TATA ELXSI
engineering creativity

TATA

C Programming

Presenter : Learning & Development Team

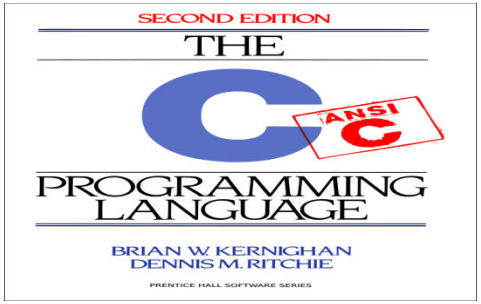
Day – 5

Disclaimer

- This material is developed for in-house training at TATA ELXSI.
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage. TATA ELXSI, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books. Excerpts of these material has been taken where there is no copy right infringements. Some examples and concepts have been sourced from the below links and are open source material

  http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

ANSI C K&R



Structures

Objectives

- Structure – Definition
- Initialization of Structure Elements
- Assignment of Structure Elements
- Typedef, function pointers and structures
- Structures as Function Arguments
- Function and structure
- Array and structure
- Pointers and structures
- Structure padding and packing
- Defining a Union
- Accessing Members of a Union
- Bit fields and their usage in optimizing
- enumerations

Structure declaration

- Structure is a collection of one or more variables of same type or different type.

- The keyword struct introduces a structure declaration, which is a list of declarations enclosed in braces.

- Syntax

```
struct  identifier   {
        data_type      data_member1;
        data_type      data_member2;
              .
              .
        data_type      data_member N;
};
```

Structure defination

- Example:

```
struct Employee  {
      char name[50];
      int     EmpID;
      float salary;
};
```

This declaration above creates the derived data type **struct Employee.**

To create object :

struct Employee   oRaj, oAdarsh; // oRaj, oAdarsh are two instance of struct Employee.

Structure definition and initialization

- Structure declaration do not occupy any memory.
- Memory will be allocated only while creating the instance of the structure.

- A structure can be initialized by following its definition with a list of initializers.
- struct Employee oEmpRaj = {“Raj kumar”,15399 ,100000.99 };
- Each a constant expression.

Structure member access

- Individual structure members can be initialized , once after creating the object .

  - struct Employee oEmpRaj;
  - strcpy(oEmpRaj.name,"Raj kumar");
  - oEmpRaj.EmpID = 15399;
  - oEmpRaj.salary = 1000000.00;

- A member of a particular structure is referred to in an expression by a construction of the form
- *structure-Object.member*
- The structure member operator "." connects the structure instance(object)and the member.

- Objects also can be created while declaring the structure.

```
struct Employee  {
      char name[50];
      int     EmpID;
      float salary;
} oRaj,    oAdarsh;
```

- Structure members can be initialized while creating the object.

```
struct Employee  {
      char name[50];
      int     EmpID;
      float salary;
}  oEmpRaj = {“Raj kumar”,15399 ,100000.99 };
```

Example 1:

```
struct Employee  {
      char name[50];
      int     EmpId;
      float salary;
}  oEmpRaj = {"Raj kumar",15399 ,100000.90 };
main ()
{
            struct Employee  oRecord;
            printf("Enter Employee  name\n");
            scanf("%s",oRecord.name);
            printf("Enter Emp ID\n");
            scanf("%d",&oRecord.EmpId );
            printf("Enter Emp salary\n");
            scanf("%f",&oRecord.salary );
```

Example 1:

```
printf("Employee One Data is \n");
printf(" Name is %s \n", oEmpRaj.name);
printf(" Employee ID is %d \n",oEmpRaj.EmpId);
printf(" Employee ID is %f \n",oEmpRaj.salary);

printf("Employee Two's Data is \n");
printf(" Name is %s \n", oRecord.name);
printf(" Employee ID is %d \n",oRecord.EmpId);
printf(" Employee ID is %f \n",oRecord.salary);
```

Structure and arrays

- struct Employee records[10]; // Declares array of 10 Employees

- Can be accessed by indexing:

  records[0] // To access first record

  records[5] // To access record at 5 position.

- Array of structure can be initialized as:

```
struct Employee   records[10] ={
                    {“Akash”, 12345 ,100000.00  }, //position 0
                                        .
                                        .
                      {“Nidhi”, 56789 ,10000000.0 } //position 9
                  } ;
```

Structures and Functions

- Function prototype

- Function returning student type
  - struct student read_student(void);

- Structure object as an argument to a function
  - void print_student (struct student);

- Structure pointer as an argument to a function
  - void read_student_p(struct student *);

- Example continued:

Structures and Functions: example2

```
struct student          {
        char name [30];
        float marks ;
};

struct student read_student(void);

struct student read_student (struct student);

struct student  read_student_p(struct student *);
```

struct student read_student(void);

Among these two cases which has no bugs

Case : 1

```
struct student read_student(   void   )   {
        struct student student2;
        printf("Enter students name");
        gets(student2.name);
        printf("Enter marks");
        scanf("%f",&student2.marks);
        return (student2);
}
```

Case : 2

```
struct student student2; //globally declared

void read_student(   void   )   {
        printf("Enter students name");
        gets(student2.name);
        printf("Enter marks");
        scanf("%f",&student2.marks);
}
```

struct student read_student (struct student);

Case : 3

```
read_student(StudObj);  //called In main

struct student  read_student(struct student  Studdent1){
        printf("Enter students name");
        gets(Student1.name);
        printf("Enter marks");
        scanf("%f",&Student1->marks);
        return Student1;
}
```

struct student read_student_p(struct student *);

Case : 4

```
read_student_p(& StudObj);  //called In main

struct student  read_student_p(struct student * StudPtr){
        printf("Enter students name");
        gets(StudPtr->name);
        printf("Enter marks");
        scanf("%f",&StudPtr->marks);
         return *StudPtr; // no need ?
}
```

- What is the difference between case 3 and case 4?

- Which one is better with respect to memory and time optimization.?

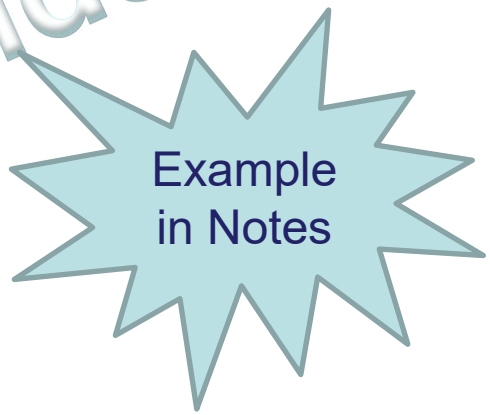
Structures and Functions: example2

```
void read_student_p(struct student *student2)
{
        printf("Enter students name");
        gets(student2->name);
        printf("Enter marks");
        scanf("%f",&student2->marks);
}
```

Structure as structure member

```
struct address
{
        int         rollnumber;
        char        street[30];
        char   city[30];
};

struct student
{
        char         name[30];
        float        marks;
        struct address adr;
};
```

Nested structure

- Structure can be nested.
- Creating instance or (pointer) of inner structure is must

```
struct student
{
        char        name[30];
        float       marks;
        struct address
        {
                int         rollnumber;
                char        street[30];
                char        city[30];
        }  adr   ;
};
```

Microsoft Word Document

Pointers and structures – case -I

- Pointers to structures is very useful in dynamic object allocation.
- Sending and returning objects to and from functions.

```
struct Date{
        int   d;    int   m;   int   y;
};
main()    {
        struct Date          *p;
        p = (struct Date *) malloc(sizeof(struct Date));
        scanf("%d %d %d",&p->d,&p -> m,&p ->y);
        printf("%d %d %d",p ->d,p ->m,p ->y);
}
```

Pointers and structures – case -II

- Pointers can be used as structure members.

```
struct sample
{
        int     *ip;
        float   *fq;
}*ptr;
```

Example

- Therefore, there are three ways to access a structure member:
  - Using the structure name
  - Using a pointer to the structure with the indirection operator (*)
  - Using a pointer to the structure with the indirect membership operator (->)

- If p_str is a pointer to the structure str, the following expressions are all equivalent to str.memb :
  - (*p_str).memb
  - p_str->memb

Pointers and Arrays of Structures

- We've seen that arrays of structures can be a very powerful programming tool, as can pointers to structures.
- The two can be combined, using pointers to access structures that are array elements.

```
struct student        {
            char            name[30];
            float           marks;
}  data[10];
```

- struct strudent *p_studt;
- p_studt = &data[0];

- Recall that the name of an array without brackets is a pointer to the first array element, so the second line could also have been written as
- p_part = data;

typedef and Structures

- A typedef keyword can be used to create a synonym for a structure or union type.
- For example, the following statements define Emp as a synonym structure:

```
typedef struct Employee  {
    char name[50];
    int    EmpId;
    float salary;
} Emp;
OR
typedef struct Employee  Emp;
```

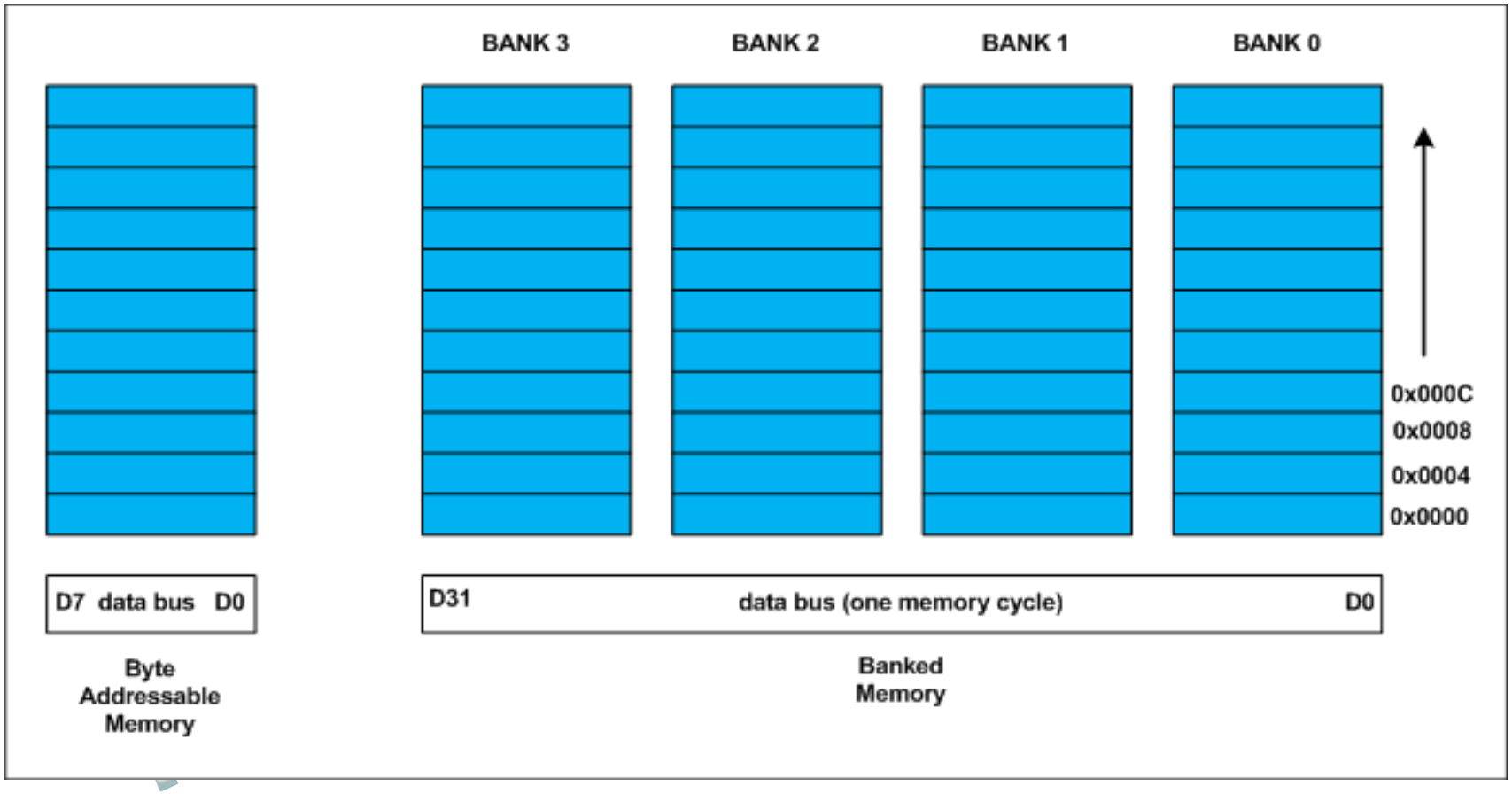
To create Object :

```
Emp   oAkash, arrEmp[10];   //We need not use key word struct
```

Structure Member Alignment, Padding and Data Packing

- Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language).
- A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.
- Historically memory is byte addressable and arranged sequentially.
- If the memory is arranged as single bank of one byte width, the processor needs to issue 4 memory read cycles to fetch an integer.
- It is more economical to read all 4 bytes of integer in one memory cycle.

Data Alignment:

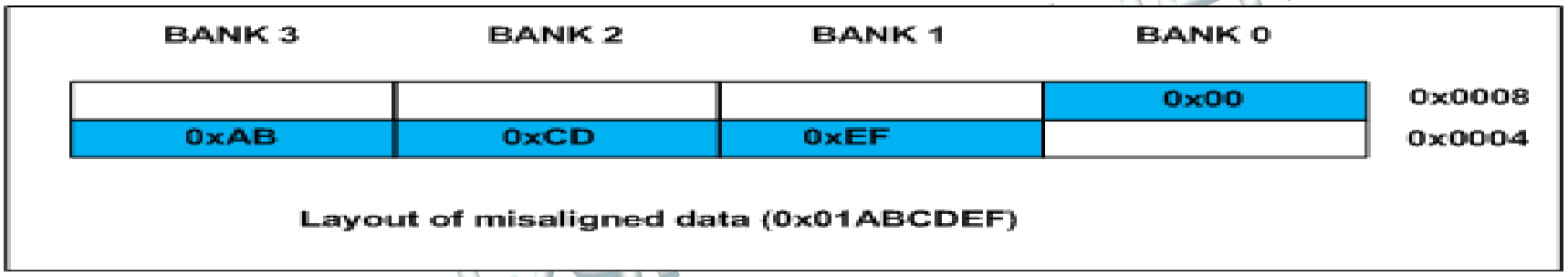


- To take such advantage, the memory will be arranged as group of 4 banks as shown in the above figure.

Data Alignment:

- Where as, if the integer is allocated at an address other than multiple of 4, it spans across two rows of the banks as shown in the below figure.

- Such an integer requires two memory read cycle to fetch the data.



- A variable's ***data alignment*** deals with the way the data stored in these banks. For example, the natural alignment of ***int*** on 32-bit machine is 4 bytes.

- When a data type is naturally aligned, the CPU fetches it in minimum read cycles.

Structure Padding

- Because of the alignment requirements of various data types, every member of structure should be naturally aligned.

```
typedef struct structa_tag
{
   char       c;
   short int   s;
} structa_t;
```

- The total size of struct_t will be sizeof(char) + 1 (padding) + sizeof(short), 1 + 1 + 2 = 4 bytes.
- The compiler will insert a padding byte after the char to ensure short int will have an address multiple of 2 (i.e. 2 byte aligned).

Structure Padding

```
// structure B
typedef struct structb_tag
{
   short int   s;
   char        c;
   int         i;
} structb_t;
```

We need 1 byte padding after the char member to make the address of next int member is 4 byte aligned.

On total, the *structb_t* requires 2 + 1 + 1 (padding) + 4 = 8 bytes.

Structure Padding [Every structure will also have alignment requirements]

```
// structure C
typedef struct structc_tag    {
   char          c;
   double      d;
   int           s;
} structc_t;
```

- Applying same analysis, *structc_t* needs sizeof(char) + 7 byte padding + sizeof(double) + sizeof(int) + 4 byte padding = 1 + 7 + 8 + 4 + 4 = 24 bytes.
- However, the sizeof(structc_t) will be 24 bytes.
- It is because, along with structure members, structure type variables will also have natural alignment.

Structure Padding [Every structure will also have alignment requirements]

- structc_t structc_array[3];
- Assume, the base address of *structc_array* is 0x0000.
- If the structc_t occupies 20 (0x14) bytes as we calculated, the second structc_t array element (indexed at 1) will be at 0x0000 + 0x0014 = 0x0014.
- It is the start address of index 1 element of array. The double member of this structc_t will be allocated on 0x0014 + 0x1 + 0x7 = 0x001C (decimal 28) which is not multiple of 8 and conflicting with the alignment requirements of double.
- Inorder to avoid such misalignment, compiler will introduce alignment requirement to every structure. It will be as that of the largest member of the structure.

What is the size of this structure?

```
// structure D
typedef struct structd_tag
{
   double      d;
   int         s;
   char        c;
} structd_t;
```

Ans: 16 Bytes

How to avoid structure padding in C?

- Many compilers provide an attribute (gcc packed), pragma, and/or command line option to override the behavior.

- In gcc compiler:

```
typedef struct {
        // structure elements
} __attribute__((__packed__)) unaligned_struct;
```

How to avoid structure padding in C?

- #pragma pack (1) directive can be used for arranging memory for structure members very next to the end of other structure members (available in gcc).

- VC++ supports this feature.
- Check your compiler documentation!


```
#pragma pack(1)

typedef struct {
        // structure elements
} struct_t
```

Example structure packing

structure packing
Ex

C:\Users\vikas.karanth\Desktop\sample.exe

```
size of structure1 in bytes : 16

   Address of id1         = 2293424
   Address of id2         = 2293428
   Address of name        = 2293432
   Address of c           = 2293433
   Address of percentage  = 2293436

size of structure2 in bytes : 16

   Address of id1         = 2293408
   Address of name        = 2293412
   Address of id2         = 2293414
   Address of c           = 2293418
   Address of percentage  = 2293420
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

union

- Unions are declared using the keyword union followed by the declarations of the union's members, enclosed in braces.

- Union is derived type of structure.

- Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.

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

- The variable u will be large enough to hold the largest of the three types;

union

- Syntactically, members of a union are accessed as

  union-name.member

  or

  union-pointer->member

- If the variable utype is used to keep track of the current type stored in U, then one might see code such as

```
if (utype == INT)       printf("%d\n", u.ival);
if (utype == FLOAT)     printf("%f\n", u.fval);
if (utype == STRING)    printf("%s\n", u.sval);
else     printf("bad type %d in utype\n", utype);
```

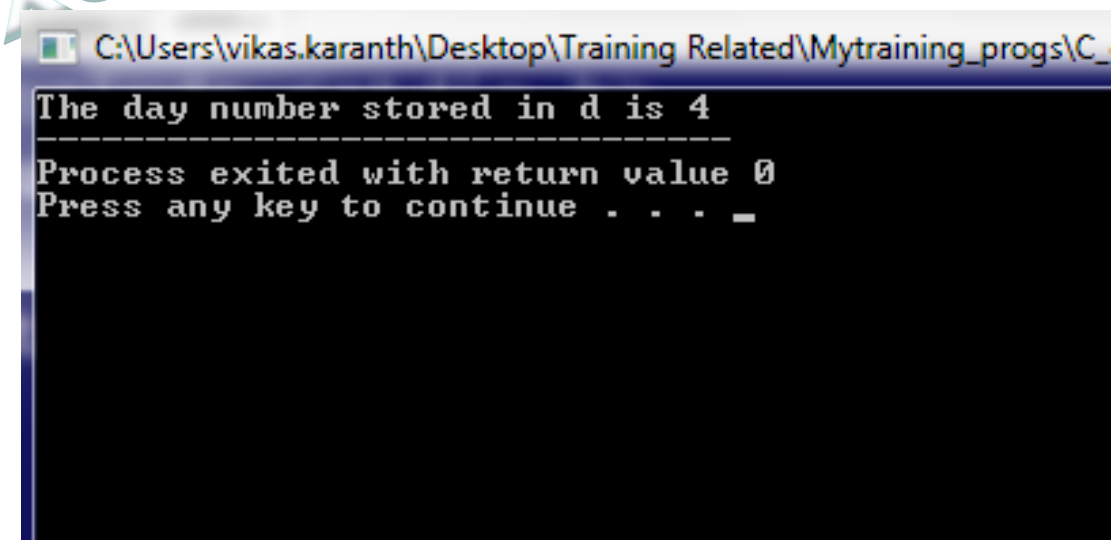
enumerations

- *Enumerations* are user defined types that have integral values; associated with each enumeration is a set of named constants
- Enumerations behave like integer constants.
- enum type_name{ value1, value2,...,valueN };
- Here, type_name is the name of enumerated data type or tag.
- value1,value2,....,valueN are values of type type_name.
- By default, value1 will be equal to 0, value2 will be 1 and so on but, the programmer can change the default value.

Example : enum

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```



enum

- We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

```
enum day {sunday = 1, monday, tuesday = 5, wednesday, thursday = 10, friday,
saturday};

int main()
{
    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday, wednesday, thursday,
friday, saturday);
    return 0;
}
```

enum

- Two enum names can have same value.
- We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.
- The value assigned to enum names must be some integeral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.
- All enum constants must be unique in their scope. For example, the following program fails in compilation.

All enum constants must be unique in their scope.

- The symbolic constants declared in the enum should be unique

- enum state {working, failed};
- enum result {failed, passed};

- int main() { return 0; } // This program gives compile time error

Quiz - time

- Predict the output of following C programs
- Program 1:

```
#include <stdio.h>
enum day {sunday = 1, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Quiz - time

Program 2:

```
#include <stdio.h>
enum State {WORKING = 0, FAILED, FREEZED};
enum State currState = 2;

enum State FindState() {
    return currState;
}

int main() {
   (FindState() == WORKING)? printf("WORKING"): printf("NOT WORKING");
   return 0;
}
```

Quiz - time

- Predict the output of following C programs
- Program 3:

```
#include <stdio.h>
enum day {sunday = 1, tuesday, Wednesday=-91, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Bit fields

- In C, we can specify size (in bits) of structure and union members.

- The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

- // A space optimized representation of date

```
struct date
{
   // d has value between 1 and 31, so 5 bits are sufficient
   unsigned int d: 5;
    // m has value between 1 and 12, so 4 bits are sufficient
   unsigned int m: 4;
   unsigned int y;
};
```

- A special unnamed bit field of size 0 is used to force alignment on next boundary.
- We cannot have pointers to bit field members as they may not start at a byte boundary.
- It is implementation defined to assign an out-of-range value to a bit field member.
- Bit fields cannot be static.
- Array of bit fields is not allowed.

A special unnamed bit field of size 0 is used to force alignment on next boundary.

```
struct test2
{
   unsigned int data1: 5;
   unsigned int: 0;
   unsigned int data2: 8;
};
```

Size is 8 bytes

We cannot have pointers to bit field members as they may not start at a byte boundary.

```
#include <stdio.h>
struct test
{
   unsigned int x: 5;
   unsigned int y: 5;
}Obj;

main()
{
     scanf("%d %d",&Obj.x,&Obj.y);
}
```

[Error] cannot take address of bit-field 'x'

It is implementation defined to assign an out-of-range value to a bit field member.

```
struct test
{
   unsigned int x: 5;
   unsigned int y: 5;
}Obj;

main()
{
    Obj.x=100;
    Obj.y=100;
}
```

- **[Warning] large integer implicitly truncated to unsigned type [-Woverflow]**

Bit fields cannot be static.

```
#include <stdio.h>
struct test
{
   static unsigned int x: 5;
   unsigned int y: 5;
}Obj;

main()
{
    Obj.x=10;
    Obj.y=10;
}
```

- **[Error] expected specifier-qualifier-list before 'static'**

Array of bit fields is not allowed.

```
#include <stdio.h>
struct test
{
   unsigned int x[10]: 5;
   unsigned int y: 5;
}Obj;
main()
{
    Obj.x=10;
}
```

[Error] bit-field 'x' has invalid type

Quiz - time

- Assume that unsigned int takes 4 bytes and long long takes 8 bytes.

```
#include <stdio.h>
struct test
{
   unsigned int x;
   unsigned int y: 33;
   unsigned int z;
};
```

- Output : Compilation time error
- It limits to word size

- How to make this run?

Quiz - time

```
union test
{
  unsigned int x: 3;
  unsigned int y: 3;
  int z;
};
```

```
int main()
{
   union test t;
   t.x = 5;
   t.y = 4;
   t.z = 1;
   printf("t.x = %d, t.y = %d, t.z = %d",
          t.x, t.y, t.z);
   return 0;
}
```

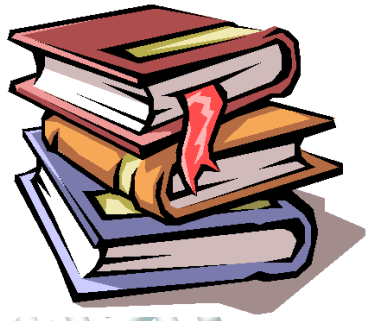
Quiz - time

```
struct test1
{
   unsigned int data1: 5;
   unsigned int data2: 8;
};
```

```
struct test2
{
   unsigned int data1: 5;
   unsigned int: 0;
   unsigned int data2: 8;
};
```

```
int main()
{
   printf("Size of test1 is %d bytes\n", sizeof(struct test1));
   printf("Size of test2 is %d bytes\n", sizeof(struct test2));
   return 0;
}
```

references



- The GNU C Reference Manual : https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

- The GNU C Reference Manual : http://www.cprogramming.com/

- C Programming Language

by Brian W. Kernighan , Dennis Ritchie

- C: The Complete Reference

by Herbert Schildt

Thank you

TATA ELXSI

ITPB Road Whitefield
Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com