CHAPTER 11

STRUCTURES, UNIONS, AND ENUMERATIONS

1. what Is the similarity between a structure, union and an enumeration?

Ans: All of them let you define new data type.

2. Would the following declaration work

```
typedef  struct s
{
  int a;
  float b;
}s;
```

Ans: YES.

3. can a structure contain a pointer to itself?

Ans: Certainly, such structures are called self-referential structure.

4. point out error if any in the following code

```
typedef struct
{
  int data;
  NODEPTR link;
}*NODEPTR;
```

Ans: A typedef defines a new name for a type, and in similar cases like the one shown bellow you can define a new structure type and a typedef for it at the same time.

```
typedef struc
{
  char name[20];
   int age;
}emp;
```

However, in the structure defined in Q4 there is an error because a typedef

declaration can not be used until it is defined. In the given code fragment the typedef declaration is not yet defined at the point where the link field is declared.

5. How will you eliminate the problem in Q4 above?

Ans: To fix this code, first give the structure name (“struct node”). Then declare the link field as a simple struct node * as shown bellow:

```
typedef struct node
{
  int data;
  struct node *link;
}*NODEPTR;
```

Another way to eliminate the problem is to disentangle the typedef declaration from the structure definition as shown bellow:

```
struct node
{
  int data;
  struct node *link;
};
typedef struct node *NODEPTR;
```

Yet another way to eliminate the problem is to precede the struct declaration with the typedef, in which case you could use the NODEPTR typedef when declaring the link field as bellow:

```
typedef structure node *NODEPTR;
struct node
{
  int data;
  NODEPTR next;
};
```

In this case, you declare a new typedef name involving struct node even though struct node has not been completely defined yet; this you’re allowed to do. It is a matter of style which of the above solutions would prefer.

6. Point out error if any in the following code.

```
void modify(struct emp *)
struct emp
{
  char name[20];
  int age;
};
main()
{
  struct emp e = {“sanjay”,35};
```

```
 modify(&e);
 printf("\n %s  %d", e.name, e.age);
}
void modify(struct emp *p)
{
 strupr(p->name);
 p->age=p->age+2;
}
```

Ans: structure emp is mentioned in the prototype of the function modify() before defining the structure. To solve the problem just put the prototype after declaration of the structure or the just add the statement struct emp before the prototype.

7. Would the following code work?

```
#include<alloc.h>
struct emp
{
  int len;
  char name[1];
};
main()
{
 char newname[]= "Rahul";
 struct emp p= (struct emp*) malloc (sizeof(struct emp)-1+ strlen(newname)+1);
 p->len=strlen(newname);
 strcpy(p->name,newname);
 printf("\n %d  %s", p->len, p->name);
}
```

Ans: YES, the program allocates space for the structure with the size adjusted so that the name field can hold the requested name (not just one character declaration would suggest). I don't know whether it is legal or portable. However, the code did work on all the compilers that I have tried it with.

8. can you suggest a better way to write the program in Q7 above?

Ans:- The truly safe way to implement the program is to use a char pointer instead of an array

```
#include<alloc.h>
```

```
struct emp
{
  int len;
  char *name;
};

main()
{
 char newname[]=”Rahul”;
  struct emp *p=(struct emp*) malloc(sizeof(struct emp));

  p->len=strlen(newname);
  p->name=malloc(p->len+1);
  strcpy(p->name,newname);
  printf(“\n%d  %s”,p->len,p->name);
}
```

Obviously, the convenience of having the length and the string stored in the same block of memory has now been lost, and freeing instances of this structure will require two calls to the function free();

9. How would you free the memory allocated in Q8 above?

Ans:- free(p->name);
free(p);

10. Can you rewrite the program in Q8 such that while freeing the memory only one call to free() would suffice?

Ans:-

```
#include<alloc.h>
struct emp
{
  int len;
  char *name;
}

main()
{
  char newname:- “Rahul”;
  char *buf=malloc(sizeof(strut emp)+strlen(newname)+1);
  struct emp *p= (struct emp *) buf;
  p->len=strlen(newname();
```

```
  p->name=buf+sizeof(struct emp);
  strcpy(p->name,newname);
  printf("\n %d %s",p->len,p->name);
  free(p);
}
```

11. O/p?

```
main()
{
  struct emp
   {
     char  *n;
     int age;
    };
  struct emp e1={"Dravid",23};
  struct emp e2=e1;
  strupr(e2.n);
  printf("\n%s",e1.n);
}
```

Ans:- DRAVID

When a structure is assigned, passed, or returned, the copying is done monolithically. This means that the copies of any pointer fields will point to the same place as the original. In other words, anything pointed to is not copied. Hence on changing the name through e2.n it automatically changed e1.n.

12. Point out error if any in the following code.

```
main()
{
  struct emp
   {
     char n[20];
     int age;
   };
  struct emp e1= {"Dravid",23};
  struct emp e2=e2;
  if(e1==e2)
     printf("\n The structures are equal");
}
```

Ans:- Structures can not be compared using the built in == and != operations. This is because there is no single, good way for a compiler to implement structure comparison. A simple bute by byte comparison could fail while comparing the bits present in unused padding in the structure (such padding is used to keep the

alignment of letter fields correct). A field by field comparison might require unacceptable amounts of repetitive code for large structures. Also any compiler generated comparison would not be expected to compare pointer fields appropriately in all cases; for example, it is often appropriate to compare char* fields with strcmp() rather than with ==;

13. How would you check whether the contents of two structure variable are same or not?

Ans:-

```
Struct emp
{
  char n[20];
  int age;
};
main()
{
  struct emp e1={“Dravid”,23};
  struct emp e2;
  scanf(“%s %d”,e2.n,e2.age);
  if(structcmp(e1,e2)==0)
    printf(“\n Equal”);
  else
    printf(“\n Unequal”);
}

structcmp(struct emp x, struct emp y)
{
  if(strcmp(x.n,y.n)==0)
    if(x.age==y.age)
       return(0);
  return(1);
}
```

14. How are structure passing and returning implemented by the compiler?

Ans:- When structures are passed as arguments to functions, the entire structure is typically pushed on the stack. To avoid the overhead many programmers often prefer to pass pointers to structures instead of structures. Structures are often returned from functions in a location pointed by an extra, compiler supplied “hidden” argument to the function.

15. How can I read/write structures from/to data files?

Ans:- To write out a structure we can use fwrite() as shown bellow:
Fwrite(&e, sizeof(e), 1, fp);
Where e is the structure variable. A corresponding fread() invocation can read the structure back from the file.
On calling fwrite() it writes out sizeof(e) bytes from the address &e. Data files are written as memory images with fwrite(), however will not be portable, particularly if they contain floating point fields or pointers. This is because the memory layout of structures is machine and compiler dependent. Different compilers may use different amount of padding, and the size and byte orders of fundamental types vary across machines. Therefore structures written as memory images cannot necessarily be read back in by programs running on other machines (or even compiled by other compilers) and this is an important concern if the data files you're writing will ever be interchanged between machines.

16. If the following structure is written to a file using fwrite(), can fread() read it back successfully?

```
Struct emp
{
 char *n;
 int age;
};
struct emp e= {"Sujay",15};
file *fp;
fwrite(&e, sizeof(e), 1, fp);
```

Ans:- No, since the structure contains a char pointer while writing a structure to the disk using fwrite() only the value stored in the pointer n would get written (and not the string pointed by it). When this structure is read back the address will be read back but it is quite unlikely that the desired string would be present at this address in memory.

17. Would the following program always output the size of structure as 7 bytes?

```
Struct ex
{
 char ch;
 int i;
 long int a;
};
```

Ans:- No. A compiler may leave holes in structures by padding the first char in

the structure with another byte just to ensure that the integer that follow is stored at an even location. Also there might be two extra bytes after the integer to ensure that the long integer is stored at an address which is a multiple of 4. This is done because many machines access values in memory most efficiently when the values are appropriately aligned. Some machines can not perform unaligned accesses at all and require that all the data be appropriately aligned.

Your compiler may provide an extension to give you control over the packing of structures (i.e. whether they are padded), perhaps with a #pragma, but there is no standard method.

If you're worried about wasted space, you can minimize the effects of padding by ordering the members of the structure from largest to smallest. You can sometimes get more control over size and alignment by using bitfields, although they have their own drawbacks.

18. What error does the following program give and what is the solution for it?

```
main()
{
 struct emp
   {
   char name[20];
   float sal;
 };
 struct emp e[10];
 int i;
 for(i=0;i<10;i++)
    scanf("%s %f", e[i].name, &e[i].sal);
}
```

Ans:- Error :Floating point formats not linked. What causes this error to occur? When the compile encounters a reference to the address of the float, it sets a flag to have the linker link the floating point emulator.

A floating point emulator is used to manipulate floating point numbers in runtome library functions like scanf() and atof(). There are some cases in which the reference to a float is a bit obscure and the compiler does not detect the need for the emulator.

These situations usually occur during the initial stages of program development. Normally once the program is fully developed, the emulator will be used in such a fashion that the compiler can accurately determine when to link the emulator.

To force linking of the floating point emulator into an application, just include the following function in your program.

```
Void linkfloat(void)
{
 float a=0, b=&a;   / causes emulator to be linked*/
 a=*b;  /*suppress warning –var not used*/
```

}
There is no need to call this function in your program.

19. How can I determine the byte offset of a field within a structure?

Ans:- You can use the offset macro given bellow. How to use this macro has also been shown in the program.

```
#define offset(type, mem) ( (int) ( (char*) & ( ( type*) 0)- mem – (char*) (type*) 0 ) )

main()
{
 struct a
 {
   char name[15];
   int age;
   float sal;
 }
 int offsetofname, offsetofage, offsetofsal;

 offsetofname=offset(struct a, name);
 printf(“\n%d”, offsetofname);

 offsetofage=offset(struct a, age);
 printf(“\n%d”,offsetofage);

 offsetofsal=offset(struct a, sal);
 printf(“\n%d”,offsetofsal);
}
```

The output of this program will be

```
0
15
17
```

20. The way mentioning the array name or function name without [ ] or ( ) yields their base address, what do you obtain on mentioning the structure name?

Ans:- The entire structure itself and not it’s base address.

21. What is main returning in the following program

```
struct transaction
```

```
{
   int sno;
   char desc[30];
  char dc;
  float amount;
}
main(int argc, char *argv[])
{
  struct transaction t;
  scanf("%d %s %c %f", &t.sno, t.desc, &t.dc, &t.amount);
  printf("%d %s %c %f", t.sno, t.desc, t.dc, t.amount);
}
```

Ans:- A missing semicolon at the end of the structure declaration is causing main to be declared as returning a structure.

22. O/p?

```
main()
{
  struct a
  {
    category : 5;
    scheme : 4;
  };
  printf("size= %d", sizeof(struct a));
}
```

Ans:- size=2

Since we have used bit fields in the structure and the total number of bits is turning out to be more than 8 the sizeof the structure is being reported as 2 bytes.

23. What is the difference between a structure and a union?

Ans:- A union is essentially a structure in which all of the fields overlay each other; you can use only one field at a time. You can also write to one field and read from another, to inspect a type's bit patterns or interpret them differently.

24. Is it necessary that the size of all elements in a union should be same?

Ans:- No. Union elements can be of different sizes. If so, size of the union is size

of the longest element in the union. As against this the size of the structure is the sun of the size of it's members. In both cases, the size may be increased by padding.

25. Point out the error, if any

```
main()
{
  union a
   {
    int i;
    char ch[2];
   };
  union a z1={512};
  union a z2= {0, 2};
}
```

Ans:- The ANSI C standard allows an initializer for the first member of a union. There is no standard way of initializing any other member, hence the error in initializing z2.

Many proposals have been advanced to allow more flexible union initialization, but none has been adopted yet. If you still want to initialize different members of the union then you can define several variant copies of a union, with the members in different orders, so that you can declare and initialize the one having the appropriate first member as shown bellow.

```
Union a
 {
   int i;
   char ch[2];
  };
union b
{
  char ch[2];
  int i;
};
main()
{
  union a z1={512};
  union b z2={0,2};
}
```

26. What is the difference between an enumeration and a set of preprocessor #defines?

Ans:- There is hardly any difference between the two, except that the #define has a global effect (throughout the file) whereas the enumeration can have an effect local to the block if desired. Some advantages of enumerations are that the numeric values are automatically assigned whereas in #define we have to explicitly define them. A disadvantage is that we have no control over the sizes of enumeration variables.

27. Since enumerations have integral type and enumeration constants are of type int can we freely intermix them with other integral types, without errors?

Ans:- Yes.

28. Is there an easy way to print enumeration values symbolically?

Ans:- No. You can write a small function, one pre enumeration, to map an enumeration constant to a string, either by using a switch statement or by searching an array.

29. What is the use of bit fields in structure declaration?

Ans:- Bit-fields are used to save space in structures having several binary flags or other small fields. Note that the colon notation for specifying the size of a field in bits is valid only in structures and in unions; you cannot use this mechanism to specify the size of arbitrary variables.

30. Can we have an array of bit fields?

Ans:- No.