`* is the new name for the new name for this *`type`*. The new name you defined is in addition to, not a replacement for, the existing type name

typedef Examples

- You could create a new name for `float` by using
  - `typedef float balance;`
  - Now the compiler will recognize balance as another name for `float`
- Next, you could create a `float` variable using `balance`:
  - `balance over_due;`
  - Here, `over_due` is a floating-point variable of type `balance`, which is another word for `float`
- Now that `balance` has been defined, it can be used in another `typedef`. For example,
  - `typedef balance overdraft;`
  - tells the compiler to recognize `overdraft` as another name for `balance`, which is another name for `float`

More Examples

- The declarations
  - ```
    typedef float height[100];
    height men, women;
    ```
  - defines `height` as a 100-element, floating-point array type—hence, `men` and `women` are 100-element, floating-point arrays
- Another way to express this is
  - ```
    typedef float height;
    height men[100], women[100];
    ```
  - though the former declaration is somewhat simpler
- You can also use `typedef` for pointers
  - ```
    typedef int * iptr;
    iptr p;
    ```
  - defines `iptr` as an integer pointer, hence `p` is an integer pointer
    ```

typedef and Structures

- The `typedef` feature is particularly convenient when defining structures
  - it eliminates the need to repeatedly write the `struct tag` whenever a structure is referenced
- The following statements define `traveltime` as a synonym for the indicated structure

```
– struct time
  {
   int hrs;
   int mins;
  };
  typedef struct time traveltime;
```

- Now `traveltime` can be used in place of `struct time`. Hence, the following two declarations are equivalent.

```
– struct time day1;
  traveltime day1; /* same as preceding statement */
```

More typedef Structures

- You can use the `typedef` keyword to create a synonym for `struct tag` within the structure declaration
- In general terms, a user-defined structure type can be written as
  - ```
    typedef struct tag
    {
        member1;
        member2;
        ...;
        member1;
    } newname;
    ```
  - where `newname` is the user-defined structure type. The `tag` is optional in this case

```
typedef struct time
{
    int hrs;
    int mins;
} traveltime;
```

Now, `traveltime` is another name for `struct time`. The structure *tag* `time` could have been omitted
```

Passing Structure Members to Functions

- Individual structure members can be passed to functions
- When a structure member is passed to the function, the value of the member is passed
- In the function, each structure member is treated the same as an ordinary single-valued variable

```
float adjust(char name[], int acct_no, float balance, char type)
{
   ...
}

main()
{
   struct bankaccount acc;
   ...
   /* pass individual members to the function */
  acc.balance = adjust(acc.name,acc.acct_no,acc.balance,acc.balance);
   ...
}
```

Note for `acc.name`, it is the address of `acc.name[0]` that is passed. For other members, it is the value of the members that are passed to function. It is irrelevant in the function `adjust` that structure members were passed

Passing Structure Variables to Functions

- A single structure variable can be passed to functions at once
- A structure variable is passed to a function using the normal call-by-value mechanism
  - This means that a copy of the structure variable is passed and any changes made to the contents of the parameter inside the function do not affect the structure passed as the argument
- When a structure variable is passed to a function, the formal parameter in the function must be declared as the same structure type passed

```
struct bankaccount
{
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
};

void ShowAccount(struct bankaccount acc);

main()
{
 struct bankaccount myaccount={101,'C',"Binod Chapagain",100.0};
  ...
  /* pass the structure variable myaccount */
  ShowAccount(myaccount);
}

void ShowAccount(struct bankaccount acc)
{
 ...
}
```

Returning Structure Variables from Functions

- Entire structure variable may be returned from a function
- To do this, the return type of the function must be a structure type
- When a structure variable is returned from the function, a copy of the variable is returned which can be assigned to another structure variable in the calling function

```
struct time {
   int hrs, mins;
};

/* function prototypes */
struct time addtime(struct time t1, struct time t2);
void printtime(struct time t);

main()
{
     struct time day1 = {4, 30}, day2 = { 5, 45};
     struct time totaltime;

     totaltime = addtime(day1, day2);
     printtime(totaltime);
}

/* function to add to time values */
struct time addtime(struct time t1, struct time t2)
{
     struct time total;
     total.mins = t1.mins + t2.mins;
     total.hrs = t1.hrs + t2.hrs + total.mins/60;
     total.mins %= 60;
     return total; /* return a structure variable */
}
```

Pointer to a Structure Variable

- C allows pointers to structures just as it allows pointers to any other type of variable
- Like other pointers, structure pointers are declared by placing `*` in front of the structure variable name
- For example
  - `struct banckaccount *pAccount;`
  - declares `pAccount` as pointer to a structure variable of type `struct bankaccount`

Using Structure Pointers

- To initialize a structure pointer, use the `&` operator to get the address of a structure variable
  - `struct bankaccount acc, *pAcc;`
  - `pAcc = &acc;`
- Now `pAcc` is a pointer to a structure of type `bankaccount`, and `*pAcc` is pointed structure variable (`acc`)
- `(*pAcc).acct_no` is the `acct_no` member of the pointed structure variable, `acc`
  - The parentheses are necessary in `(*pAcc).acct_no` because the precedence of the structure member operator `.` is higher then `*`
  - The expression `*pAcc.acct_no` means `*(pAcc.acct_no)`, which is illegal here because `acct_no` is not a pointer

The arrow pointer

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand
- If p is a pointer to a structure, then
  - `p->`*`member-of-structure`*
  - refers to the particular member
- The **`->`**, usually called the arrow operator, is used to access a structure member through a pointer to structure
- Hence
  - `pAcc->acct_no` is same as `(*pAcc).acct_no`

Using Structure Pointer

```
struct bankaccount
{
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
};

main()
{
   struct bankaccount acc = {1001, 'C', "Binod Chapagain",
12000.0};
   struct bankaccount *pAcc;

   pAcc = &acc;

   printf("Account No: %d\n", pAcc->acct_no);
   printf("Account Type: '%c'\n", pAcc->acct_type);
   printf("Account Holder'name: %s\n", pAcc->name);
   printf("Current Balance: %f\n", pAcc->balance);
}
```

Use of Structure Pointers

- There are two primary uses for structure pointers:
  - To pass a structure to a function using a call by reference
  - To create linked lists and other dynamic data structures that rely on dynamic allocation
- There is one major drawback to passing structure variables to functions: the overhead needed to creating a copy and passing it to the function
  - For simple structures, this overhead is not too great
  - If the structure contains many members, run-time performance may degrade to unacceptable levels

Passing Structure Pointers

- The solution is to pass a pointer to the structure
- When a structure pointer is passed to a function, only the address of a structure variable is actually passed to the function. This makes very fast function calls
- A second advantage is that passing a pointer makes it possible for the function to modify the contents of the structure used as the argument

```c
struct bankaccount {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
};

void adjust(struct bankaccount *pAcc);
main()
{
   struct bankaccount acc = {1001, 'S', "Binod Chapagain", 12000.0};

   adjust(&acc); /* pass a pointer to the structure variable acc */

   printf("Account No: %d\n", acc.acct_no);
   printf("Account Type: '%c'\n", acc.acct_type);
   printf("Account Holder'name: %s\n", acc.name);
   printf("Current Balance: %f\n", acc.balance);
}

void adjust(struct bankaccount *pAcc)
{
       float interest = 0;
       if (pAcc->acct_type == 'S')
              interest = pAcc->balance*0.12;
       pAcc->balance += interest;
}
```

Dynamically Allocated Structures

- The most important use of structure pointer is with dynamically allocated structures
- You can dynamically allocate memory for a structure variable with the `malloc()` function and assign the returned address by `malloc()` to a structure pointer variable
- To calculate the memory required by a structure use the `sizeof` operator
  - ```
    struct bankaccount *pAccount;
    pAccount = malloc(sizeof(struct bankaccount));
    ```
    ```

```
void main()
{
  struct bankaccount *pAccount;

  /* allocate memory for a structu
  pAccount = (struct bankaccount

  printf("Enter account no. : ");
  scanf("%d", &pAccount->acct_no);
  printf("Enter account type (S - Savings, C - Current, F - Fixed) ");
  scanf(" %c", &pAccount->acct_type);
  printf("Enter account holder's name: ");
  scanf(" %[^\n]", pAccount->name);
  printf("Enter initial balance: ");
  scanf("%f", &pAccount->balance);

  printf("Account No.: %d\n", pAccount->acct_no);
  printf("Account Type: %c\n", pAccount->acct_type);
  printf("Account Holder's Name: %s\n", pAccount->name);
  printf("Current Balance: Rs. %.2f\n", pAccount->balance);

   /* release the memory allocated with malloc */
  free(pAccount);
}
```

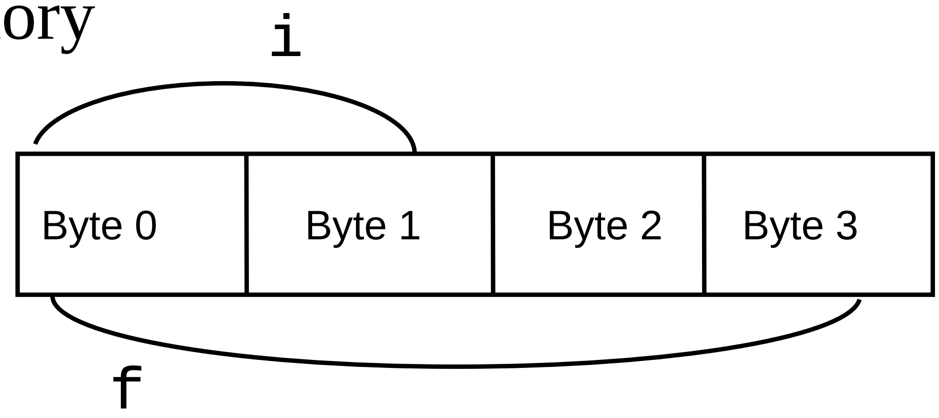
Note that `malloc()` returns pointer of type `void`. But the variable `pAccount` is a pointer of type `struct bankaccount`. So the type cast is done to convert `void *` to `struct bankaccount *`

Unions

- `union`
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a union share space
  - Conserves storage
  - Only the last data member defined can be accessed
  - The size required by a union variable is the size required by the member requiring the largest size

Union Declarations

- Same as `struct`
  - ```
    union number {
        int i;
        float f;
    };
    union number value;
    ```
- The variable `value` in memory


    ```

Operations on Union Variables

- *Skipped*
- Same as with structures
- Remember that only the value of the last defined member will be valid
  - ```
    value.i = 100;
    value.f = 102.5;
    value.i = 200;
    ```
  - Now only the `i` member of `value` will be valid
    ```

Self-Referential Structures

- A self-referential structure is a structure in which at least one of the member is a pointer to the parent structure type
- For example
  - ```
    struct node {
       char name[40];
       struct node *next;
    };
    ```
  - This structure contains two members: a 40-element character array, called `item` and a pointer to a structure of the same type (a pointer to a structure of type `node`), called `next`
- Useful in applications that involve dynamic data structures, such as lists and trees
    ```