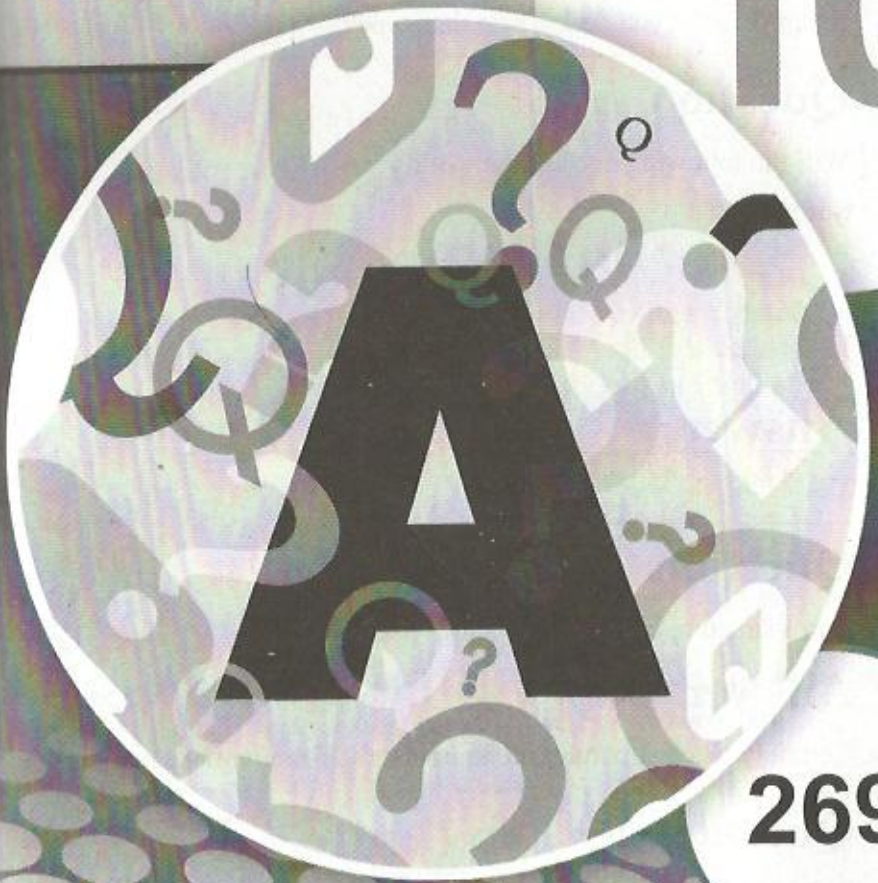
Structures, Unions & Enums

10



269

Question *10.1*

What is the similarity between a structure, union and an enumeration?

Answer

All of them let you define new data types.

Question *10.2*

Will the following declaration work?

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

Answer

Yes

Question *10.3*

Can a structure contain a pointer to itself?

Answer

Certainly. Such structures are known as self-referential structures.

Question *10.4*

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

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


Answer

A *typedef* defines a new name for a type, and in simpler cases like the one shown below you can define a new structure type and a *typedef* for it at the same time.

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

However, in the structure defined in Question 10.4 there is an error because a *typedef* declaration cannot 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.

Question *10.5*

How will you eliminate the problem in 10.4 above?

Answer

To fix this code, first give the structure a name—"node". Then declare the *link* field as a simple *struct node* * as shown below:

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

Another way to eliminate the problem is to disentangle the *typedef* declaration from the structure declaration as shown below:

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

Yet another way to eliminate the problem is to precede the structure declaration with the *typedef*, in which case you can use the *NODEPTR typedef* when declaring the *link* field as shown below:

```
typedef struct 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 declared yet; this you are allowed to do. It is a matter of style which of the above solutions will you prefer.

Question *10.6*

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

```
#include <stdio.h>
#include <string.h>
void modify ( struct emp * ) ;
```



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

int main( )
{
    struct emp e = { "Sanjay", 35 } ;
    modify ( &e ) ;
    printf ( "%s %d\n", e.name, e.age ) ;
    return 0 ;
}
void modify ( struct emp *p )
{
    strupr ( p -> name ) ;
    p -> age = p -> age + 2 ;
}
```

Answer

No error. The program would output SANJAY 37. Some compilers may give a warning since *struct emp* is mentioned in the prototype of the function *modify( )* before declaring the structure. To solve the problem just put the prototype after the declaration of the structure or just add the statement *struct emp* before the prototype.

Question *10.7*

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

```
struct emp
{
    int ecode ;
    struct emp e ;
} ;
```

Answer

Error. The structure *emp* contains a member *e* of the same type, i.e. *struct emp*. At this stage the compiler doesn't know the size of the structure.

Question *10.8*

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

```
struct emp
{
    int ecode ;
    struct emp *e ;
} ;
```

Answer

No Error. This type of structure is known as self-referential structure. Here, the structure *emp* contains *e* a pointer to *struct emp*. The compiler knows the size of the pointer to a structure even before the size of the structure is determined (since the size of any pointer is 2 bytes (under DOS) and 4 bytes (under Windows/Linux)).

Question *10.9*

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

```
typedef struct data mystruct ;
struct data
{
    int x ;
    mystruct *b ;
} ;
```


Answer

No error. Here, the typename *mystruct* is known at the point of declaring the structure, as it is already defined.

Question *10.10*

Will the following code work?

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
struct emp
{
    int len ;
    char name[1] ;
} ;
int 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 ( "%d %s\n", p -> len, p -> name ) ;
    return 0 ;
}
```

Answer

Yes. The program allocates space for the structure with the size adjusted so that the *name* field can hold the requested name.

Question *10.11*

Can you suggest a better way to write the program in 10.10 above?

Answer

The truly safe way to implement the program is to use a character pointer instead of an array as shown below:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct emp
{
    int len ;
    char *name ;
} ;
int main( )
{
    char newname[ ] = "Rahul" ;
    struct emp *p = ( struct emp * ) malloc ( sizeof ( struct emp ) ) ;
    p -> len = strlen ( newname ) ;
    p -> name = ( char * ) malloc ( p -> len + 1 ) ;
    strcpy ( p -> name, newname ) ;
    printf ( "%d %s\n", p -> len, p -> name ) ;
    return 0 ;
}
```

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( )*.

Question *10.12*

How will you free the memory allocated in 10.11 above?

Answer

```
free ( p -> name ) ;
```



```
free ( p ) ;
```

Question *10.13*

Can you rewrite the program in 10.11 such that while freeing the memory only one call to *free( )* will suffice?

Answer

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct emp
{
    int len ;
    char *name ;
} ;
int main( )
{
    char newname[ ] = "Rahul" ;
    char *buf = ( char * ) malloc ( sizeof ( struct 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 ( "%d %s\n", p -> len, p -> name ) ;
    free ( p ) ;
    return 0 ;
}
```

Question *10.14*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    struct value
    {
        int bit1 : 1 ;
        int bit3 : 4 ;
        int bit4 : 4 ;
    } bit ;
    printf ( "%d\n", sizeof ( bit ) ) ;
    return 0 ;
}
```

A. 1
B. 2
C. 4
D. 9

Answer

B in TC/TC++ and C in gcc under Linux and Visual Studio under Windows.

Question *10.15*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    struct value
    {
        int bit1 : 1 ;
        int bit3 : 4 ;
        int bit4 : 4 ;
    } bit = { 1, 2, 2 } ;
    printf ( "%d %d %d\n", bit.bit1, bit.bit3, bit.bit4 ) ;
    return 0 ;
}
```

A. 1 2 2
B. 0 2 2
C. -1 2 2
D. 1 -2 -2

Answer

C

Question *10.16*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    enum value { VAL1 = 0, VAL2, VAL3, VAL4, VAL5 } var ;
    printf ( "%d\n", sizeof ( var ) ) ;
    return 0 ;
}
```

A. 1
B. 2
C. 4
D. 10

Answer

B in TC/TC++, and C in gcc under Linux and Visual Studio under Windows.

Question *10.17*

What will be the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main( )
{
    struct emp
    {
        char *n ;
        int age ;
    } ;
    struct emp e1 = { "Dravid", 23 } ;
    struct emp e2 = e1 ;
    strupr ( e2.n ) ;
    printf ( "%s\n", e1.n ) ;
    return 0 ;
}
```

Answer

Causes exception

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. i.e. contents of *e2.n* (address of Dravid) is copied into *e1.n*. Moreover, n is a pointer to a constant string. So when we attempt to change this constant string into uppercase an error is reported at runtime.

To eliminate this error we should define *n* as an array as shown below:

```
struct emp
{
    char n[ 20 ] ;
    int age ;
} ;
```


Question *10.18*

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

```
#include <stdio.h>
int main( )
{
    struct emp
    {
        char n[20] ;
        int age ;
    } ;
    struct emp e1 = { "Dravid", 23 } ;
    struct emp e2 = e1 ;
    if ( e1 == è2 )
        printf ( "The structures are equal\n" ) ;
    return 0 ;
}
```

Answer

Structures can't be compared using the built-in == and != operations.

Question *10.19*

How will you check whether the contents of two structure variables are same or not?

Answer

```
#include <stdio.h>
#include <string.h>
struct emp
{
    char n[20] ;
    int age ;
} ;
structcmp ( struct emp, struct emp ) ;
int main( )
{
    struct emp e1 = { "Dravid", 23 } ;
    struct emp e2 ;
    scanf ( "%s %d", e2.n, &e2.age ) ;
    if ( structcmp ( e1, e2 ) == 0 )
        printf ( "The structures are equal\n" ) ;
    else
        printf ( "The structures are unequal\n" ) ;
    return 0 ;
}

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

In short, if you need to compare two structures, you will have to write your own function which carries out the comparison field by field.

Question *10.20*

How are structure passing and returning implemented by the compiler?

Answer

When structures are passed as arguments to functions, the entire structure is pushed on the stack. For big structures this is an extra overhead. This overhead can be avoided by passing pointers to structures instead of actual structures. To return structures a hidden argument generated by the compiler is passed to the function. This argument points to a location where the returned structure is copied.

Question *10.21*

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

Answer

To write out a structure we can use *fwrite( )* as shown below:

```
fwrite ( &e, sizeof ( e ), 1, fp ) ;
```

where *e* is a structure variable. A corresponding *fread( )* invocation can read the structure back from a file.

On calling *fwrite( )* it writes out *sizeof (e)* bytes starting from the address *&e*. A word of caution—data files written as memory images with *fwrite( )* may not be portable, especially if they contain floating-point fields or pointers. This is because of the following reasons:

- How the structures are laid out in memory may vary from one machine to another.

- How the structures are laid out in memory may vary from one compiler to another.
- Number of bytes that are padded at the end of each structure element may vary from one compiler to another.
- The order in which the bytes are laid out for integers and floats may vary from one machine to another.

Hence structures written as memory images on one machine may not get successfully read on another machine. Also, program compiled using different compilers may give different results. This is an important concern if the data files are to used on different machines.

Question *10.22*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    enum days { MON = -1, TUE, WED = 6, THU, FRI, SAT } ;
    printf ( "%d %d %d %d %d %d\n", MON, TUE, WED, THU, FRI, SAT ) ;
    return 0 ;
}
```

A. -1 0 1 2 3 4
B. -1 2 6 3 4 5
C. -1 0 6 2 3 4
D. -1 0 6 7 8 9

Answer

D

Question *10.23*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    enum days { MON = -1, TUE, WED = 6, THU, FRI, SAT } ;
    printf ( "%d %d %d %d %d %d", ++MON, TUE, WED, THU, FRI, SAT ) ;
    printf ( "\n" ) ;
    return 0 ;
}
```

A. -1 0 1 2 3 4
B. Error
C. 0 1 6 3 4 5
D. 0 0 6 7 8 9

Answer

B. ++ or -- cannot be done on an enum value. However, MON + TUE will work.

Question *10.24*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    union var
    {
        int a, b ;
    } ;
```

```
    union var v ;
    v.a = 10 ;
    v.b = 20 ;
    printf ( "%d\n", v.a ) ;
    return 0 ;
}
```

A. 10
B. 20
C. 30
D. 0

Answer

B

Question *10.25*

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 ;
fp = fopen ( "names.dat", "wb" ) ;
fwrite ( &e, sizeof ( e ) , 1, fp ) ;
```

Answer

No, since the structure contains a *char* pointer while writing the structure to the disk using *fwrite( )* only the value stored in the pointer *n* will get written (and not the string pointed to by it). When this structure is read back the address will be read back but it is

quite unlikely that the desired string will be present at this address in memory.

Question *10.26*

If a char is one byte wide, an integer is 2 bytes wide and a long integer is 4 bytes wide, then will the following structure always occupy 7 bytes?

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

Answer

No. A compiler may leave holes in structures by padding the first *char* in the structure with three more bytes just to ensure that the integer that follows is stored at a location which is multiple of 4. Such alignment is done by machines to improve the efficiency of accessing values. In Visual Studio the size of this structure would be reported as 12 bytes.

The wastage of space due to padding of structure elements can be reduced by arranging the structure elements in the order largest to smallest while declaring the structure.

Many compilers provide a *#pragma pack* directive to give you control over the alignment of structure elements. For example if we execute the following code snippet it would output 9.

```
# pragma pack(1)
struct ex
{
    char ch ;
```

```
    int i ;
    long int a ;
} ;
```

The #pragma pack directive indicates that the elements of this structure should be aligned at addresses which are multiples of 1. Since every address is multiple of 1, the integer after character begins at immediately next address.

Question *10.27*

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

```
#include <stdio.h>
int main( )
{
    struct bits
    {
        float f : 2 ;
    } bit ;
    printf ( "%d\n", sizeof ( bit ) ) ;
    return 0 ;
}
```

Answer

Bit fields can be set only for a *signed int* or an *unsigned int*. Here, we are trying to set bit fields for a *float*, hence the error.

Question *10.28*

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

```
#include <stdio.h>
int main( )
{
```



```
    struct bits
    {
        int i : 40 ;
    } bit ;
    printf ( "%d\n", sizeof ( bit ) ) ;
    return 0 ;
}
```

Answer

Error: 'Bit field too large'. Bit fields can be set only for a *signed* or an *unsigned int*. As a result, we can supply a bit field not more than 16 bits for TC/TC++ and 32 bits for Visual Studio / gcc.

Question *10.29*

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

```
#include <stdio.h>
int main( )
{
    struct emp
    {
        char name[20] ;
        float sal ;
    } ;
    struct emp e[10] ;
    int i ;
    for ( i = 0 ; i <= 9 ; i++ )
        scanf ( "%s %f", e[ i ].name, &e[ i ].sal ) ;
    return 0 ;
}
```

Answer

Error: Floating point formats not linked. What causes this error to occur? When the compiler encounters a reference to the address of a *float*, it sets a flag to have the linker link in the floating point emulator. A floating point emulator is used to manipulate floating point numbers in runtime library functions like *scanf( )* and *atof( )*. There are some cases in which the reference to the *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 in 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 ; /* cause emulator to be linked */
    a = *b ; /* suppress warning - var not used */
}
```

There is no need to call this function from your program.

Note that this error occurs when you build and run the program in TC/TC++. No such error occurs when you build the program using gcc or Visual Studio.

Question *10.30*

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

Answer

You can use the offset macro given below. How to use this macro has also been shown in the program.

```
#include <stdio.h>
#define offset( type, mem ) ( int ) & ( ( ( type * ) 0 ) -> mem )

int 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 ( "\t%d", offsetofage ) ;
    offsetofsal = offset ( struct a, sal ) ;
    printf ( "\t%d\n", offsetofsal ) ;
    return 0 ;
}
```

If an integer is 4 bytes wide then the output of this program will be:

0 16 20

Question *10.31*

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

Answer

The entire structure itself and not its base address.

Question *10.32*

What is *main( )* returning in the following program?

```
#include <stdio.h>
struct transaction
{
    int sno ;
    char desc[30] ;
    char dc ;
    float amount ;
}
/* Here is the main program. */
int main ( int argc, char *argv[ ] )
{
    struct transaction t ;
    scanf ( "%d %s %c %f\n", &t.sno, t.desc, &t,dc, &t.amount ) ;
    printf ( "%d %s %c %f\n", t.sno, t.desc, t,dc, t.amount ) ;
    return 0 ;
}
```

Answer

A missing semicolon at the end of the structure declaration is causing *main( )* to be declared as a function returning a structure. The connection is difficult to see because of the intervening comment.

Question *10.33*

Point out the error, if any, in the following program.


```
#include <stdio.h>
int main( )
{
    struct a
    {
        category : 5 ;
        scheme : 4 ;
    } ;
    printf ( "size = %d\n", sizeof ( struct a ) ) ;
    return 0 ;
}
```

Answer

The bit fields must be of the type *signed int* or *unsigned int*. Here we have not defined any type for *category* or *scheme*. Hence the error.

Question *10.34*

What is the difference between a structure and a union?

Answer

A union is essentially a structure in which all of the fields overlay each other. At a time only one field can be used. We can write to one field and read from another.

Question *10.35*

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

Answer

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 a structure is the sum of the size of its members. In both cases the size may get increased because of padding.

Question *10.36*

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

```
#include <stdio.h>
int main( )
{
    union a
    {
        int i ;
        char ch[2] ;
    } ;
    union a z1 = { 512 } ;
    union a z2 = { 0, 2 } ;
    return 0 ;
}
```

Answer

The 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*.

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 below.

```
#include <stdio.h>
union a
{
    int i ;
    char ch[2] ;
} ;
union b
{
    char ch[2] ;
    int i ;
} ;
int main( )
{
    union a z1 = { 512 } ;
    union b z2 = { 0, 2 } ;
    return 0 ;
}
```


Question *10.37*

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

Answer

There is hardly any difference between the two, except that a *#define* has a global effect (throughout the file), whereas, an 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 of enumeration is that we have no control over the sizes of enumeration variables.

Question *10.38*

Is there an easy way to print enumeration values symbolically?

Answer

No. You can write a small function (one per enumeration) to map an enumeration constant to a string, either by using a *switch* statement or by searching an array.

Question *10.39*

What is the use of bit fields in a structure declaration?

Answer

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.

Question *10.40*

Can we have an array of bit fields? [Yes/No]

Answer

No

Question *10.41*

Which of the following statements are correct about the program given below?

```
#include <stdio.h>
int main( )
{
    union a
    {
        int i ;
        char ch [ 2 ] ;
    } ;
    union a u1 = { 512 } ;
    union a u2 = { 0, 2 } ;
    return 0 ;
}
```


A. *u2* CANNOT be initialized as shown.
B. *u1* can be initialized as shown.
C. To initialize *char ch[ ]* of u2 '.' operator should be used.
D. The code causes an error 'Declaration syntax error'.

Answer

A, B, C

Question *10.42*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
#include <malloc.h>
int main( )
{
    struct node
    {
        int data ;
        struct node *link ;
    } ;
    struct node *p, *q ;
    p = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
    q = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
    printf ( "%d %d\n" , sizeof ( p ) , sizeof ( q ) ) ;
    return 0 ;
}
```

A. 2 2
B. 8 8
C. 5 5
D. 4 4

Answer

A in TC/TC++ under DOS, and D in gcc under Linux and Visual Studio under Windows.

Question *10.43*

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

```
#include <stdio.h>
int main( )
{
    struct emp
    {
        char name[ 25 ] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;
    e.name = "Rahul" ;
    e.age = 25 ;
    printf ( "%s %d\n", e.name, e.age ) ;
    return 0 ;
}
```

Answer

We are trying to initialize string in

e.name = "Rahul" ;

This would result into an error "Lvalue required" as on the left hand side of = there is base address of the string.

Question *10.44*

Which of the following statements are correct about the program given below?

```
#include <stdio.h>
int main( )
{
    struct emp
    {
        char name[20] ;
        int age ;
        float sal ;
    } ;
    struct emp e[2] ;
    int i = 0 ;
    for ( i = 0 ; i < 2 ; i++ )
        scanf ( "%s %d %f", e[ i ].name, &e[ i ].age, &e[ i ].sal ) ;
    for ( i = 0 ; i < 2 ; i++ )
        printf ( "%s %d %f\n", e[ i ].name, e[ i ].age, e[ i ].sal ) ;
    return 0 ;
}
```

A. Error: '*scanf( )* function cannot be used for structure elements'
B. The code runs successfully.
C. Error: 'Floating-point formats not linked. Abnormal program termination'.
D. Error: 'Structure variable must be initialized'.

Answer

C in TC/TC++ under DOS, and B in gcc under Linux and Visual Studio under Windows.

Question *10.45*

Which of the following statement is correct about the statement given below?

```
maruti.engine.bolts = 25 ;
```

A. Structure *bolts* is nested within structure *engine*.
B. Structure *engine* is nested within structure *maruti*.
C. Structure *maruti* is nested within structure *engine*.
D. Structure *maruti* is nested within structure *bolts*.

Answer

B

Question *10.46*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    struct byte
    {
        int one:1;
    } ;
    struct byte var = { 1 } ;
    printf ( "%d\n", var.one ) ;
    return 0 ;
}
```

A. 1
B. -1
C. 0
D. Error

Answer

B

Question *10.47*

Which of the following statement correctly assigns 45 to *month* using pointer variable *pdt*?

```
#include <stdio.h>
struct date
{
    int day ;
    int month ;
    int year ;
} ;
int main( )
{
    struct date d ;
    struct date *pdt ;
    pdt = &d ;
    return 0 ;
}
```

A. pdt.month = 12 ;
B. &pdt.month = 12 ;
C. d.month = 12 ;
D. pdt -> month = 12 ;

Answer

D

Question *10.48*

Which of the following statement is true?

A. User has to explicitly define the numeric values of enumerations.
B. User has a control over the size of enumeration variables.
C. Enumeration can have an effect local to the block, if desired.
D. Enumerations have a global effect throughout the file.

Answer

C

Question *10.49*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    enum status { pass, fail, atkt } ;
    enum status stud1, stud2, stud3 ;
    stud1 = pass ;
    stud2 = atkt ;
    stud3 = fail ;
    printf ( "%d %d %d\n", stud1, stud2, stud3 ) ;
    return 0 ;
}
```

A. 0 1 2
B. 1 2 3
C. 0 2 1
D. 1 3 2

Answer

C

Question *10.50*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
int main( )
{
    int i = 4, j = 8 ;
    printf ( "%d %d %d\n", i | j & j | i, i | j && j | i, i ^ j ) ;
    return 0 ;
}
```

A. 12 1 12
B. 112 1 12
C. 32 1 12
D. -64 1 12

Answer

A

Question *10.51*

Which of the following is the correct output for the program given below, if size of *int* is 4 bytes?

```
#include <stdio.h>
int main( )
{
    union a
    {
```

```
        int i ;
        char ch[2] ;
    } ;
    union a u ;
    u.ch[0] = 0 ;
    u.ch[1] = 2 ;
    u.ch[1] = 0 ;
    u.ch[1] = 0 ;
    printf ( "%d\n", u.i ) ;
    return 0 ;
}
```

A. 512
B. 2
C. 0
D. None of the above

Answer

A

Question *10.52*

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

```
#include <stdio.h>
struct tryit
{
    char *str ;
    int i ;
} t ;
int main( )
{
    scanf ( "%s %d", t.str, &t.i ) ;
    printf( "%s\n", t.str ) ;
    return 0 ;
}
```


Answer

This program causes a runtime-error. Here, we are trying to store the string at the address to which **str** (of **t**) points to. Since, the pointer is not initialized it holds garbage value and so the result will be undefined. Hence the error.

Question *10.53*

Which of the following is the correct output for the program given below?

```
#include <stdio.h>
struct course
{
    int courseno ;
    char coursename[ 25 ] ;
} ;
int main( )
{
    struct course c[ ] = {
                          { 102 , "Thermal" } ,
                          { 103 , "Manufacturing" },
                          { 104 , "Design" }
                        } ;
    printf ( "%d ", c[ 1 ] .courseno ) ;
    printf ( "%s\n", ( * ( c + 2 ) ).coursename ) ;
    return 0 ;
}
```

A. 103 Design
B. 102 Thermal
C. 103 Manufacturing
D. 104 Design

Answer

A

Question *10.54*

State True or False:

A. A structure can contain similar or dissimilar elements.

B. The '.' operator can be used access structure elements using a structure variable.

C. A structure can be nested inside another structure.

D. It is not possible to create an array of pointer to structures.

E. One of the elements of a structure can be a pointer to the same structure.

F. On declaring a structure zero bytes are reserved in memory.

G. A union cannot be nested in a structure.

H. The '->' operator can be used to access structure elements using a pointer to a structure variable.

I. The alignment of structure members in memory varies from one compiler to another.

J. Memory gets allocated for the members of a structure only when a structure variable is defined.

K. Nested unions are not allowed.

L. The scope of a structure should be global if it is to be used in all functions in the program.

M. The alignment of structure members in memory can be controlled using the *#pragma pack* preprocessor directive.

N. Bit fields CANNOT be used in a *union*.

O. By default a structure variable will be of auto storage class.

P. Union elements can be of different sizes.

Q. Size of the *union* is size of the longest element in the *union*.

R. If we initialize one element of the union it also initializes other elements of the *union*.

S. *union* variables can have local scope or global scope.

T. The elements of a *union* are always accessed using & operator.

U. A pointer to a *union* CANNOT be created.

Answer

A. True
B. True
C. True
D. False
E. True
F. True
G. False
H. True
I. True
J. True

K. False
L. True
M. True
N. True
O. True
P. True
Q. True
R. True
S. True
T. False
U. False