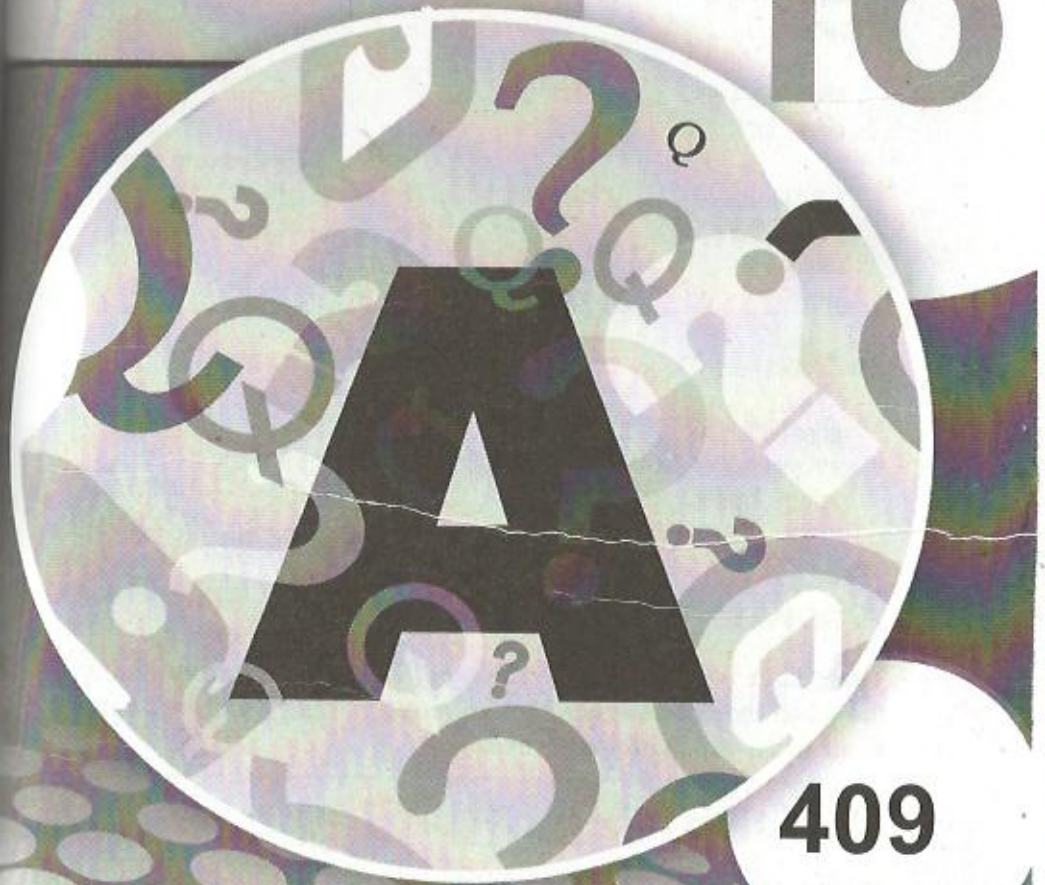
Memory Allocation

16



409

Question *16.1*

What will be the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char *s ;
    char * fun( ) ;

    s = fun( ) ;
    printf ( "%s\n", s ) ;
    return 0 ;
}

char * fun( )
{
    char buffer[30] ;
    strcpy ( buffer, "RAM - Rarely Adequate Memory" ) ;
    return ( buffer ) ;
}
```

Answer

The output is unpredictable since *buffer* is an *auto* array and will die when the control goes back to *main( )*. Thus *s* will be pointing to an array, which no longer exists.

Question *16.2*

What is the solution for the problem in program 16.1 above?

Answer

```
#include <stdio.h>
```



```
#include <string.h>
int main( )
{
    char *s ;
    char *fun( ) ;
    s = fun( ) ;
    printf ( "%s\n", s ) ;
    return 0 ;
}
char * fun( )
{
    static char buffer[30] ;
    strcpy ( buffer, "RAM - Rarely Adequate Memory" ) ;
    return ( buffer ) ;
}
```

Question *16.3*

Does there exist any other solution for the problem in 16.1?

Answer

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main( )
{
    char *s ;
    char *fun( ) ;
    s = fun( ) ;
    printf ( "%s\n", s ) ;
    free ( s ) ;
    return 0 ;
}
char *fun( )
{
    char *ptr ;
```

```
    ptr = ( char * ) malloc ( 30 ) ;
    strcpy ( ptr, "RAM - Rarely Adequate Memory" ) ;
    return ( ptr ) ;
}
```

Question *16.4*

How will you dynamically allocate a 1-D array of integers?

Answer

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int main( )
{
    int *p, i ;
    p = ( int * ) malloc ( MAX * sizeof ( int ) ) ;
    for ( i = 0 ; i < MAX ; i++ )
    {
        p[i] = i ;
        printf ( "%d\n", p[i] ) ;
    }
    return 0 ;
}
```

Question *16.5*

How will you dynamically allocate a 2-D array of integers?

Answer

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 3
#define MAXCOL 4
```



```
int main( )
{
    int *p, i, j ;
    p = ( int * ) malloc ( MAXROW * MAXCOL * sizeof ( int ) ) ;
    for ( i = 0 ; i < MAXROW ; i++ )
    {
        for ( j = 0 ; j < MAXCOL ; j++ )
        {
            p[ i * MAXCOL + j ] = i ;
            printf ( "%d ", p[ i * MAXCOL + j ] ) ;
        }
        printf ( "\n" ) ;
    }
    return 0 ;
}
```

Question *16.6*

How will you dynamically allocate a 2-D array of integers such that we are able to access any element using 2 subscripts, as in *arr[i][j]*?

Answer

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main( )
{
    int **p, i, j ;
    p = ( int ** ) malloc ( MAXROW * sizeof ( int * ) ) ;
    for ( i = 0 ; i < MAXROW ; i++ )
        p[i] = ( int * ) malloc ( MAXCOL * sizeof ( int ) ) ;
    for ( i = 0 ; i < MAXROW ; i++ )
    {
        for ( j = 0 ; j < MAXCOL ; j++ )
```

```
        {
            p[i][j] = i ;
            printf ( "%d ", p[i][j] ) ;
        }
        printf ( "\n" ) ;
    }
    return 0 ;
}
```

Question *16.7*

How will you dynamically allocate a 2-D array of integers such that we are able to access any element using 2 subscripts, as in *arr[i][j]*? Also, the rows of the array should be stored in adjacent memory locations.

Answer

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main( )
{
    int **p, i, j ;
    p = ( int ** ) malloc ( MAXROW * sizeof ( int * ) ) ;
    p[0] = ( int * ) malloc ( MAXROW * MAXCOL * sizeof ( int ) ) ;
    for ( i = 0 ; i < MAXROW ; i++ )
        p[i] = p[0] + i * MAXCOL ;
    for ( i = 0 ; i < MAXROW ; i++ )
    {
        for ( j = 0 ; j < MAXCOL ; j++ )
        {
            p[i][j] = i ;
            printf ( "%d ", p[i][j] ) ;
        }
        printf ( "\n" ) ;
```



```
    }
    return 0 ;
}
```

Question *16.8*

How many bytes will be allocated by the following code?

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main( )
{
    int ( *p )[MAXCOL] ;
    p = ( int ( * ) [MAXCOL] ) malloc ( MAXROW * sizeof ( *p ) ) ;
    return 0 ;
}
```

Answer

24 bytes if integer is 2 bytes wide and 48 bytes if integer is 4 bytes wide.

Question *16.9*

Which of the following is the correct output for the code snippet given below?

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main( )
{
    int ( *p )[MAXCOL] ;
```

```
    p = ( int ( * ) [MAXCOL] ) malloc ( MAXROW * sizeof ( *p ) ) ;
    printf ( "%d %d\n", sizeof ( p ), sizeof ( *p ) ) ;
    return 0 ;
}
```

Answer

2 8 if integer is 2 bytes wide
4 16 if integer is 4 bytes wide

Question *16.10*

In the following code *p* is a pointer to an array of MAXCOL elements. Also, *malloc( )* allocates memory for MAXROW such arrays. Will these arrays be stored in adjacent locations? How will you access the elements of these arrays using *p*?

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main( )
{
    int i, j ;
    int ( *p )[MAXCOL] ;
    p = ( int ( * )[MAXCOL] ) malloc ( MAXROW * sizeof ( *p ) ) ;
    return 0 ;
}
```

Answer

The arrays are stored in adjacent locations. You can confirm this by printing their addresses using the following loop.

```
int i ;
```



```
for ( i = 0 ; i < MAXROW ; i++ )
    printf ( "%d ", p[i] ) ;
printf ( "\n" ) ;
```

To access the array elements we can use the following set of loops.

```
for ( i = 0 ; i < MAXROW ; i++ )
{
    for ( j = 0 ; j < MAXCOL ; j++ )
        printf ( "%d ", p[i][j] ) ;
}
printf ( "\n" ) ;
```

Question *16.11*

How will you dynamically allocate a 3-D array of integers?

Answer

```
#include <stdio.h>
#include <stdlib.h>
#define MAXX 3
#define MAXY 4
#define MAXZ 5
int main( )
{
    int ***p, i, j, k ;
    p = ( int *** ) malloc ( MAXX * sizeof ( int ** ) ) ;
    for ( i = 0 ; i < MAXX ; i++ )
    {
        p[i] = ( int ** ) malloc ( MAXY * sizeof ( int * ) ) ;
        for ( j = 0 ; j < MAXY ; j++ )
            p[i][j] = ( int * ) malloc ( MAXZ * sizeof ( int ) ) ;
    }
    for ( k = 0 ; k < MAXZ ; k++ )
    {
        for ( i = 0 ; i < MAXX ; i++ )
```

```
        {
            for ( j = 0 ; j < MAXY ; j++ )
            {
                p[i][j][k] = i + j + k ;
                printf ( "%d ", p[i][j][k] ) ;
            }
            printf ( "\n" ) ;
        }
        printf ( "\n\n" ) ;
    }
    printf ( "\n" ) ;
    return 0 ;
}
```

Question *16.12*

How many bytes of memory will the following code reserve?

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    int *p ;
    p = ( int * ) malloc ( 256 * 256 ) ;
    if ( p == NULL )
        printf ( "Allocation failed" ) ;
    return 0 ;
}
```

Answer

In TC/TC++ it will fail to allocate any memory because 256 * 256 is 65536 which when passed to *malloc( )* will become a negative number since the formal argument of *malloc( )*, an *unsigned int*, can accommodate numbers only upto 65535.

In Visual Studio/gcc it will not report any error since here an unsigned integer is 4 bytes wide and can accommodate a number 65536.

Question *16.13*

How will you free the memory allocated by the following program?

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main( )
{
    int **p, i ;
    p = ( int ** ) malloc ( MAXROW * sizeof ( int * ) ) ;
    for ( i = 0 ; i < MAXROW ; i++ )
        p[i] = ( int * ) malloc ( MAXCOL * sizeof ( int ) ) ;
    return 0 ;
}
```

Answer

```
for ( i = 0 ; i < MAXROW ; i++ )
    free ( p[i] ) ;
free ( p ) ;
```

Question *16.14*

How will you free the memory allocated by the following program?

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define MAXROW 3
#define MAXCOL 4
int main( )
{
    int **p, i, j ;
    p = ( int ** ) malloc ( MAXROW * sizeof ( int * ) ) ;
    p[0] = ( int * ) malloc ( MAXROW * MAXCOL * sizeof ( int ) ) ;
    for ( i = 0 ; i < MAXROW ; i++ )
        p[i] = p[0] + i * MAXCOL ;
    return 0 ;
}
```

Answer

```
free ( p[0] ) ;
free ( p ) ;
```

Question *16.15*

Will the following code work at all times?

```
#include <stdio.h>
int main( )
{
    char *ptr ;
    gets ( ptr ) ;
    printf ( "%s\n", ptr ) ;
    return 0 ;
}
```

Answer

No. Since *ptr* is an uninitialised pointer it must be pointing at some unknown location in memory. The string that we type in will get

stored at the location to which *ptr* is pointing, thereby overwriting whatever is present at that location.

Question *16.16*

The following code is improper though it may work some times. How will you improve it?

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char *s1 = "Cyber" ;
    char *s2 = "Punk" ;
    strcat ( s1, s2 ) ;
    printf ( "%s\n", s1 ) ;
    return 0 ;
}
```

Answer

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char s1[25] = "Cyber" ;
    char *s2 = "Punk" ;
    strcat ( s1, s2 ) ;
    printf ( "%s\n", s1 ) ;
    return 0 ;
}
```

Question *16.17*

What will be the output of the second *printf( )* in the following program?

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    int *p ;
    p = ( int * ) malloc ( 20 ) ;
    printf ( "%u", p ) ; /* suppose this prints 1314 */
    free ( p ) ;
    printf ( "%u\n", p ) ;
    return 0 ;
}
```

Answer

1314

Question *16.18*

Something is logically wrong with this program. Can you point it out?

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    struct ex
    {
        int i ;
        float j ;
        char *s ;
    } ;
    struct ex *p ;

    p = ( struct ex * ) malloc ( sizeof ( struct ex ) ) ;
    p -> s = ( char * ) malloc ( 20 ) ;
    free ( p ) ;
```



```
    return 0 ;
}
```

Answer

The memory chunk at which *s* is pointing has not been freed. Just freeing the memory pointed to by the *struct* pointer *p* is not enough. Ideally we should use:

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

In short, when we allocate structures containing pointers to other dynamically allocated objects, before freeing the structure we have to first free each pointer in the structure.

Question *16.19*

To *free( )* we only pass the pointer to the block of memory that we want to deallocate. Then how does *free( )* know how many bytes it should deallocate?

Answer

In most implementations of *malloc( )* the number of bytes allocated is stored adjacent to the allocated block. Hence, it is simple for *free( )* to know how many bytes to deallocate.

Question *16.20*

What will be the output of the following program?

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

```
    int *p ;
    p = ( int * ) malloc ( 20 ) ;
    printf ( "%d\n", sizeof ( p ) ) ;
    free ( p ) ;
    return 0 ;
}
```

Answer

2 in TC/TC++
4 in Visual Studio/gcc

Question *16.21*

Can I increase the size of a statically allocated array? [Yes/No] If yes, how?

Answer

No

Question *16.22*

Can I increase the size of a dynamically allocated array? [Yes/No] If yes, how?

Answer

Yes, using the *realloc( )* function as shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main( )
{
    char *p ;
```



```
    p = ( char * ) malloc ( 20 ) ;
    strcpy ( p, "Go Embedded!" ) ;
    printf ( "String = %s Address = %p\n", p, p ) ;
    p = ( char * ) realloc ( p, 60 ) ;
    strcpy ( p, "Go Embedded! Because that's where future is" ) ;
    printf ( "String = %s Address = %p\n", p, p ) ;
    return 0 ;
}
```

Question *16.23*

Suppose we use *realloc( )* to increase the allocated space for a 20-integer array to a 40-integer array. Will it increase the array space at the same location at which the array is present or will it try to find a different place for the bigger array?

Answer

Both. If the first strategy fails then it adopts the second. If the first is successful it returns the same pointer that you passed to it otherwise it returns a different pointer for the newly allocated space.

Question *16.24*

When reallocating memory if any other pointers point into the same piece of memory do we have to readjust these other pointers or do they get readjusted automatically?

Answer

If *realloc( )* expands allocated memory at the same place then there is no need of readjustment of other pointers. However, if it allocates a new region somewhere else the programmer has to readjust the other pointers.

Question 16.25

What's the difference between *malloc( )* and *calloc( )* functions?

Answer

As against *malloc( )*, *calloc( )* needs two arguments, the number of elements to be allocated and the size of each element. For example,

```
p = ( int * ) calloc ( 10, sizeof ( int ) ) ;
```

will allocate space for a 10-integer array. Additionally, *calloc( )* will also set each of this element with a value 0. Thus the above call to *calloc( )* is equivalent to:

```
p = ( int * ) malloc ( 10 * sizeof ( int ) ) ;
memset ( p, 0, 10 * sizeof ( int ) ) ;
```

Question 16.26

Which function should be used to free the memory allocated by *calloc( )*?

Answer

The same that we use with *malloc( )*, i.e. *free( )*.

Question 16.27

malloc( ) allocates memory from the heap and not from the stack. [True/False]

Answer

True

Question 16.28

How much maximum memory can we allocate in a single call to *malloc( )*?

Answer

The largest possible block that can be allocated using *malloc( )* depends upon the host system—particularly the size of physical memory and the OS implementation.

Theoretically the largest number of bytes that can allocated should be the maximum value that can be held in *size_t* which is implementation dependent. For TC/TC++ compilers the maximum number of bytes that can be allocated is equal to 64 KB.

Question 16.29

What if we are to allocate 100 KB of memory?

Answer

Use the standard library functions *farmalloc( )* and *farfree( )*. These functions are specific to TC/TC++ under DOS.

Question 16.30

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

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main( )
{
    char *ptr ;
    *ptr = ( char * ) malloc ( 30 ) ;
    strcpy ( ptr, "RAM - Rarely Adequate Memory" ) ;
```

```
    printf ( "%s\n", ptr ) ;
    free ( ptr ) ;
    return 0 ;
}
```

Answer

While assigning the address returned by *malloc( )* we should use *ptr* and not **ptr*.

Question *16.31*

What is difference between Dynamic memory allocation and Static memory allocation?

Answer

In Static memory allocation during compilation arrangements are made to facilitate memory allocation memory during execution. Actual allocation is done only at execution time. In Dynamic memory allocation no arrangement is done at compilation time. Memory allocation is done at execution time.

Question *16.32*

Which header file should be included to dynamically allocate memory using functions like *malloc( )* and *calloc( )*?

Answer

<stdlib.h>

Question *16.33*

Specify any two Library functions with an example of each to dynamically allocate memory?

Answer

The two library functions are *malloc( )* and *calloc( )*. Examples are given below.

(a)
```
/* example of malloc( ) */
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    int *a ;
    a = ( int * ) malloc ( 2 ) ;
    *a = 10 ;
    printf ( "%d\n", *a ) ;
    return 0 ;
}
```

(b)
```
/* example of calloc( ) */
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    float *a ;
    a = ( float * ) calloc ( 1, sizeof ( float ) ) ;
    *a = 10.10 ;
    printf ( "%f\n", *a ) ;
    free ( a ) ;
    return 0 ;
}
```

Question *16.34*

When we dynamically allocate memory is there any way to free memory during run time?

Answer

Yes. Memory can be freed using *free( )* function.

Question 16.35

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

```
#include <stdlib.h>
int main( )
{
    int *a[3] ;
    a = ( int * ) malloc ( sizeof ( int ) * 3 ) ;
    free ( a ) ;
    return 0 ;
}
```

Answer

We cannot store the address of allocated memory in *a*. We should store the address in *a[i]*.

Question 16.36

Which is the proper way to allocate memory for array of pointers?

Answer

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    int *a[ 3 ] ;
    int i, j ;
    for ( i = 0; i < 3 ; i++ )
        a[ i ] = ( int *) malloc ( sizeof ( int ) * 4 ) ;
```



```
    for ( i = 0; i < 3 ; i++ )
    {
        for ( j = 0; j < 4 ; j++ )
            a[ i ][ j ] = j ;
    }
    for ( i = 0; i < 3; i++ )
    {
        for ( j = 0; j < 4 ; j++ )
            printf ( "%d ", a [ i ][ j ] ) ;
        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;

    for ( i = 0 ; i < 3 ; i++ )
        free ( a[ i ] ) ;
    return 0 ;
}
```

Question 16.37

What is the purpose of *farmalloc( )* and *farfree( )*?

Answer

To allocate a block larger than 64K, we use *farcalloc( )* function. *farmalloc( )* allocates a block of memory from the far heap . Using this function blocks larger than 64K can also be allocated. Far pointers are used to access the allocated blocks.

farfree( ) releases a block of memory previously allocated from the far heap.

Both these functions are specific to TC/TC++ under DOS.

Question 16.38

What will be the output of the following program?

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    union test
    {
        int i ;
        float f ;
        char c ;
    } ;
    union test *t ;
    t = ( union test * ) malloc ( sizeof ( union test ) ) ;
    t -> i = 10 ;
    printf ( "%d\n", t->c ) ;
    t -> f = 10.10f ;
    printf ( "%f\n", t->f ) ;
    t -> c = 'a' ;
    printf ( "%c\n", t -> i ) ;
    return 0 ;
}
```

Answer

```
10
10.100000
a
```

Question *16.39*

How will you access members of union using the variable *po* in the following program?

```
#include <stdio.h>
union office
{
    int id ;
    char name[10] ;
    float sal ;
}
int main( )
{
    union office *o, **po ;
    o = ( union office * ) malloc ( sizeof ( union office ) ) ;
    po = &o ;
    return 0 ;
}
```


Answer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
union office
{
    int id ;
    char name[10] ;
    float sal ;
} ;
int main( )
{
    union office *o, **po ;
    o = ( union office * ) malloc ( sizeof ( union office ) ) ;
    po = &o ;
    (*po) -> id = 1 ;
    printf ( "%d\n", ( *po ) -> id ) ;
    strcpy ( ( *po ) -> name, "Namita" ) ;
    printf ( "%s\n", ( *po ) -> name ) ;
    ( *po ) -> sal = 15000 ;
    printf ( "%f\n",( *po ) -> sal ) ;
    return 0 ;
}
```

Question *16.40*

How will you dynamically allocate a 1D array of structures?

Answer

```
#include <stdio.h>
#include <stdlib.h>
struct employee
{
    char name[ 15 ] ;
    int age ;
} ;
int main( )
{
    struct employee *p ;
    p = ( struct employee * ) malloc ( sizeof ( struct employee ) * 5 ) ;
    return 0 ;
}
```

Question *16.41*

Which of the following statement correctly allocates memory dynamically for a 2D array in the program given below?

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    int *p, i, j ;
    /* add statement here */
    for ( i = 0 ; i < 3 ; i++ )
    {
        for ( j = 0 ; j < 4 ; j++ )
        {
            p[ i * 4 + j ] = i ;
```



```
            printf ( "%d", p[ i * 4 + j ] ) ;
        }
    }
    printf ( "\n" ) ;
    return 0 ;
}
```

A. `p = ( int * ) malloc ( 3 * 4 * sizeof ( int ) ) ;`
B. `p = ( int * ) malloc ( 3 * sizeof ( int ) ) ;`
C. `p = malloc ( 3 * 4 * sizeof ( int ) ) ;`
D. `p = ( int * ) malloc ( 3 * 4 ) ;`

Answer

A

Question *16.42*

Which of the following options will let you access the elements of the array using *p* in the program given below?

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    int i, j ;
    int ( *p )[3] ;
    p = ( int ( * )[3] ) malloc ( 3 * sizeof ( *p ) ) ;
    return 0 ;
}
```

A.
```
for ( i = 0 ; i < 3 ; i++ )
{
    for ( j = 0 ; j < 3 ; j++ )
        printf ( "%d\n", p[ i + j ] ) ;
```

```
    }
```

B.
```
for ( i = 0 ; i < 3 ; i++ )
    printf ( "%d\n", p[ i ] ) ;
```

C.
```
for ( i = 0 ; i < 3 ; i++ )
{
    for ( j = 0 ; j < 3 ; j++ )
        printf ( "%d\n", p[ i ][ j ] ) ;
}
```

D.
```
for ( j = 0 ; j < 3 ; j++ )
    printf ( "%d\n", p[ i ][ j ] ) ;
```

Answer

C

Question *16.43*

Which of the following options correctly frees the memory pointed to by *s* and *p* in the program given below?

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
    struct ex
    {
        int i ;
        float j ;
        char *s ;
    } ;
    struct ex *p ;
    p = ( struct ex * ) malloc ( sizeof ( struct ex ) ) ;
    p -> s = ( char * ) malloc ( 20 ) ;
    return 0 ;
}
```

A.
```
free ( p -> s ) ;
free ( p ) ;
```
B.
```
free ( p ) ;
free ( p -> s ) ;
```
C.
```
free ( p -> s ) ;
```
D.
```
free ( p ) ;
```


Answer

A

Question *16.44*

Which of the following is the correct prototype of the *malloc( )* function in C?

A. `int * malloc ( int ) ;`
B. `char * malloc ( char ) ;`
C. `unsigned int * malloc ( unsigned int ) ;`
D. `void * malloc ( size_t ) ;`

Answer

D

Question *16.45*

State True or False:

A. If *malloc( )* successfully allocates memory then it returns the number of bytes it has allocated.

B. *malloc( )* returns a *float* pointer if memory is allocated for storing floats and a *double pointer* if memory is allocated for storing doubles.

C. *malloc( )* returns a NULL if it fails to allocate requested memory.

Answer

A. False
B. False
C. True

Question *16.46*

Is it necessary to cast the address returned by *malloc( )*?

Answer

It is necessary to do the typecasting if you are using TC / TC++ / Visual Studio compilers. If you are using gcc there is no need to typecast the returned address. Note that ANSI C defines an implicit type conversion between *void* pointer types (the one returned by *malloc( )*) and other pointer types.