Structures & Unions

Structures

- ***W***e've seen variables of simple data types, such as float, char, and int.
- Variables of such types represent one item of information: a height, an amount, a count, and so on.
- But just as groceries are organized into bags, employees into departments, and words into sentences, it's often convenient to organize simple variables into more complex entities.
- The C++ construction called the *structure* is one way to do this.

Structures

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling .
- The variables in a structure can be int, some can be float, and so on. This is unlike the array, in which all the variables must be the same type. The data items in a structure are called the **members** of the structure.

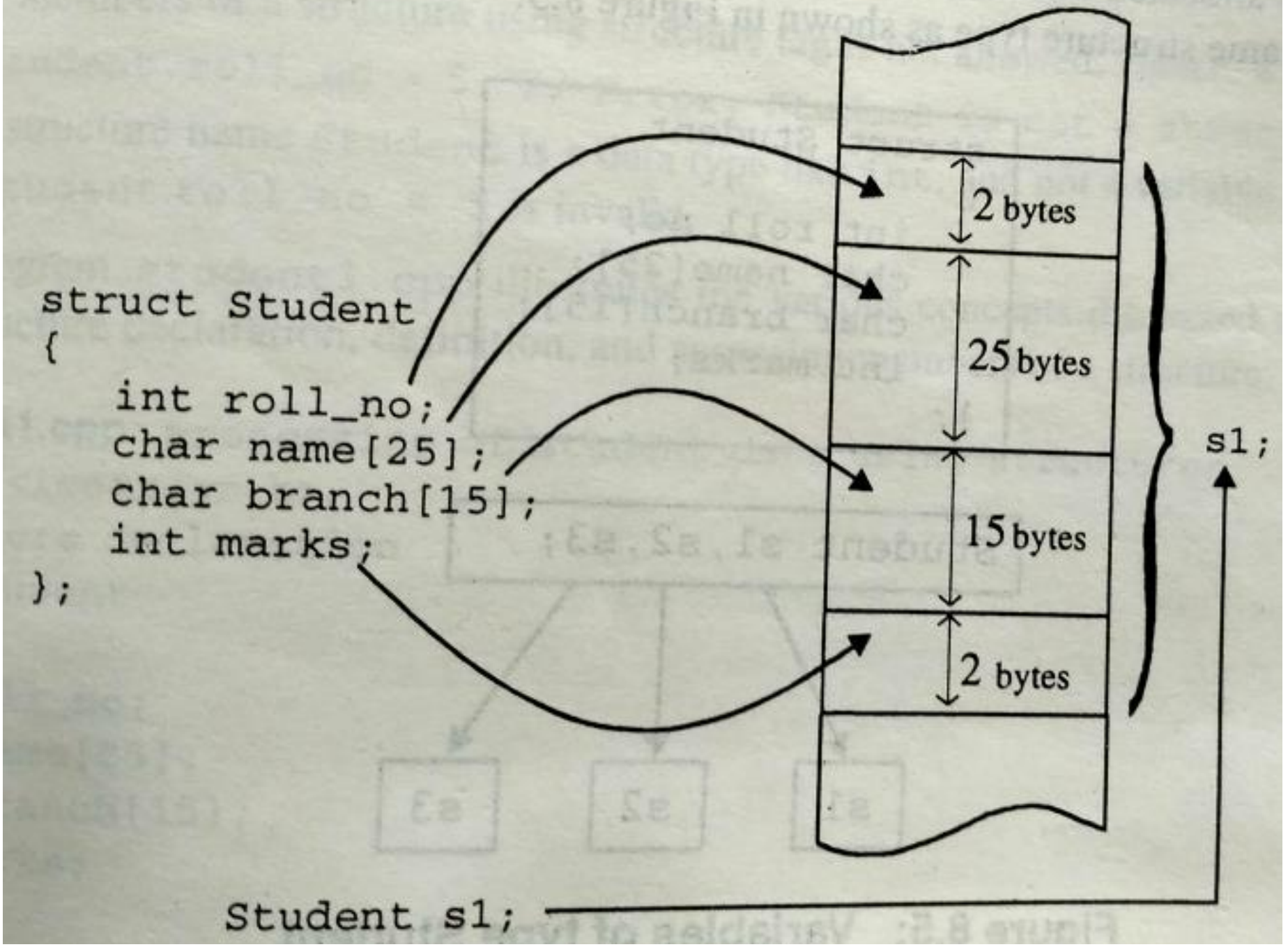
- **Structures assist program organisation by**
  - Grouping logically related data, and giving this set of variables a higher-level name and more abstract representation.
  - Enabling related variables to be manipulated as a single unit rather than as separate entities.
  - Reducing the number of parameters that need to be passed between functions.
  - Providing another means to return multiple values from a function.

- The general format of a structure definition is

```
struct  tag_name
{
Data_type  member1;
Data_type  member2;
---------------
};
```

Example:-1

```
struct  student
        {       int  rollno;
                int  age;
                char  name[10];
                float  height;
        };
```


struct Student
{
int roll_no;
char name[25];
char branch[15];
int marks;
};
2 bytes
25 bytes
15 bytes
2 bytes
s1;
Student s1;

Rules for defining structure

- The structure template is terminated with a semicolon.
- While the entire declaration is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
- The tag name such as student can be used to declare structure variables of its type, later in the program.

Declaring Structure Variables

Example-2

1. Declare them at the structure definition.

```
struct student
 {     int rollno;
       int age;
       char name[10];
       float height;
 }s1, s2, s3;
```

2. Define the variables at some point *after* the structure definition.

```
student s1, s2, s3;
```

Note:

- Members of a structure themselves are not variables.
- They (members) do not occupy any memory until they are associated with the structure variables such as s1, s2, s3 etc.
- Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined.
- They may also appear before main. In such cases, the definition is global and can be used by other functions as well.

- The **sizeof** operator is used to find the size of structure.

sizeof(struct x)→ gives the number of bytes required to hold all the members of the structure x.

- If y is a simple structure variable of type struct x, then the expression **sizeof(y)** →would also give the same answer.

Giving values to members.

The link between member and a structure variable is established using the member **operator '.'** which is also known as **'dot operator' .**

For example,

s1. rollno is the variable representing the rollno of student s1 and can be treated like any other ordinary variable.

Here is how we would assign values to the members of student s1

strcpy(s1.name, “Anil”);

s1.rollno=1335;

s1.age=18;

s1.height=5.8;

We can also use cin to give the values through the keyboard.

cin>>s1.name>>s1.age>>s1.rollno>>s1.height;

Example-3

```
4   struct  personal
5   {
6       char name[20];
7       int day;
8       char month[10];
9       int year;
10      float salary;
11  };
```

```
12  int main()
13  {
14      personal person;
15      cout<<"Input values\n";
16      cin>>person.name>>person.day
17         >>person.month>>person.year
18         >>person.salary;
19
20      cout<< person.name<<endl
21          <<person.day<<endl
22          <<person.month<<endl
23          <<person.year<<endl
24          <<person.salary<<endl;
25  }
```

Structure initialization

Example-4

```
struct   part
 {   int   modelnumber;
     int    partnumber;
     float  cost;
};
int  main()
{
part   part1  =  {  6244,  373,  217.55  };
}
```

- ✓ The values to be assigned to the structure members are surrounded by braces and separated by commas.
- ✓ The first value in the list is assigned to the first member, the second to the second member and so on.
- ✓ The order of values in the initializer list matches the order of declarations in the structure.

Comparison of structure variables

- *If person1 and person2 belong to the same structure, then the following operation is valid.*

```
person1 = person2;   // Assign person2 to person1
But    if(person1==person2 )  // illegal
         if(person1!=person2)    //illegal
```

Example-5

```
4   struct  person
5   {
6       char name[20];
7       int age;
8       float salary;
9   };
10  int main()
11  {
12      person person1 = {"abc", 28, 36000};
13      person person2 = {"xyz", 25, 30000 }, p3;
14
15      p3 = person2;
16
17      cout<<p3.name<<"\n"<<p3.age<<"\n"<<p3.salary<<"\n";
18
19      p3 = person1;
20
21      cout<<p3.name<<"\n"<<p3.age<<"\n"<<p3.salary;
22      return 0;
23  }
```

Arrays of structures

For example,

class1 student[100]; → *defines an array called student, that consists of 100 elements.*

Example-6

```
3  struct marks
4  {
5      int subject1, subject2, subject3;
6  };
7  int main()
8  {
9      marks students[] = { {45, 67, 78}, {75, 55, 69}, {55, 79, 90} };
10     // ..................
11 }
```

This declares the student as an array of 3 elements and initializes their members as follows.

student[0].subject1=45;
student[0].subject2=67;
student[0].subject3=78;
……
……
student[2].subject3=90;

```
marks students[] = { 45, 67, 78, 75, 55, 69, 55, 79, 90 };
```

Arrays within structures

We can use single or multidimensional arrays inside a structure.

```
struct  marks
{
int   number;
float   sub[3];
} student[2];
```

The member sub contains 3 elements sub[0], sub[1],sub[2]. These elements can be accessed using appropriate subscripts.

```
4  struct marks
5  {
6      int sub[3];
7      int total;
8  };
9
10 int main()
11 {
12     marks student[3] = { {45,67,78,0}, {75,55,69,0}, {55,79,90,0} };
13     marks subtotal{0,0,0,0};
14     // ...
15 }
```

student[1].sub[2]→ *refers to the mark in third subject by the second student.*

STRUCTURES WITHIN STRUCTURES

- *Structure within structure means nesting of structures.*

- *Consider the following structure defined to store information about the salary of employees*

```
struct  salary
{
   char   name[20];
   char   dept[10];
   int   basic_pay;
   int   dearness_allowance;
   int   h_allowance;
   int   city_allowance;
}employee;
```

STRUCTURES WITHIN STRUCTURES

```
3   struct date
4   {
5       int day, month, year;
6   };
7
8   struct student
9   {
10      unsigned int RegNo;
11      char name[20];
12      date birth_date;
13  };
14
15  int main()
16  {
17      student s = {111, "Anil", 20, 5, 2000 };
18      cout<<s.RegNo<<"\n"<<s.name<<"\n"
19      <<s.birth_date.day<<"-"<<s.birth_date.month
20      <<"-"<<s.birth_date.year;
21      // ..................
22      return 0;
23  }
```

```
4  struct student
5  {
6      unsigned int RegNo;
7      char name[20];
8      struct date
9      {
10         int day, month, year;
11     }birth_date;
12 };
13
14 int main()
15 {
16     student s = {111, "Anil", 20, 5, 2000 };
17     cout<<s.RegNo<<"\n"<<s.name<<"\n"
18     <<s.birth_date.day<<"-"<<s.birth_date.month
19     <<"-"<<s.birth_date.year;
20     // ..................
21     return 0;
22 }
```

- *It is also permissible to nest more than one type of structures.*

```
struct personal_record
{
   struct   name_part   name;
   struct   addr_part   address;
   struct   date  date_of_birth;
.....
....
};
struct personal_record person1;
```

Unions

- ✓ *Unions look similar to structures.*
- ✓ *They have identical declaration syntax and member access.*
- ✓ *The major difference between them in terms of storage.*
- ✓ *In structures, each member has its own storage location.*
- ✓ *Whereas all the members of a union use the same location.*
- ✓ *This implies that, although a union may contain many numbers of different types. It can handle only one member at a time.*

```cpp
union item
{
    int i;
    double  d;
    char c;
};
```

```cpp
int main()
{
    item U;
    cout<<sizeof(U);

    U.i= 11;
    cout<<"\nU.i: "<<U.i<<"\n";

    U.d= 12.5;
    cout<<"U.i: "<<U.i<<"\n"; //Invalid
    cout<<"U.c: "<<U.c<<"\n"; //Invalid
    cout<<"U.d: "<<U.d<<"\n";

    U.c = 'a';
    cout<<"U.i: "<<U.i<<"\n"; //Invalid
    cout<<"U.c: "<<U.c<<"\n";
    cout<<"U.d: "<<U.d<<"\n"; //Invalid
    return 0;
}
```

```
8
U.i: 11
U.i: 0
U.c:
U.d: 12.5
U.i: 97
U.c: a
U.d: 12.5
```

- *A union holds the value of one-variable at a time.*
- *The compiler allocates a piece of storage that is large enough to hold the biggest member of the union.*
- *In the declaration above, the member d requires 8 bytes which is the largest among the members*

User defined Type declarations

- ***typedef***
  - *Type definition - lets you define your own identifiers.*
- ***enum***
  - *Enumerated data type - a type with restricted set of values.*

User defined Type Declaration

- ***typedef*** **:** general declaration format

  typedef **type identifier;**

  The "type" refers to an existing data type and "identifier" refers to the new name given to the data type.

  After the declaration as follows:

  typedef **int marks;**

  typedef **float units;**

  we can use these to declare variables as shown

  marks *m1,m2[10];* *//m1 & m2[10]* are declared as integer variables

  units *u1, u2;* *//u1 & u2* are declared as floating point variables

  The main advantage of typedef is that we can create meaningful data type names for increasing the readability of the program.

User defined Type Declaration

- ***enum*** **data type :** general declaration format

  enum identifier {value1, value2,..,value$_n$};

  The “identifier” is a user defined enumerated data type which can be used to declare variables that can have one of the values known as *enumeration constants.*

  After this declaration as follows:

  enum identifier v1,v2,…v$_n$;

  enumerated variables v1,v2,..,v$_n$ can only have one of the values value1, value2,…,value$_n$

User defined Type Declaration

E.g.:

```
enum  day {Monday, Tuesday,… ,Sunday} ;
enum  day week_st, week_end;
        week_st = Monday; week_end= Friday;
        if(week_st == Tuesday)
                week_end = Saturday;
```

Compiler automatically assigns integer starting with 0 to all enumeration constants. But can be overridden.

E.g.:

```
enum  day { Monday=1, Tuesday,…, Sunday};
```

Monday is assigned 1 & subsequent constants incremented by one.

enum – example

If declared; *enum* day { Monday=1, Tuesday,…, Sunday};

```
cout<<"\n\nEnter  n >1 &<7., 0 for Exit:- ";
  cin>>i;
  switch(i)
  {
            case Monday:
                        cout<<“Monday.";
                        break;
            case Tuesday
                        cout<<“ Tuesday.";
                        break;
            case Wednesday:
                        cout<<“ Wednesday.";
                        break;
            case Thursday:
                        cout<<“ Thursday.";
                        break;
            case Friday:
                        cout<<“ Friday.";
                        break;
            case Saturday:
                        cout<<“\Saturday.";
                        break;
            case Sunday:
                        cout<<“Sunday.";
                        break;
            case 0:
                 exit(0);
            default:
            cout<<"\nInvalid Entry. Enter  0-7.";
                        break;
}
```