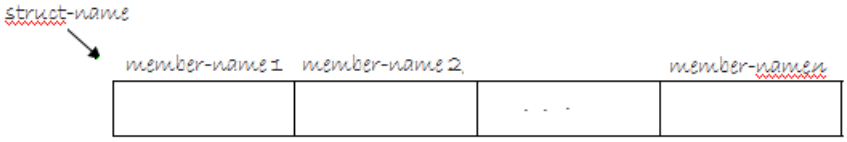
Struct .....

With array, we can only declare one data type. It is single type aggregate data type. Struct overcomes this problem by declaring composite data types which can consist different types.

A struct is a collection of related data items stored in one place and can be referenced by more than one names. These data items are different basic data types(simple data type(int, double, char.,.), array, struct). So, the number of bytes required to store them may also vary. It is a user-defined composite type and in order to use a struct, we must first declare a struct template. The variables in a struct are called elements or members.

SYNTAX FOR DEFINING STRUCT IS:

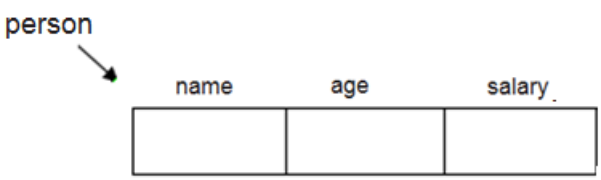
```
struct struct_name {
        datatype member_name 1;
        datatype member_name 2;
        datatype member_name3 ;
              .
              .
              .
        datatype member_namen ;
  } [struct_variables];
```



- The struct_name becomes a user-define type and it used the same way as other built-in data types.

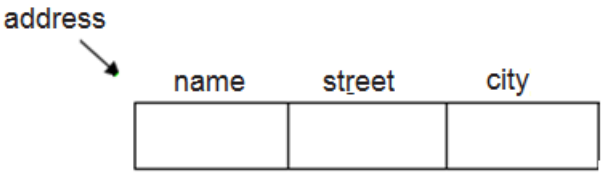
Example 1:In the following define **person** structure that consist of three members(name, age, salary):

```
struct address  {
      char name[30];
      int age;
      float salary;
   };
```



Example 2: In the following define **adress** structure that consist of three members(name, street, city):

```
struct address {
        char name[30];
        char street[40];
        char city[20];
  };
```



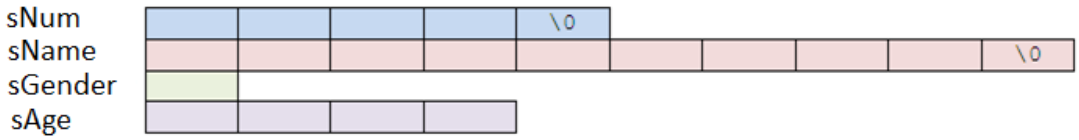
Example 3: In the following define **student** structure that consist of six members as following:

```
struct student{
     char id[6];
     char name[30];
     char gender;              // student gender M (for Male) or F (for Female)
     int  age;
     double avg[4];           // student average for 4 years
     int  mark[4][5];         // student marks in 5 lesson for 4 years
  };
```

Example3: To store a student's record with the elements sNum (identification number), shName, shGender and sAge, we can declare the following struct:

```
struct student {
    char sNum[5];
    char sName[10];
    char sGender;
    int sAge;
    };
```

The structure can be illustrated as follow: (note the different data



Example4: Declure MyEmployee struct has three members: eName, eNum, and eDepatment. The eName member is a 20-element array, and eNum and eDepatment are simple members with int types, respectively.

```
struct MyEmployee {  // defines a structure variable named EmpData
   char eName[20];
   int eNum;
   int eDepatment;
} EmpData;
```

Example5: Declare MyCube struct has two members : width and length:

```
struct MyCube{
      int width, length;
   } area;
```

Notes:

- You have the option of declaring variables when the struct type is defined by placing one or more comma-separated variable names between the closing brace and the semicolon.
- Struct variables can be initialized. The initialization for each variable must be enclosed in braces.
- Both struct types and variables follow the same scope as normal variables, as do *all identifiers*. If you define a struct within a function, then you can only use it within that function. Likewise if you define a struct outside of any function then it can be used in any place throughout your program.

Accessing Structure Member

Individual members of a structure are accessed through the use of the **. (dot)** operator.

Syntax:

structure_name . member_name

The information contained in one structure may be assigned to another structure of the same type using a single assignment statement. That is, you do not need to assign the value of each member separately.

☒ **Program to assign structure to another structure.**

```
#include <iostream.h>
void main() {
   struct {
       int a;
       int b;
    } x, y;
```

```
   x.a = 10;
   y = x;  // assign one structure to another
   cout << y.a;
}
```

- **Program to assign data to members of a structure variable and display it.**

```
#include <iostream.h>
#include <stdio.h>
 struct  person  {
       char name[50];
       int age;
       float salary;
    };

void main() {
   person p1;
   cout << "Enter Full name: ";
   cin.get(p1.name, 50);
   cout << "Enter age: ";
   cin >> p1.age;
   cout << "Enter salary: ";
   cin >> p1.salary;
   cout << "\nDisplaying Information." << endl;
   cout << "Name: " << p1.name << endl;
   cout <<"Age: " << p1.age << endl;
   cout << "Salary: " << p1.salary;
 }
```

Initializing Structures

You can initialize a structure at the time that it is declared as the following example:

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

declare variable d_date a and initialize structure as follows:

```
Date d_date = {12, 31, 2004};
```

Structures and Functions

☒ Pass a member of a structure to a function, consider this structure:

```
struct test{
   char x;
   int y;
   float z;
   char s[10];
} t_test ;

func(t_test.x);          // passes character value of x
func2(t_test.s[2]);      // passes character value of s[2]
```

☒ Use a structure as a parameter (call by value), the type of the argument must match the type of the parameter.

```
#include <iostream.h>
struct test {
     int a, b;
     char ch;
  } ;

void f1(struct test p) {
    cout<< p.a;
  }

void main(){
     struct test  arg;
     arg.a = 1000;
     f1(arg);
   }
```

☒ Use a structure as a parameter (call by reference)

```
#include <iostream>
 struct  person  {
    char name[50];
    int age;
    float salary;
   } p1;

void readinfo( person & p1 )  {
    cout << "Enter Full name: ";
    cin.get(p1.name, 50);
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;
  }

void printinfo ( person &p1 )
  {
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout <<"Age: " << p1.age << endl;
```

```
    cout << "Salary: " << p1.salary;}

void main() {
    readinfo(p1);
    printinfo(p1);}
```

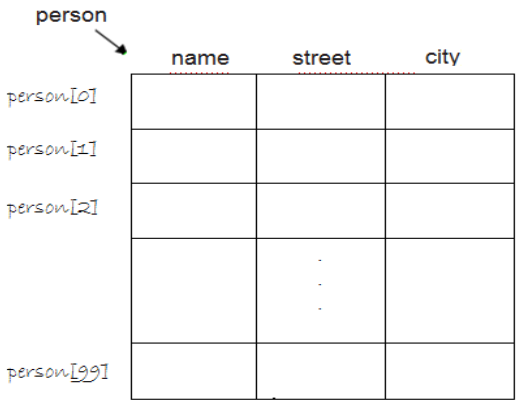
Arrays of Structures

The most common usage of structures is in arrays of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, in the following declare a 100 element array of structures of type **address**:

```
        struct address addr_info[100]; // addr_info is array of structure address
```

and:

```
 cout <<  addr_inf[2].name //  to print  the name in element 2.
```



for 100 student records, we can declare a structure like the following:

```
struct student{
        int sNum, sAge;
        char sName[80];
        char sGender;
}studRecord[100];
```

- Or something like the following statements,

```
struct student{
        int sNum, sAge;
        char sName[80];
        char sGender;
};
```

- And later we can declare something like this,

```
struct student studRecord[100];
```

☒ Program to read and print data for n person.

```
#include <iostream.h>
#include <stdio.h>
 struct  person {
    char name[50];
    int age;
    float salary;
   }p1;

void readinfo( person p[], int n )
  { for ( int i=0 ; i<n ; i++ )
      { cout << "\nEnter Information for person:" << i+1 << endl;
        cout << "Enter Full name: ";
        gets(p[i].name);
        cout << "Enter age: ";
        cin >> p[i].age;
        cout << "Enter salary: ";
        cin >> p[i].salary;
      }
  }

void printinfo ( person p[], int n )
  { cout << "\nDisplaying Information for persons:" << endl;
    cout << "Name\t\t\t";
    cout << "Age\t\t\t";
    cout << "Salary";
    cout << endl;
    for ( int i=0 ; i<n ; i++ )
      {
        cout << p[i].name ;
        cout << "\t\t" << p[i].age ;
        cout << "\t\t" << p[i].salary;
        cout << endl;
      }
  }
void main() {
    const int size = 10;
    person p[size];
    int n;
    cout << " Enter number of persons: " ;
    cin >> n;
    readinfo ( p, n );
    printinfo( p, n );
 }
```

When a structure is a member of another structure, it is called a **nested structure**, for example : consider this structure:

```
struct x {
    int a[10][10];
    float b;
  } y;
```

And to access the element (3,7) in **a** of structure **y**, write: y.a[3][7]
And consider this structure:

```
struct emp {
        struct address addr;
        float wage;
    } worker;
```

To access member city , write: cout << worker.addr.city

Unions

A *union* is a memory location that is shared by two or more different types of variables. Declaring a **union** is similar to declaring a structure.

SYNTAX FOR DEFINING UNION IS:

```
union union_name {
        datatype member_name 1;
        datatype member_name 2;
              ...
        type member_namen ;
    } [union_variables];
```

Example:

```
union u_type {
  int i;
  char ch;
 }uVar;
```

In **cVar**, both integer **i (2 byte)** and character **ch (1 byte)** share the same memory location. When a **union** variable is declared, the compiler automatically allocates enough storage to hold the largest member of the **union**. For example (assuming 2-byte integers and 1 byte for char).

```
struct StudentRecord  {
    int student_number;
    char grade;
  };
int main( ){
   StudentRecord   your_record;
   your_record.student_number = 2001;
   your_record.grade = 'A';}
```