Structures

… and other user-defined data types

Basic Definitions

What is a Structure?

It is a convenient construct for representing a group of logically related data items.

Particularly relevant when the constituent data items are of different types.

Compare with arrays that are collections of items of the same type.

Better readability even when the constituent data items are of the same type.

- Examples:
  - Student name (string), roll number (string), height (float), and marks (int).
  - Real part and imaginary part of a complex number (pair of double).

Structures help in organizing composite data in a programmer-friendly way.

The individual structure elements are called *members* or *fields*.

This is our first look at a user-defined data type.

Defining a Structure

The composition of a structure may be defined as:

```
struct  ⟨name of structure⟩  {
            ⟨data type⟩   ⟨member1 name⟩ ;
            ⟨data type⟩   ⟨member2 name⟩ ;
                  . . .
            ⟨data type⟩   ⟨memberk name⟩ ;
};
```

For example:

```
struct point {
    float xcoord;
    float ycoord;
};
```

Example

A structure definition:

```
struct student {
    char name[30];
    char roll_number[12];
    float height;
    int total_marks;
};
```

Defining structure variables:

```
struct student a1, a2, a3;
```

A new data type

- struct is the required C keyword
- Do not forget the ; at the end of the structure defn
- The individual members can be ordinary variables, pointers, arrays, or other structures (any data type)
- The member names within a particular structure must be distinct from one another
- A member name can be the same as the name of a variable defined outside the structure

Structure Definition versus Structure Variable Declaration

Structure Definition

```
struct point {
   float xcoord;
   float ycoord;
};
```

- No memory is allocated
- Only defining a new data type

Structure Variable Declaration

```
struct point a, b, c;
```

- Here a, b, c are variables of the type struct point
- Memory is allocated for a, b, c.
- Variable declaration is allowed *after* definition

Structure Variable Declaration can be clubbed with Definition

Separately

```
struct point {
     float xcoord;
     float ycoord;
};

struct point a, b, c;
```

Together

```
struct point {
     float xcoord;
     float ycoord;
} a, b, c;
```

- The struct definition can be reused elsewhere
- Like:

```
struct point p, q;
```

Another way

```
struct {
     float xcoord;
     float ycoord;
} a, b, c;
```

- In this case we do not have a name for the struct
- Hence we cannot reuse the struct definition

Each structure type variable (like a, b, c) has its own copy of each member of that structure type.

Type Definitions

The *typedef* construct

The *typedef* construct can be used to give new names to (existing) data types in C.

```
typedef float kilometers_per_hour;   // kilometers_per_hour is a new name for float
                                     // Note: no variable is allocated space here


kilometers_per_hour speed;           // Here speed is a declared variable
speed = 40;
```

More complicated examples of *typedef*

```
typedef int intarray[50];
intarray  A,        // A is an array of 50 integers
          B[20],    // B is an array of 20 arrays of 50 integers (a 20 x 50 array)
          *C;       // C is a pointer to an array of 50 integers

typedef int *intptr;
intptr    p,        // p is an int pointer
          q[10],    // q is an array of 10 int pointers
          *r;       // r is a pointer to an int pointer
```

These declarations are equivalent to:

```
int A[50], B[20][50], (*C)[50];
int *p, *q[10], **r;
```

Structures and *typedef*

Without *typedef*

```
struct complex
{
    float  real;
    float  imag;
};

struct complex a, b, c;
```

Here **struct complex** is a **new data type**.

With *typedef*

```
typedef struct {
    float  real;
    float  imag;
} complex;

complex a, b, c;
```

Here **complex** is a **new data type**. Since struct is not followed by a name, this data type can be addressed only by the given name **complex**.

Accessing members and using structures

Accessing the members of a structure

- The members of a structure are accessed individually, as separate entities.

- A structure member can be accessed by writing

  ⟨variable-name⟩.⟨member-name⟩

  where *variable* refers to the name of a structure-type variable, and *member* refers to the name of a member within the structure.

Dot operator is used to access members of a structure, through a structure-type variable

```
struct point  {
      float xcoord;
      float ycoord;
} a, b;
a.xcoord = 2.5;  a.ycoord = 3.2;
b.xcoord = b.ycoord = 0;
```

Structure initialization

Structure variables may be initialized following similar rules of an array. The values are provided within braces separated by commas.

An example:

```
struct complex a = {1.0,2.0}, b = {-3.0,4.0};
```

⬇

```
a.real = 1.0;
a.imag = 2.0;
b.real = -3.0;
b.imag = 4.0;
```

Example: Addition of two complex numbers

```
#include  <stdio.h>
int main( )
{
   struct complex {
        float real;
        float imag;
   } a, b, c;

   scanf ("%f %f", &a.real, &a.imag);
   scanf ("%f %f", &b.real, &b.imag);

   c.real = a.real + b.real;
   c.imag = a.imag + b.imag;
   printf ("a + b = %f + %f j\n", c.real, c.imag);
}
```

Structure declaration can be outside main() as well. This is necessary if a program has multiple functions using a structure type.

Assignment of Structure Variables

```
struct class {
    int  number;
    char name[20];
    float marks;
 };
```

```
int main()
{
   struct class student1 = {111, "Rao", 72.50};
   struct class student2 = {222, "Reddy", 67.00};
   struct class student3;

   student3 = student2;
}
```

A structure variable can be directly assigned to another variable of the same type.
All the individual members get assigned / copied.

But two structure variables CANNOT be compared for equality or inequality

```
if (student1 == student2) . . .
```

this cannot be done

An interesting observation

```
int a[5] = {10, 20, 30, 40, 50};
int b[5];

b = a;
```

X This is not allowed

```
struct list {
    int x[5];
};

struct list a, b;
a.x[0] = 10;
a.x[1] = 20;
a.x[2] = 30;
a.x[3] = 40;
a.x[4] = 50;

b = a;
```

This is allowed !!

Structures can be copied directly – even if they contain arrays !!

Assigning all the members of a structure together

```
struct student {
   char name[50];
   float CGPA;
   int height;
} s;

s = { "Foolan Barik", 8.79, 176 };                    NOT ALLOWED

s = (struct student){ "Foolan Barik", 8.79, 176 };    ALLOWED
```

Size of a structure

```
struct student {
   char name[50];
   float CGPA;
   int height;
} s;
```

Calculation shows a total space of 50 + 4 + 4 = 58 bytes to store all the members of struct student.

But sizeof(struct student) or sizeof(s) may be 60 (the nearest larger multiple of 4) or even 64 (the nearest larger multiple of 8).

```
struct student {
   char *name;
   float CGPA;
   int height;
} s;
```

Assume 64-bit addresses. Then sizeof(struct student) or sizeof(s) would be 8 + 4 + 4 = 16.

This will be true irrespective of how much memory you malloc to s.name.

Arrays of structures

Arrays of Structures

Once a structure data type has been defined, we can declare an array of structures.

```
struct class {
        int   number;
        char  name[20];
        float marks;
 };
 struct class student[50];
```

- The individual members can be accessed as:

`student[k].marks`	*marks of the kth student*
`student[k].name`	*name of the kth student*
`student[k].name[j]`	*jth character in the name of the kth student*

Example: Store a list of students (name, CGPA), compute average CGPA

```
#include <stdio.h>

struct student{
     float cgpa;
     char name[10];
};
int main(
{
    int i; float avg;
    struct student st[5];
    printf("Enter records of 5 students\n");
    for (i=0; i<5; i++) {
      printf ("Enter Cgpa:");
      scanf ("%f",&st[i].cgpa);
      printf ("Enter Name:");
      scanf ("%s”,st[i].name);
    }
```

```
    // compute average cgpa
    avg = 0.0;
    for (i=0;i<5;i++)
        avg += st[i].cgpa;

    avg = avg / 5.0;
    printf ("Avg cgpa:%f", avg);

    return 0;
}
```

Note: `&st[i].cgpa` is to be interpreted as `&(st[i].cgpa)`, not as `(&st[i]).cgpa` (an address cannot have a member).

A structure may contain other structures

```
typedef struct {
   double x, y;
} point;

typedef struct {
   point A, B, C;
   double area;
} triangle;

triangle T = { {1.0, 2.0}, {-4.0, 5.0}, {3.0, -6.0}, -1 };

T.area = T.A.x * (T.B.y - T.C.y) +
         T.B.x * (T.C.y - T.A.y) +
         T.C.x * (T.A.y - T.B.y);
T.area /= 2;
if (T.area < 0) T.area = -T.area;
printf("Area of T = %lf\n", T.area);
```

Output

```
Area of T = 17.000000
```

Structure containment cannot be recursive

It is not allowed that `struct a` contains `struct a` members.

It is not allowed that `struct a` contains `struct b`, and `struct b` contains `struct a`.

It is allowed that a structure contains a pointer to a structure of the same type.

These are called self-referencing pointers.

Such pointers are used extensively to create chains and other types of linked data structures.

```
typedef struct _student {          // A name after struct is mandatory
   char name[50];
   float CGPA;
   struct _student *next;          // A self-referencing pointer
} student;
```

Structures and functions

Structures are passed by value to functions

```
#include <stdio.h>

typedef struct {
      float real;
      float imag;
} COMPLEX;

void swap ( COMPLEX a, COMPLEX b )
{
    COMPLEX tmp;

    tmp = a;   a = b;   b = tmp;
}
```

```
void print ( COMPLEX a )
{
   printf ("(%f, %f) ",
                    a.real, a.imag);
}

main( )
{
   COMPLEX x = {4.0, 5.0},
           y = {10.0, 15.0};

   print(x); print(y); printf("\n");
   swap(x, y);
   print(x); print(y); printf("\n");
}
```

Program output

```
(4.000000, 5.000000) (10.000000, 15.000000)
(4.000000, 5.000000) (10.000000, 15.000000)
```

No swapping takes place actually, similar to what we saw for integers, floats, and so on.

Structures can be returned from functions

```
#include <stdio.h>

typedef struct {
        float real;
        float imag;
} COMPLEX;


COMPLEX add ( COMPLEX a, COMPLEX b )
{
    COMPLEX tmp;
    tmp.real = a.real + b.real;
    tmp.imag = a.imag + b.imag;
    return tmp;
}
```

```
main( )
{
     COMPLEX x = {4.0, 5.0},
             y = {10.0, 15.0};
     COMPLEX z;

     z = add(x, y);
     printf (" %f, %f \n",
                 z.real, z.imag);
}
```

Program output

14.000000, 20.000000

Pointers to Structures

Pointers and Structures

```
struct class {
    int  roll;
    char name[20];
    float marks;
};

struct class *ptr;
```

Once `ptr` points to a structure variable, the members can be accessed as:

```
ptr -> roll;
ptr -> name;
ptr -> marks;
```

- The symbol "–>" is called the *arrow* operator.

Arrow operator used to access members of a structure, through a structure-type pointer.

`ptr -> member` is a shortcut for `(*ptr).member`.

Use of pointers to structures

```
#include <stdio.h>
struct complex  {
     float  real;
     float  imag;
};

main( )
{
   struct complex  a, b, c;
   scanf ("%f %f", &a.real, &a.imag );
   scanf ("%f %f", &b.real, &b.imag );
   add( &a, &b, &c ) ;
   printf ("%f %f\n", c.real, c.imag );
}
```

```
void add (struct complex *x,
  struct complex *y, struct complex *t)
{
     t->real = x->real + y->real;
     t->imag = x->imag + y->imag;
}
```

A Warning

When using structure pointers, we should take care of operator precedence.

- Member operator "." has higher precedence than the dereferencing operator "*"

  `ptr -> roll` and `(*ptr).roll` mean the same thing.

  `*ptr.roll` will lead to error.

- The operator "–>" enjoys the highest priority among operators.

  `++ptr -> roll` will increment roll, not ptr.

  `(++ptr) -> roll` will increment the pointer (to the next structure in an array) and access roll in the next structure.

Practice problems

1. Extend the complex number program to include functions for addition, subtraction, multiplication, and division
2. Define a structure for representing a point in two-dimensional Cartesian coordinate system. Using this structure for a point, do the following.
   a. Write a function to return the distance between two given points
   b. Write a function to return the middle point of the line segment joining two given points
   c. Write a function to compute the area of a triangle formed by three given points
   d. Write a main function and call the functions from there after reading in appropriate inputs (the points) from the keyboard

3. Define a structure STUDENT to store the following data for a student: name (null-terminated string of length at most 20 chars), roll no. (integer), CGPA (float). Then

   a. In main, declare an array of 100 STUDENT structures. Read an integer n and then read in the details of n students in this array
   b. Write a function to search the array for a student by name. Returns the structure for the student if found. If not found, return a special structure with the name field set to empty string (just a '\0')
   c. Write a function to search the array for a student by roll no.
   d. Write a function to print the details of all students with CGPA > x for a given x
   e. Call the functions from the main after reading in name/roll no/CGPA to search

Unions

Unions

- In a struct, space is allocated as the sum of the space required by its members.
- In a union, space is allocated as the union of the space required by its members.
  - We use union when we want only one of the members, but don't know which one.

Suppose that we wish to store the height of a student in one of the following formats.

- In the FPS system, it is a string like 5'10" (let us use a character array of size 8).
- In the CGS system, it is an integer like 178 (4 bytes are needed).
- In the MKS system, it is a floating-point number like 1.78 (4 bytes are needed).
- If we use a structure with all these members, we need 8 + 4 + 4 = 16 bytes of space.
- A union will use only max(8, 4, 4) = 8 bytes of space.
- You need to store an additional flag to tell which representation it is.

Union example

```c
typedef struct {
  char name[50];
  float CGPA;
  char htype;
  union {
     char fps[8];
     int cgs;
     float mks;
  } height;
} student;
```

```c
int main ()
{
  student S;
  printf("size of the union is %lu\n", sizeof(S.height));
  printf("Enter type of height (f/c/m): ");
  scanf("%c", &S.htype);
  printf("Enter height: ");
  if (S.htype == 'f') scanf("%s", S.height.fps);
  else if (S.htype == 'c') scanf("%d", &S.height.cgs);
  else if (S.htype == 'm') scanf("%f", &S.height.mks);
  switch (S.htype) {
     case 'f' : printf("%s\n", S.height.fps); break;
     case 'c' : printf("%d\n", S.height.cgs); break;
     case 'm' : printf("%.2f\n", S.height.mks); break;
     default: printf("Unknown height type\n");
  }
}
```

Output

```
size of the union is 8
Enter type of height (f/c/m): f
Enter height: 6'5''
6'5''
```