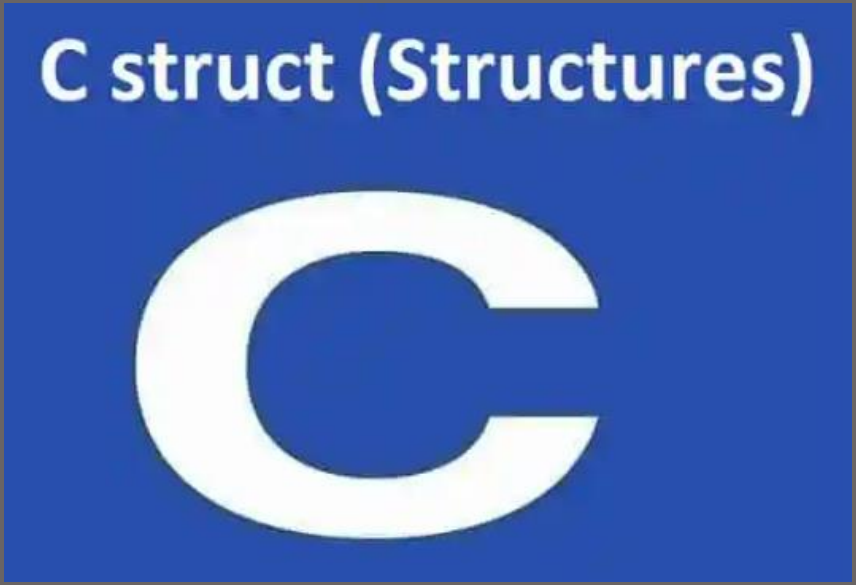
Structures in C

Lecture 11 – ICT1132



Piyumi Wijerathna
Department of ICT
Faculty of Technology

Overview

- Basic Concepts of Structure
- Process a Structure
- Structure with typedef keyword
- Structures as Function Arguments
- Unions in C Language
- Static Variables

Introduction

Some information about a person:

- Name,
- NIC,
- Salary…..etc.

- You can easily create different variables for name, NIC and salary to store these information separately.

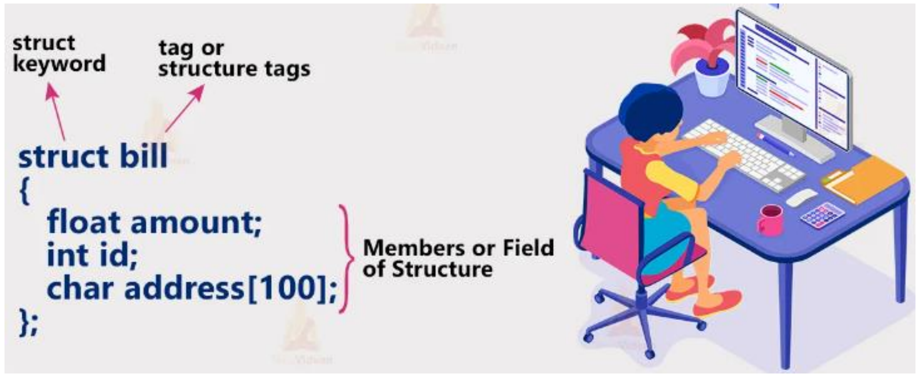
- However, in the future, you would want to store information about multiple persons.

- You can have a collection of all related information under a single name Person.

Introduction

Structure is a collection of variables of different types under a single name.

- The variables that make up the structure are called ***members***, ***elements* or *fields***.
- Generally, all the members of a structure are logically related.
  - **Ex:** name, NIC, address, age, height etc.



```
struct bill
{
   float amount;
   int id;
   char address[100];
};
```

Structures

- Structures hold data that belong **together**.
- Examples:
  - Student record: student id, name, major, gender, start year, …
  - Bank account: account number, name, currency, balance, …
  - Address book: name, address, telephone number, …
- In database applications, structures are called records.

Structures

- Individual components of a struct type are called members (or fields).
- Members can be of different types (simple, array or struct).
- A struct is named as a whole while individual members are named using field identifiers.
- Complex data structures can be formed by defining arrays of structs.

struct basics

- **Definition of a structure:**

```
struct <struct-type>{
     <type> <identifier_list>;
     <type> <identifier_list>;
     ...
} ;
```

Each identifier defines a member of the structure.

- **Example:**

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

The “Date” structure has 3 members, day, month & year.

struct examples

- Example:

```
struct StudentInfo{
        int Id;
        int age;
        char Gender;
        double GPA;
};
```

The “StudentInfo” structure has 4 members of different types.

- Example:

```
struct StudentGrade{
        char Name[15];
        char Course[9];
        int Lab[5];
        int Homework[3];
        int Exam[2];
};
```

The “StudentGrade” structure has 5 members of different array types.

struct examples

- Example:

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

The “Date” structure has 3 members.

- Example:

```
struct BankAccount{
      char Name[15];
      int AcountNo[10];
      double balance;
      Date Birthday;
};
```

The “BankAcount” structure has simple, array and structure types as members.

struct examples

- Example:

```
struct studentRecord{
    char Name[15];
    int  Id;
    char Dept[5];
    char gender;
};
```

The “StudentRecord” structure has 4 members.

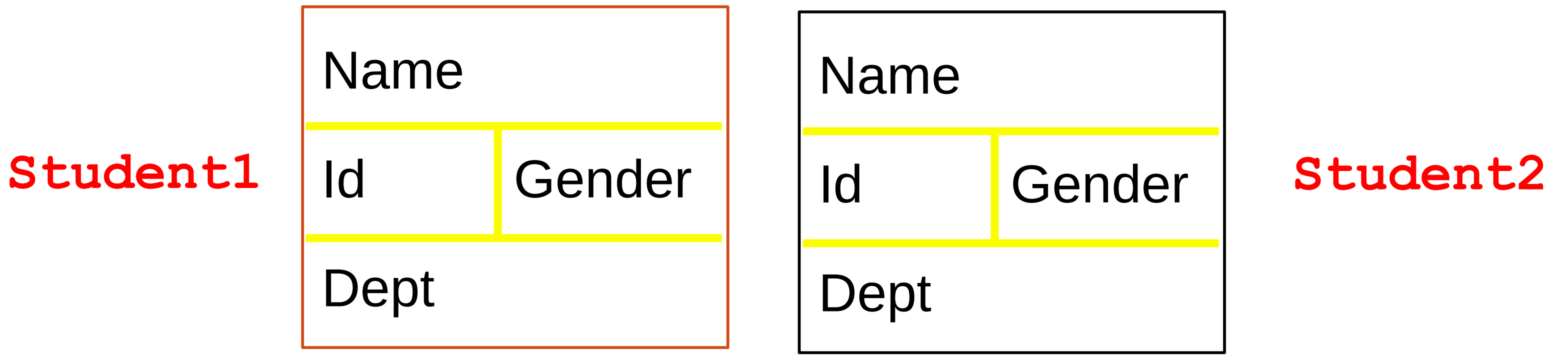
struct basics

- Declaration of a variable of struct type:

```
struct <struct-type> <identifier_list>;
```

- Example:

```
struct StudentRecord Student1, Student2;
```



Student1 and **Student2** are variables of **StudentRecord** type.

Ex.1: struct basics

- The members of a **struct** type variable are accessed with the dot (.) operator:

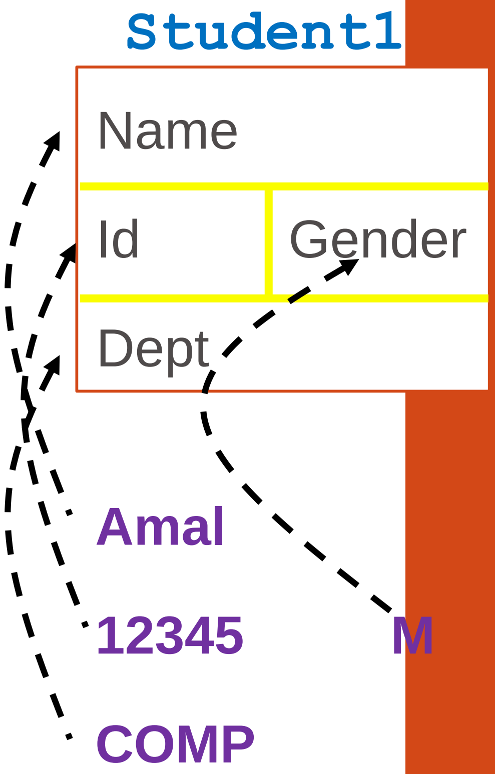
```
<struct-variable>.<member_name>;
```

- Example:

```
strcpy(Student1.Name, “Amal");
Student1.Id = 12345;
strcpy(Student1.Dept, "COMP");
Student1.gender = 'M';
```

```
printf(“%s”,Student1.Name);
```

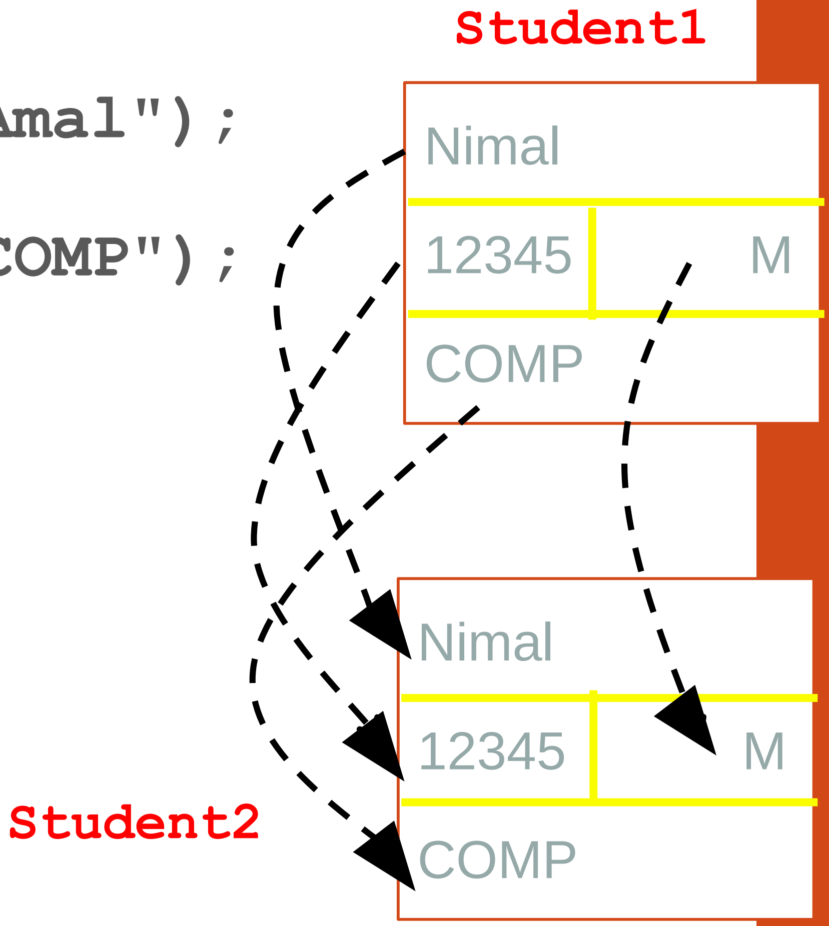
```
printf("The student is “);
switch (Student1.gender){
      case 'F': printf("Ms. “); break;
      case 'M': printf ("Mr. “); break;
}
```



Ex.2: struct-to-struct assignment

- The values contained in one struct type variable can be assigned to another variable of the same struct type.
- Example:

```
strcpy(Student1.Name,“Amal");
Student1.Id = 12345;
strcpy(Student1.Dept,"COMP");
Student1.gender = 'M';

Student2 = Student1;
```



```
#include <stdio.h>
#include <string.h>
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

      /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   return 0;}
```

Initializing structures

```
struct employee emp1 = {“ Tom”, “Hanks”, 39, ‘M’, 50000.00};
```

firstName	Tom
lastName	Hanks
age	39
gender	M
salary	50000.00

Accessing members of a structure

- // Input salary for emp1

```
scanf("%f", &emp1.salary);
```

- //print the salary of emp1

```
printf("%.2f", emp1.salary);
```

- //assign the gender for emp1

```
emp1.gender = 'M';
```

structure with *typedef* keyword

- To make the code clear, you can use ***typedef*** keyword to create a synonym for a structure.
- ***typedef*** allows to define types with a different name (to name the structure as a whole).

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

int main() {
comp comp1, comp2;
 }
```

two structure variables *comp1* and *comp2* are created by this *comp* type.

typedef

Ex:-

```
typedef struct{
                    int house_number;
                    char street_name[50];
                    int zip_code;
                    char country[50];
          } address;
 int main(){
          address billing_addr;
          address shipping_addr;
}
```

typedef Example

```
#include<stdio.h>
typedef struct employee
{
     char  name[50];
     float   salary;
} emp;

 void main( ){
     emp e1;
     printf("\nEnter Employee record\n");
     printf("\nEmployee name\t");
     scanf("%s",e1.name);
     printf("\nEnter Employee salary \t");
     scanf("%f",&e1.salary);
     printf("\nEmployee name is %s",e1.name);
     printf("\nSalary is %f",e1.salary);
 }
```

Structures as Function Arguments

We can pass a structure as a function argument in the same way as we pass any other variable, an array or a pointer.

Structure can be passed to functions by two methods:

- **Passing by value (passing actual value as argument)**
- **Passing by reference (passing address of an argument)**

Passed by value

```
#include <stdio.h>
struct student
{
    char name[50];
    int id;
};
void display(struct student stu);  ←——  // function prototype should be below
                                        to the structure declaration otherwise
                                        compiler shows error

int main(){
    struct student stud;
    printf("Enter student's name: ");
    scanf("%s", stud.name);
    printf("Enter id number:");
    scanf("%d", &stud.id);
    display(stud);    // passing structure variable stud as argument
    return 0;
}
void display(struct student stu){
    printf("Output\nName: %s",stu.name);
    printf("\nID: %d",stu.id);
}
```

Arrays of Structures

- We can declare an array of structures.
  - Ex:

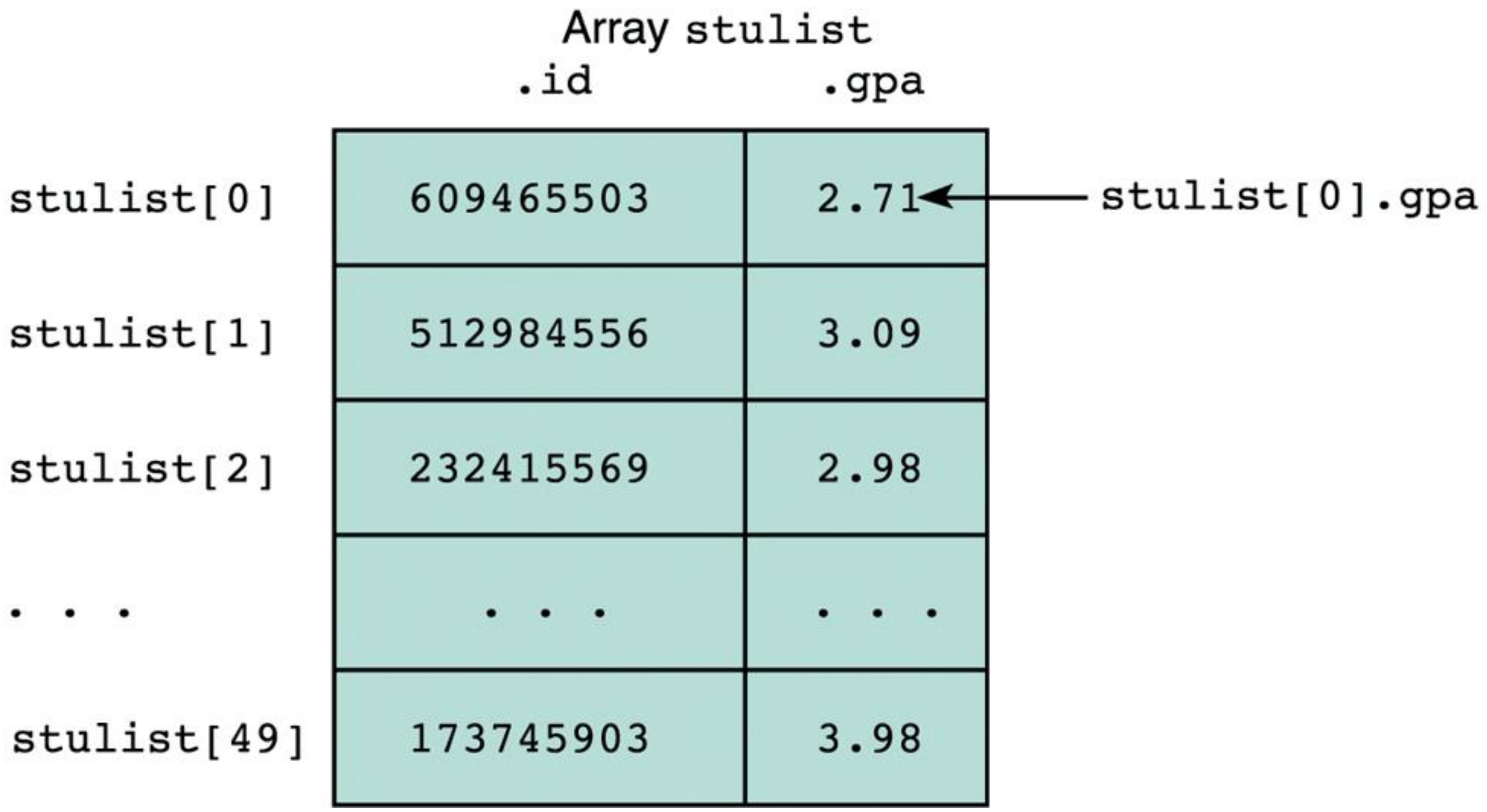
```
typedef struct{
    int id;
    double gpa;
} student_t;
```

  - Usage:

```
student_t stulist[50];

stulist[0].id = 9223;

stulist[0].gpa = 3.0;
```

Arrays of Structures

- The array of structures can be simply manipulated as arrays of simple data types.



	.id	.gpa
stulist[0]	609465503	2.71
stulist[1]	512984556	3.09
stulist[2]	232415569	2.98
. . .	. . .	. . .
stulist[49]	173745903	3.98

Structures with File Handling

Writing data from a structure to a file

```
# include <stdio.h>
struct customer{
        int accno;
        char name[30];
        double balance;
};
void main( ){
        struct  customer c1= {123, "Amal", 200.00};
        FILE *cfPtr;
        cfPtr = fopen("customer.dat", "w");
        if ( cfPtr == NULL)
                printf("File cannot be open");
        else{
                fprintf(cfPtr, "%d %s %.2f", c1.accno, c1.name, c1.balance);
                fclose(cfPtr);
        }
}
```

Writing data from a structure to a file (Data taken from Keyboard)

```
//define the structure here
void main( ){
        struct  customer c1;
        FILE *cfPtr;
        cfPtr = fopen("customer.dat", "w");
        if (cfPtr == NULL){
                printf("File cannot be open");
                return -1;
        }
        printf("Input the account number");
         scanf("%d", &c1.accno);
         printf("Input the name");
         scanf("%s", c1.name);
         printf("Input the account balance");
         scanf("%lf",&c1.balance);
        fprintf(cfPtr, "%d %s %.2f", c1.accno, c1.name, c1.balance);
        fclose(cfPtr); }
```

Writing multiple records from a structure to a file

```
struct  customer cus[5];
int i;
FILE *cfPtr;
cfPtr = fopen("customer.dat", "w");
if (cfPtr == NULL){
        printf("File cannot be open");
        return -1;
}
for(i = 0; i <5; ++i){
        printf("Input the account number");
        scanf("%d", &cus[i].accno);
        printf("Input the name");
        scanf("%s", cus[i].name);
        printf("Input the account balance");
        scanf("%lf",& cus[i].balance);
        fprintf(cfPtr, "%d %s %.2f\n", cus[i].accno, cus[i].name, cus[i].balance);
}
fclose(cfPtr);
```

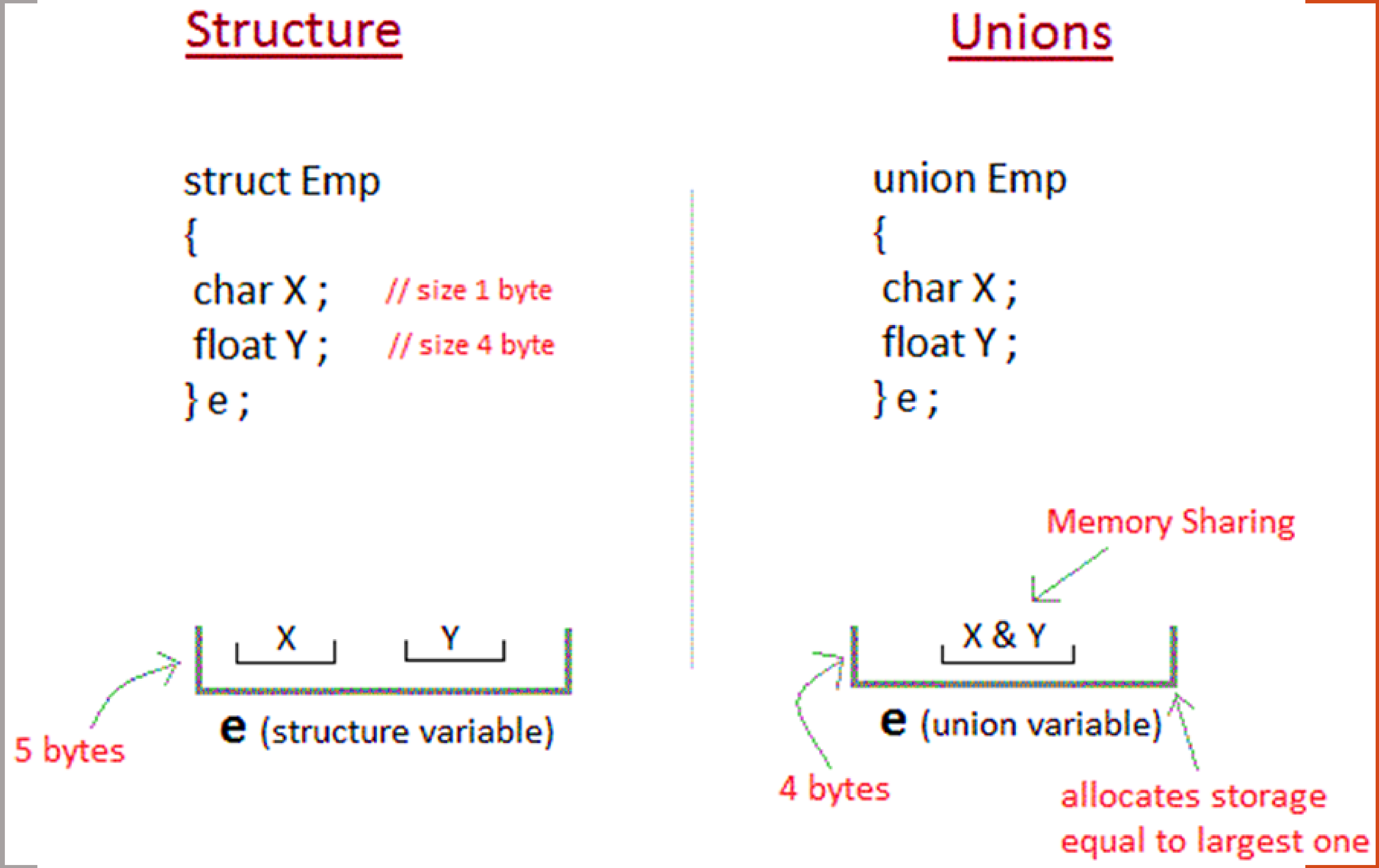
Reading data from a file to a structure

```
# include <stdio.h>
struct customer{
        int accno;
        char name[ 30];
        double balance;
};
void main( ){
        struct  customer c1;
        FILE *cfPtr;
        cfPtr = fopen("customer.dat", “r");
        if ( cfPtr == NULL)
                printf("File cannot be open");
        else{
                fscanf(cfPtr, "%d %s %lf", &c1.accno, c1.name,
&c1.balance);
                printf(“ %d %s %f”, c1.accno,c1.name, c1.balance);
        }
        fclose(cfPtr);
}
```

Unions in C Language

- **Unions** are conceptually similar to **structures**.
- The syntax of union is also similar to that of structure.
- The only differences is in terms of storage.
- In **structure** each member has its own storage location.
- But **union** uses a single shared memory location which is equal to the size of its largest data member.
- *Hence the Members of a union can only be accessed one at a time.*

Unions in C Language



Structure

```
struct Emp
{
 char X ;      // size 1 byte
 float Y ;     // size 4 byte
}e ;
```

Unions

```
union Emp
{
 char X ;
 float Y ;
}e ;
```

Defining a Union

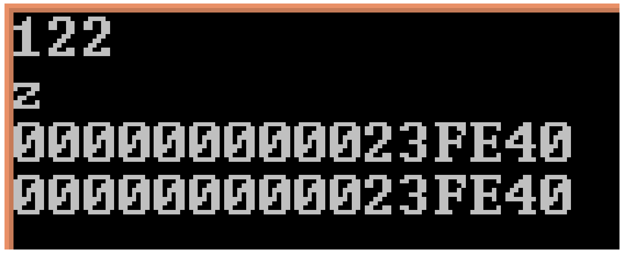
```
union [union tag] {

    member definition;

    member definition;

    ...

   member definition;

};
```

```
union Data {
   int i;
   float f;
   char str[20];
} ;
```

Example

```
#include <stdio.h>

union item{
        int a;
        char ch;
};

int main( ){
        union item it;
        it.a = 12;
        it.ch='z';

        printf("%d\n",it.a);
        printf("%c\n",it.ch);

        printf("%p\n",&it.a);
        printf("%p\n",&it.ch);

         return 0;
        }
```

Output



```
122
z
000000000023FE40
000000000023FE40
```

Example Cont.

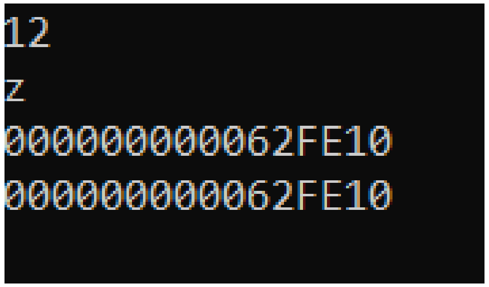
- In example we can see that the value of a member of union got corrupted because the final value assigned to the variable ch has occupied the memory location.

- This is the reason that the value of ch member is getting printed very well and value of a member is not printed as we desired.

- To correct this issue, we have to use one variable at a time which is the main purpose of having unions.

```c
#include <stdio.h>

union item{
	int a;
	char ch;
};

int main( ){
	union item it;

	it.a = 12;
	printf("%d\n",it.a);

	it.ch='z';
	printf("%c\n",it.ch);

	printf("%p\n",&it.a);
	printf("%p\n",&it.ch);

	 return 0;
	}
```

Output



```
12
z
000000000062FE10
000000000062FE10
```

Here, all the members are getting printed very well because one member is being used at a time.

Global and Local variables

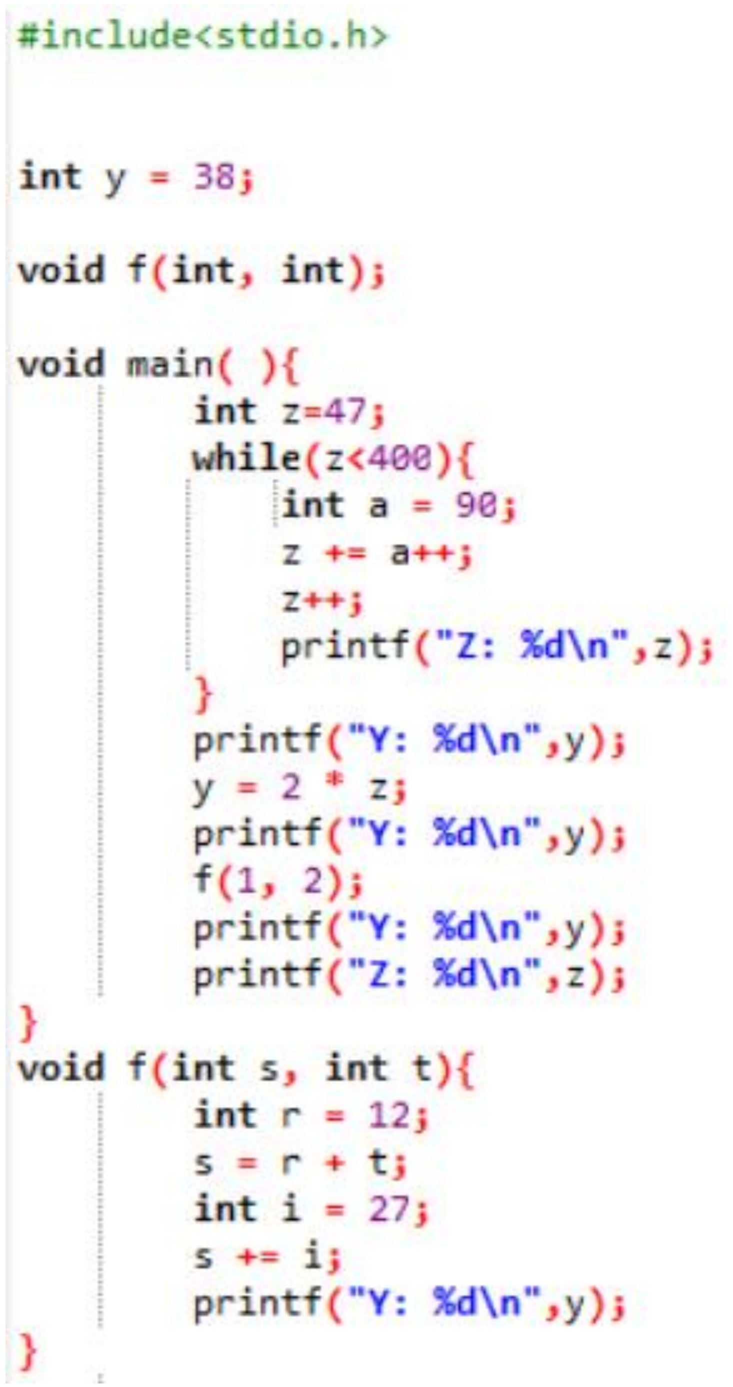
Global and Local variables

- A **global** declaration is made outside the bodies of all functions and outside the main program. It is normally grouped with the other global declarations and placed at the beginning of the program file.
- A **local** declaration is one that is made inside the body of a function. Locally declared variables cannot be accessed outside of the function they were declared in.
- It is possible to declare the same identifier name in different parts of the program.

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;
  printf ("value of a = %d, b = %d and g = %d\n", a,
b, g);

  return 0;
}
```

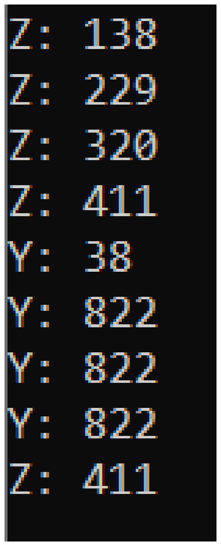
```
int y = 38;
void f(int, int);

void main( ){
        int z=47;
         while(z<400){
                int a = 90;

                z += a++;
                z++;
          }
         y = 2 * z;
         f(1, 2);
}
void f(int s, int t){
        int r = 12;
         s = r + t;
         int i = 27;
         s += i;
         printf("Y:%d\n",y);
}
```

What will happen if you declare another y here?

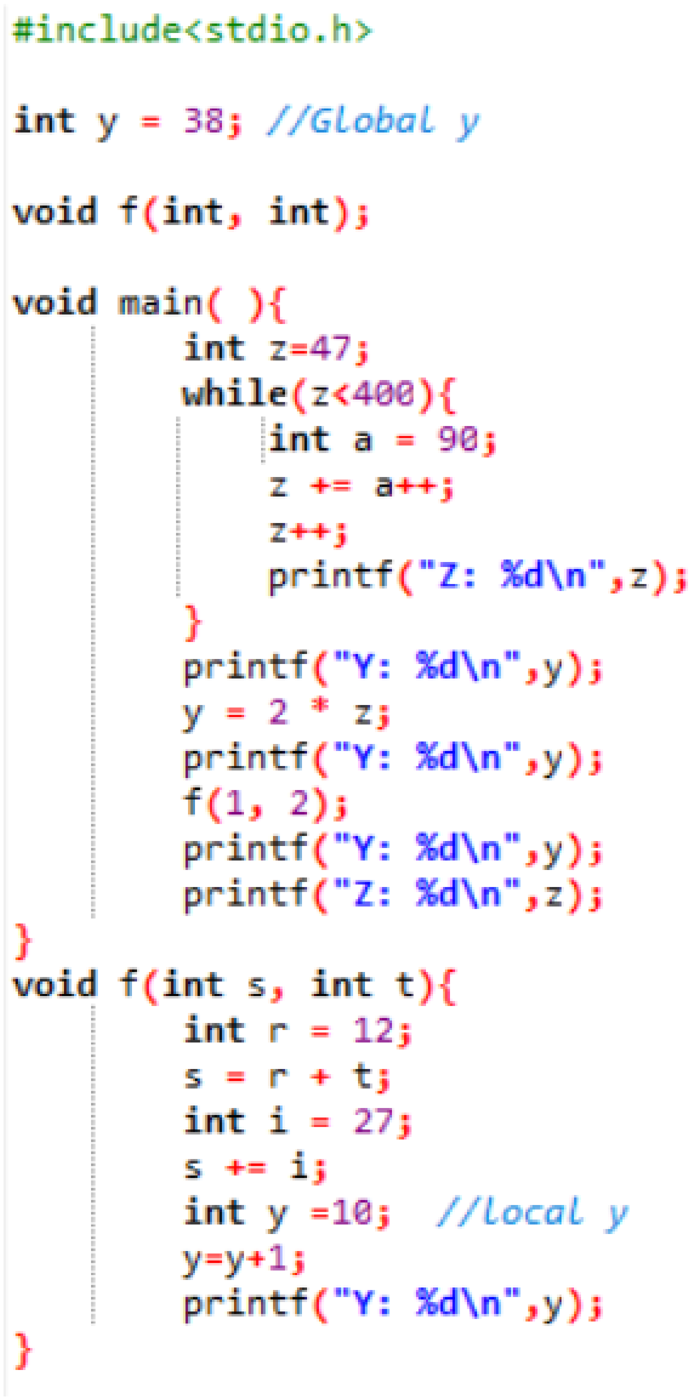


```c
#include<stdio.h>


int y = 38;

void f(int, int);

void main( ){
        int z=47;
        while(z<400){
            int a = 90;
            z += a++;
            z++;
            printf("Z: %d\n",z);
        }
        printf("Y: %d\n",y);
        y = 2 * z;
        printf("Y: %d\n",y);
        f(1, 2);
        printf("Y: %d\n",y);
        printf("Z: %d\n",z);
}
void f(int s, int t){
        int r = 12;
        s = r + t;
        int i = 27;
        s += i;
        printf("Y: %d\n",y);
}
```

Output

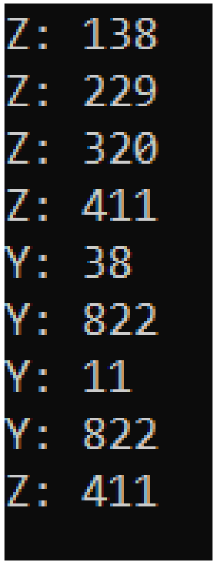


```
Z: 138
Z: 229
Z: 320
Z: 411
Y: 38
Y: 822
Y: 822
Y: 822
Z: 411
```



```c
#include<stdio.h>

int y = 38; //Global y

void f(int, int);

void main( ){
        int z=47;
        while(z<400){
            int a = 90;
            z += a++;
            z++;
            printf("Z: %d\n",z);
        }
        printf("Y: %d\n",y);
        y = 2 * z;
        printf("Y: %d\n",y);
        f(1, 2);
        printf("Y: %d\n",y);
        printf("Z: %d\n",z);
}
void f(int s, int t){
        int r = 12;
        s = r + t;
        int i = 27;
        s += i;
        int y =10;  //local y
        y=y+1;
        printf("Y: %d\n",y);
}
```

Output



```
Z: 138
Z: 229
Z: 320
Z: 411
Y: 38
Y: 822
Y: 11
Y: 822
Z: 411
```

Static variables

Static variables

What is a Static Variable?

- Static variables in C have the same value during block of code or during entire program.

Which means…….

- A static variable remains in memory while the program is running.
- Default value of the static variable is zero.
- It can be initialized only once.

<u>Declaration:</u>

```
static int x;

static float y;
```

What is the difference between Static and Global?

Static Variable	Global Variable
• Static variables are not accessible by other files.	• Global variables can be accessed by other files.
• It can be declared outside of all functions or within a function using static keyword before the data type of variable . • If static variables are declared outside of all functions it will have global scope or it is declared within a function it will have scope within a function.	• Global variables are declared outside of all functions.
• It will retain until the life of program. • Static variables are initialized only once at the time of declaration only.	• Global variable’s life is until the life of program.

Static variable

```
 #include<stdio.h>
int fun()
{
  static int count = 1;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output is 2 3

```
#include<stdio.h>
int fun()
{
  int count = 1;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output is 2 2

Any
Question

THANK YOU… !