Structures

5th March 2012

What is a Structure?

- It is a convenient tool for handling a group of logically related data items.
  - Student name, roll number, and marks
  - Real part and complex part of a complex number

– Combine heterogeneous data to form a named collection

- Essential for building up “interesting” data structures — e.g., data structures of multiple values of different kinds

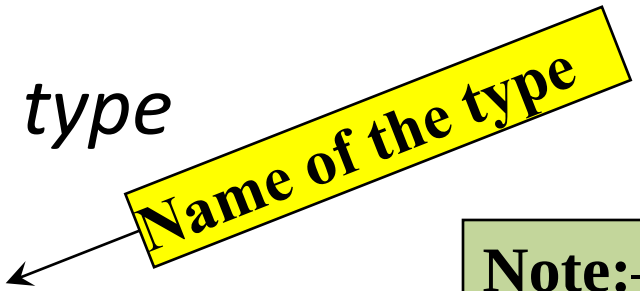
Definition — *Structure*

- A collection of one or more variables, typically of different types, grouped together under a single name for convenient handling

- Known as **`struct`** in *C*

`struct`

- Defines a new *type*
- E.g.,



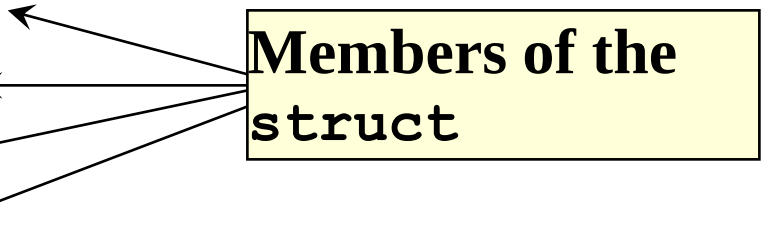

```
struct motor {
  float volts;
  float amps;
  int phases;
  float rpm;
};      //struct motor
```

Note:– name of type is optional if you are just declaring a single `struct`

`struct`

- Defines a new *type*
- E.g.,

```
struct motor {
  float volts;
  float amps;
  int phases;
  float rpm;
};      //struct motor
```



A *member* of a `struct` is analogous to a *field* of a class in Java

Declaring `struct` variables

```
struct motor p, q, r;
```

- Declares and sets aside storage for three variables – **`p`**, **`q`**, and **`r`** – each of type **`struct motor`**

```
struct motor M[25];
```

- Declares a 25-element array of **`struct motor`**; allocates 25 units of storage, each one big enough to hold the data of one **`motor`**

```
struct motor *m;
```

- Declares a pointer to an object of type **`struct motor`**

Example

- A structure definition:

```
struct student {
    char  name[30];
    int   roll_number;
    int  total_marks;
    char  dob[10];
};
```

- Defining structure variables:

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

A new data-type

Structures

- Compound data:

- A date is
  - an **`int month`** and
  - an **`int day`** and
  - an `int year`

```
struct ADate {
  int  month;
  int  day;
  int  year;
};

struct ADate date;

date.month = 9;
date.day = 1;
date.year = 2005;
```

Structure Representation & Size

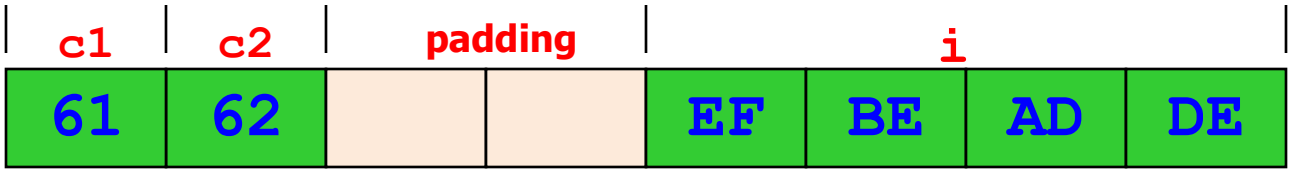
`sizeof(struct …)` =

sum of **`sizeof`**(field)

\+ alignment padding

Processor- and compiler-specific

```
struct CharCharInt {
  char  c1;
  char  c2;
  int   i;
} foo;
foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xDEADBEEF;
```



x86 uses "little-endian" representation

Members

```
struct  name {
    char first[10];
    char midinit;
    char last[20];
} sname, ename;
```

- To access the members of a structure, we use the member access operator “**.**”.

```
strcpy (sname.first, “Aritra”);
sname.midinit = ‘K’;
strcpy (sname.last, “Saha”) ;
```

typedef

a typedef is a way of *renaming* a type

```
typedef struct {
    char first[10];
    char midinit;
    char last[20];
} NAMETYPE;

NAMETYPE sname,ename;
```

```
struct name {
    char first[10];
    char midinit;
    char last[20];
} ;

typedef struct name nameType;

nameType name1, name2;
```

Operations on `struct`

- Copy/assign

```
struct motor p, q;
p = q;
```

- Get address

```
struct motor p;
struct motor *s
s = &p;
```

- Access members

```
p.volts;
(*s).amps;        s->amps;
```

Things you can and can't do

- You can

  Use = to assign whole struct variables
- You can

  Have a struct as a function return type
- You cannot

  Use == to directly compare struct variables; can compare fields directly
- You cannot

  Directly scanf or printf structs; can read fields one by one.

Operations on `struct` (function call)

- Passing an argument by value is an instance of *copying* or *assignment*
- Passing a return value from a function to the caller is an instance of *copying* or *assignment*

```
struct motor f(struct motor g) {
  struct motor h = g;
  ...;
  return h;
}
```

Struct initializers

/* typedef structs go on top */

StudentRecord s1 = {"V Singhal", "00CS1002", 167, 8.31};

Using components of struct variables

```
s1.height = 169;
s1.cgpa = 8.4;
scanf ("%s", s1.rollno) ;
```

Example: Complex number addition

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

int main( ) {
        complex a, b, c;
        scanf (“%f %f”, &a.real, &a.imaginary);
        scanf (“%f %f”, &b.real, &b.imaginary);

        c.real = a.real + b.real;
        c.imaginary = imaginary + b.imaginary;
        printf (“\n %f + %f j”, c.real, c.imaginary);
}
```

Example: Complex number

```
#include  <stdio.h>

typedef struct  {
        float  real;
        float  imaginary;
} complex;

complex read_complex ( ) {
        complex c;
        scanf ("%f %f", &c.real, &c.imaginary);
        return c;
}

void print_complex (complex c) {
        printf (" %f + i %f  ", c.real, c.imaginary);
}
```

Complex number arithmetic

```
complex add_complex (complex c1, complex c2) {
        complex csum;
        csum.real = c1.real + c2.real;
        csum.imaginary = c1.imaginary + c2.imaginary;
        return csum;
}

complex sub_complex (complex c1, complex c2) {
        complex cdiff;
        cdiff.real = c1.real + c2.real;
        cdiff.imaginary = c1.imaginary + c2.imaginary;
        return cdiff;
}
```

Complex number arithmetic

```
complex mult_complex (complex c1, complex c2) {
        complex cprod;
        cprod.real =
        cprod.imaginary =
        return cprod;
}
int main (  ) {
        complex c1, c2, c3, c4;
        read_complex (c1) ; read_complex (c2) ;
        c3 = add_complex (c1, c2) ;
        printf (“Sum of”) ;
        print_complex (c1) ;
        printf (“and”);
        print_complex (c2);
        printf(“is”) ;
        print_complex (c3);
 }
```

Unions

- A **`union`** is like a **`struct`**, but only one of its members is stored, not all
  - I.e., a single variable may hold different types at different times
  - Storage is enough to hold largest member
  - Members are overlaid on top of each other

```
union {
  int ival;
  float fval;
  char *sval;
} u;
```

Unions (continued)

- It is *programmer's responsibility* to keep track of which type is stored in a **`union`** at any given time!

```
struct taggedItem {
  enum {iType, fType, cType} tag;
  union {
   int ival;
   float fval;
   char *sval;
  } u;
};
```

Unions (continued)

- It is *programmer's responsibility* to keep track of which type is stored in a `union` at any given time!

```
struct taggedItem {
    enum {iType, fType, cType} tag;
    union {
      int ival;
      float fval;
      char *sval;
    } u;
};
```

Members of `struct` are:–

```
enum tag;
union u;
```

Value of `tag` says which member of `u` to use

Unions (continued)

- **`unions`** are used much less frequently than **`structs`** — mostly
  - in the inner details of operating system
  - in device drivers
  - in embedded systems where you have to access registers defined by the hardware

Arrays of Structures

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

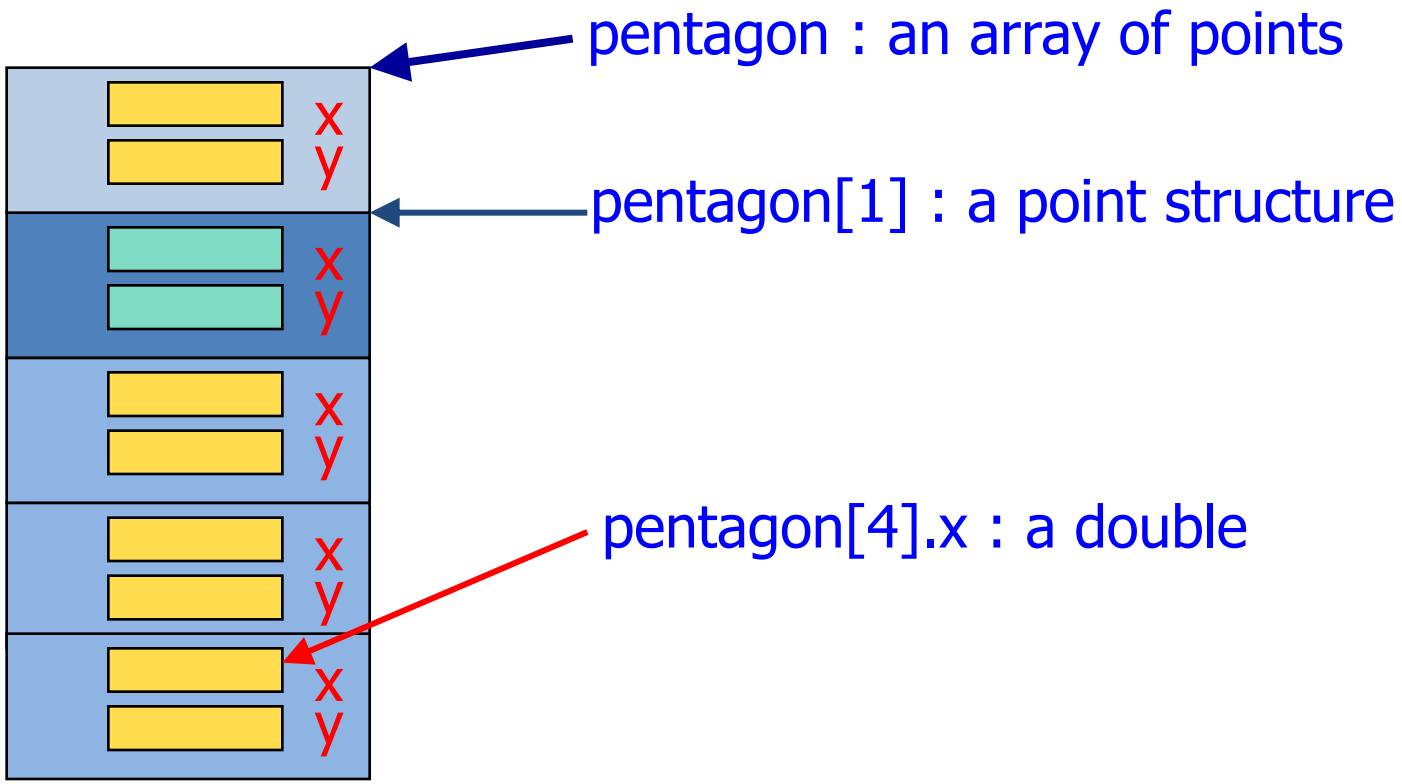
```
struct student class[50];
```

  - The individual members can be accessed as:

```
class[i].name
class[5].roll_number
```

struct Arrays

```
typedef struct {
        double x;
        double y;
} point;
point pentagon[5];
```



Using Arrays of structs

```
StudentRecord   class[MAXS];
...
for (i=0; i<nstudents; i++)   {
       scanf (“%d%d”, &class[i].midterm, &class[i].final);
      class[i].grade = (double)(class[i].midterm + class[i].final)/50.0;
}
```

Passing Arrays of structs

- An array of structs is an array.
- When any array is an argument (actual parameter), its address is passed, not copied [as for any array]

```
int avg (StudentRec class[MAX]) {
      … …
}
int main ( ) {
        StudentRec bt01[MAX];
        int average;
        …
        average = avg_midpt(bt01) ;
}
```

A function using struct array

```
int fail (StudentRecord  slist [ ]) {
    int i, cnt=0;
    for (i=0; i<CLASS_SIZE; i++)
        cnt += slist[i].grade == 'F';
    return cnt;
}
```

Exercise Problems

1. Define a structure for representing a point in two-dimensional Cartesian co-ordinate system.
   - Write a function to compute the distance between two given points.
   - Write a function to compute the middle point of the line segment joining two given points.
   - Write a function to compute the area of a triangle, given the co-ordinates of its three vertices.