Contents

Introduction

There is no dearth of good C programming books in the market. However, I found that there is not much material which could help a C programmer to test his programming strengths, help improve his confidence and in the process hone his C skills. Hence this book.

This is not a text book on C. In fact it is far from it.

It contains a lot of questions segregated topic-wise according to my perception of the language. Almost all the questions are real one's asked by real people attempting to learn or program in C.

There is no reason why you should read the questions in the same order as they appear in this book. You can pick up any topic that you think you are good at (or poor at) and try to test your skills on that topic.

There is a good chance that if you are learning or using C and you have questions about C that aren't answered in any of the other books you've checked, you would find them answered here. It would be too much to expect that you would find in this book answer to every question you would have when you're programming in C. This is because many of the questions that may come up in your programming would have to do with your problem domain, whereas this book concentrates only on the C language. Also it doesn't cover every aspect of every operating system under which C is running. Problems specific to an operating systems, and general-purpose algorithms are properly discussed in books devoted to those topics.

At the end of each chapter you would find correct answers to the questions in that chapter. You would find some answers more elaborate than others. At first sight this may seem unnecessary. However, I have done this to give you the complete picture rather than oversimplifying or leaving out important details.

I have tried to avoid the questions whose answers are most obvious because the idea was not to increase the number of questions, but to present questions which would force the readers to think twice before answering. That's in tune with the spirit of C - be precise, brevity has its own importance.

So roll your sleeves and get on with the real questions. Good luck!!

Yashavant P. Kanetkar
Nov., 1996

Chapter 1

Declarations and Initializations

Q 1.1

What would be the output of the following program?

```
main( )
{
    char far *s1, *s2 ;
    printf ( "%d %d", sizeof ( s1 ), sizeof ( s2 ) ) ;
}
```

Q 1.2

What would be the output of the following program?

```
int x = 40 ;
main( )
{
    int x = 20 ;
    printf ( "\n%d", x ) ;
}
```

Q 1.3

What would be the output of the following program?

```
main( )
{
    int x = 40 ;
    {
        int x  = 20 ;
        printf ( "\n%d ", x ) ;
    }
    printf ( "%d", x ) ;
}
```

Q 1.4

Is the following statement a declaration or a definition?

```
extern int i ;
```

Q 1.5

What would be the output of the following program?

```
main( )
{
    extern int i ;
    i = 20 ;
    printf ( "%d", sizeof ( i ) ) ;
}
```

A. 2
B. 4
C. Would vary from compiler to compiler
D. Error, *i* undefined

Q 1.6

Is it true that a global variable may have several declarations, but only one definition? <Yes/No>

Q 1.7

Is it true that a function may have several declarations, but only one definition? <Yes/No>

Q 1.8

In the following program where is the variable *a* getting defined and where is it getting declared?

```
main( )
{
    extern int a ;
    printf ( "%d", a ) ;
}
int a = 20 ;
```

Q 1.9

What would be the output of the following program?

```
main( )
{
    extern int a ;
    printf ( "\n%d", a ) ;
}
int a = 20 ;
```

A. 20
B. 0
C. Garbage value
D. Error

Q 1.10

What's the difference between a defintion and declaration of a variable?

Q 1.11

If the definition of an external variable occurs in the source file before its use in a particular function, then there is no need for an *extern* declaration in the function. <True/False>

Q 1.12

Suppose a program is divided into three files *f1*, *f2* and *f3*, and a variable is defined in the file *f1* but used in the files *f2* and *f3*. In such a case would we need the external declaration for the variables in the files *f2* and *f3*? <Yes/No>

Q 1.13

When we mention the prototype of a function are we defining the function or declaring it?

Q 1.14

What's the difference between the following declarations?


```
extern int fun( ) ;
int fun( ) ;
```

Q 1.15

Why does the following program report a redeclaration error of function *display()*?

```
main( )
{
    display( ) ;
}
void display( )
{
    printf ( "\nCliffhanger" ) ;
}
```

Q 1.16

What would be the output of the following program?

```
main( )
{
    extern int fun ( float ) ;
    int a ;
    a = fun ( 3.14 ) ;
    printf ( "%d", a ) ;
}

int fun ( aa )
float aa ;
{
    return ( ( int ) aa ) ;
}
```

A. 3
B. 3.14
C. 0
D. Error

Q 1.17

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

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

/* some more code may go here */

fun ( int aa )
{
    int bb ;
    bb = aa * aa ;
    return ( bb ) ;
}

main( )
{
    int a ;
    a = fun ( 20 ) ;
    printf ( "\n%d", a ) ;
}
```


Q 1.18

If you are to share the variables or functions across several source files how would you ensure that all definitions and declarations are consistent?

Q 1.19

How would you rectify the error in the following program?

```
f ( struct emp ) ;
/* any other prototypes may go here */
struct emp
{
    char name[20] ;
    int age ;
} ;
main( )
{
    struct emp e = { "Soicher", 34 } ;
    f ( e ) ;
}

f ( struct emp ee )
{
    printf ( "\n%s %d", ee.name, ee.age ) ;
}
```

Q 1.20

Global variables are available to all functions. Does there exist a mechanism by way of which I can make it available to some and not to others.

Q 1.21

What do you mean by a translation unit?

Q 1.22

What would be the output of the following program?

```
main( )
{
    int a[5] = { 2, 3 } ;
    printf ( "\n%d %d %d", a[2], a[3], a[4] ) ;
}
```

A. Garbage values
B. 2 3 3
C. 3 2 2
D. 0 0 0

Q 1.23

What would be the output of the following program?

```
main( )
{
    struct emp
    {
        char name[20] ;
        int age ;
        float sal ;
    } ;
    struct emp e = { "Tiger" } ;
    printf ( "\n%d %f", e.age, e.sal ) ;
}
```


A. 0 0.000000
B. Garbage values
C. Error
D. None of the above

Q 1.24

Some books suggest that the following definitions should be preceded by the word *static*. Is it correct?

```
int a[ ] = { 2, 3, 4, 12, 32 } ;
struct emp e = { "sandy", 23 } ;
```

Q 1.25

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

```
main( )
{
    int ( *p )( ) = fun ;
    ( *p )( ) ;
}
fun( )
{
    printf ( "\nLoud and clear" ) ;
}
```

Q 1.26

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

```
main( )
{
```

```
    union a
    {
        int i ;
        char ch[2] ;
    } ;
    union a z = 512 ;
    printf ( "%d %d", z.ch[0], z.ch[1] ) ;
}
```

Q *1.27*

What do you mean by scope of a variable? What are the 4 different types of scopes that a variable can have?

Q *1.28*

What are the different types of linkages?

Answers

A *1.1*

4 2

A *1.2*

20. Whenever there is a conflict between a local variable and a global variable it is the local variable which gets a priority.

A *1.3*

20 40. In case of a conflict between local variables, the one which is more local that gets the priority.

A *1.4*

Declaration

A *1.5*

D. *extern int i* is a declaration and not a definition, hence the error.

A *1.6*

Yes

A *1.7*

Yes

A *1.8*

extern int a is the declaration whereas *int a = 20* is the definition.

A *1.9*

A

A 1.10

In the defintion of a variable space is reserved for the variable and some initial value is given to it, whereas a declaration only identifies the type of the variable for a function. Thus definition is the place where the variable is created or assigned storage whereas declaration refers to places where the nature of the variable is stated but no storage is allocated.

A 1.11

True

A 1.12

Yes

A 1.13

We are declaring it. When the function alongwith the statements belonging to it are mentioned we are defining the function.

A 1.14

There is no difference except for the fact that the first one gives a hint that the function *fun()* is probably in another source file.

A 1.15

Here *display()* is called before it is defined. In such cases the compiler assumes that the function *display()* is declared as


```
int display( ) ;
```

That is, an undeclared function is assumed to return an *int* and accept an unspecified number of arguments. Then when we define the function the compiler finds that it is returning *void* hence the compiler reports the discrepancy.

A 1.16

D. The error occurs because we have mixed the ANSI prototype with K & R style of function definition.

When we use ANSI protptype for a function and pass a *float* to the function it is promoted to a *double*. When the function accepts this *double* into a *float* a type mismatch occurs hence the error.

The remedy for this error could be to define the function as:

```
int fun ( float aa )
{
    ..
}
```

A 1.17

Because of the missing semicolon at the end of the structure declaration (the intervening comment further obscures it) the compiler believes that *fun()* would return something of the the type *struct emp*, whereas in actuality it is attempting to return an *int*. This causes a mismatch, hence an error results.

A 1.18

The best arrangement is to place each definition in a relevant .c file. Then, put an external declaration in a header file (.h file) and use *#include* to bring in the declaration wherever needed.

The .c file which contains the definition should also include the header file, so that the compiler can check that the definition matches the declaration.

A 1.19

Declare the structure before the prototype of *f()*.

A 1.20

No. The only way this can be achieved is to define the variable locally in *main()* instead of defining it globally and then passing it to the functions which need it.

A 1.21

A translation unit is a set of source files seen by the compiler and translated as a unit: generally one .c file, plus all header files mentioned in *#include* directives.

A 1.22

D. When an automatic array is partially initialised, the remaining array elements are initialised to 0.

A 1.23

A. When an automatic structure is partially initialised, the remaining elements of the structure are initialised to 0.

A 1.24

Pre-ANSI C compilers had such a requirement. Compilers which conform to ANSI C standard do not have such a requirement.

A 1.25

Here we are initialising the function pointer *p* to the address of the function *fun()*. But during this initialisation the function has not been defined. Hence an error.

To eliminate this error add the prototype of the *fun()* before declaration of *p*, as shown below:

```
extern int fun( ) ;
```

or simply

```
int fun( ) ;
```

A 1.26

In a pre-ANSI compiler a *union* variable cannot be initialised. However, ANSI C permits initialisation of first memeber of the *union*.

A 1.27

Scope indicates the region over which the variable's declaration has an effect. The four kinds of scopes are: file, function, block and prototype.

A 1.28

There are three different types of linkages: external, internal and none. External linkage means global, non-static variables and functions, internal linkage means static variables and functions with file scope, and no linkage means local variables.

Chapter 2

Control Instructions

Q 2.1

What would be the output of the following program?

```
main( )
{
    int i = 4 ;
    switch ( i )
    {
        default :
            printf ( "\nA mouse is an elephant built by the Japanese" ) ;
        case 1 :
            printf ( "\nBreeding rabbits is a hare raising experience" ) ;
            break ;
        case 2 :
            printf ( "\nFriction is a drag" ) ;
            break ;
        case 3 :
            printf ( "\nIf practice makes perfect, then nobody's perfect" ) ;
    }
}
```

Q 2.2

Point out the error, if any, in the *for* loop.

```
main( )
{
    int i = 1 ;
    for ( ;; )
    {
        printf ( "%d", i++ ) ;
        if ( i > 10 )
            break ;
    }
}
```

A. The condition in the *for* loop is a must.
B. The two semicolons should be dropped.
C. The *for* loop should be replaced by a *while* loop.
D. No error.

Q 2.3

Point out the error, if any, in the *while* loop.

```
main( )
{
    int i = 1 ;
    while ( )
    {
        printf ( "%d", i++ ) ;
        if ( i > 10 )
            break ;
    }
}
```

A. The condition in the *while* loop is a must.
B. There should be at least a semicolon in the *while( )*.
C. The *while* loop should be replaced by a *for* loop.
D. No error.

Q 2.4

Point out the error, if any, in the *while* loop.

```
main( )
{
    int i = 1 ;
    while ( i <= 5 )
    {
        printf ( "%d", i ) ;
        if ( i > 2 )
            goto here ;
    }
}
fun( )
{
    here:
        printf ( "\nIf it works, Don't fix it." ) ;
}
```

Q 2.5

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

```
main( )
{
    int i = 4, j = 2 ;
    switch ( i )
    {
        case 1 :
            printf ( "\nTo err is human, to forgive is against company policy." ) ;
            break ;
        case j :
            printf ( "\nIf you have nothing to do, don't do it here." ) ;
            break ;
```

```
    }
}
```

Q 2.6

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

```
main( )
{
    int i = 1 ;
    switch ( i )
    {
        case 1 :
            printf ( "\nRadioactive cats have 18 half-lives." ) ;
            break ;
        case 1 * 2 + 4 :
            printf ( "\nBottle for rent - inquire within." ) ;
            break ;
    }
}
```

Q 2.7

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

```
main( )
{
    int a = 10 ;
    switch ( a )
    {
    }
    printf ( "Programmers never die. They just get lost in the processing" ) ;
}
```


Q 2.8

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

```
main( )
{
    int i = 1 ;
    switch ( i )
    {
        printf ( "Hello" ) ; /* common for both cases */
        case 1 :
            printf ( "\nIndividualists unite!" ) ;
            break ;
        case 2 :
            printf ( "\nMoney is the root of all wealth." ) ;
            break ;
    }
}
```

Q 2.9

Rewrite the following set of statements using conditional operators.

```
int a = 1, b ;
if ( a > 10 )
    b = 20 ;
```

Q 2.10

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

```
main( )
{
    int a = 10, b ;
```

```
    a >= 5 ? b = 100 : b = 200 ;
    printf ( "\n%d", b ) ;
}
```

Q *2.11*

What would be the output of the following program?

```
main( )
{
    char str[] = "Part-time musicians are semiconductors" ;
    int a = 5 ;
    printf ( a > 10 ? "%50s" : "%s", str ) ;
}
```

A. Part-time musicians are semiconductors
B. Part-time musicians are semiconductors
C. Error
D. None of the above

Q *2.12*

What is more efficient a *switch* statement or an *if-else* chain?

Q *2.13*

Can we use a *switch* statement to switch on strings?

Q *2.14*

We want to test whether a value lies in the range 2 to 4 or 5 to 7. Can we do this using a *switch*?

Q *2.15*

The way *break* is used to take the control out of *switch* can *continue* be used to take the control to the beginning of the *switch*? <Yes/No>

Answers

A *2.1*

A mouse is an elephant built by the Japanese
Breeding rabbits is a hare raising experience

A *2.2*

D

A *2.3*

A

A *2.4*

goto cannot take control to a different function.

A *2.5*

Constant expression required in the second case, we cannot use *j*.

A 2.6

No error. Constant expressions like 1 * 2 + 4 are acceptable in cases of a *switch*.

A 2.7

Though never required, there can exist a *switch* which has no cases.

A 2.8

Though there is no error, irrespective of the value of *i* the first *printf()* can never get executed. In other words, all statements in a *switch* have to belong to some case or the other.

A 2.9

```
int a= 1, b, dummy ;
a > 10 ? b = 20 : dummy = 1 ;
```

Note that the following would not have worked:

```
a > 10 ? b = 20 : ; ;
```

A 2.10

lvalue required in function *main()*. The second assignment should be written in parentheses as follows:

```
a >= 5 ? b = 100 : ( b = 200 ) ;
```


A 2.11

A

A 2.12

As far as efficiency is concerned there would hardly be any difference if at all. If the cases in a *switch* are sparsely distributed the compiler may internally use the equivalent of an *if-else* chain instead of a compact jump table. However, one should use *switch* where one can. It is definitely a cleaner way to program and certainly is not any less efficient than the *if-else* chain.

A 2.13

No. The cases in a *switch* must either have integer constants or constant expressions.

A 2.14

Yes, though in a way which would not be very practical if the ranges are bigger. The way is shown below:

```
switch ( a )
{
    case 2 :
    case 3 :
    case 4 :
        /* some statements */
        break ;
    case 5 :
    case 6 :
    case 7 :
```

```
    /* some other statements */
    break ;
}
```

A *2.15*

No. *continue* can work only with loops and not with *switch*.

Chapter 3

Expressions

Q *3.1*

What would be the output of the following program?

```
main( )
{
    static int a[20] ;
    int i = 0 ;
    a[i] = i++ ;
    printf ( "\n%d %d %d", a[0], a[1], i ) ;
}
```

Q *3.2*

What would be the output of the following program?

```
main( )
{
    int i = 3 ;
    i = i++ ;
    printf ( "%d", i ) ;
}
```

Q 3.3

The expression on the right hand side of && and || operators does not get evaluated if the left hand side determines the outcome. <True/False>

Q 3.4

What would be the output of the following program?

```
main()
{
    int i = 2 ;
    printf ( "\n%d %d", ++i, ++i ) ;
}
```

A. 3 4
B. 4 3
C. 4 4
D. Output may vary from compiler to compiler.

Q 3.5

What would be the output of the following program?

```
main()
{
    int x = 10, y = 20, z = 5, i ;
    i = x < y < z ;
    printf ( "\n%d", i ) ;
}
```

A. 1
B. 0

C. Error
D. None of the above.

Q 3.6

Are the following two statements same? <Yes/No>

```
a <= 20 ? b = 30 : c = 30 ;
( a <= 20 ) ? b : c = 30 ;
```

Q 3.7

Can you suggest any other way of writing the following expression such that 30 is used only once?

```
a <= 20 ? b = 30 : c = 30 ;
```

Q 3.8

How come that the C standard says that the expression

```
j = i++ * i++ ;
```

is undefined, whereas, the expression

```
j = i++ && i++ ;
```

is perfectly legal.

Q 3.9

If *a[i] = i++* is undefined, then by the same reason *i = i + 1* should also be undefined. But it is not so. Why?

Q 3.10

Would the expression *p++ = c be disallowed by the compiler.

Q 3.11

In the following code in which order the functions would be called?

```
a = f1 ( 23, 14 ) * f2 ( 12/4 ) + f3( ) ;
```

A. f1, f2, f3
B. f3, f2, f1
C. The order may vary from compiler to compiler
D. None of the above

Q 3.12

In the following code in which order the functions would be called?

```
a = ( f1 ( 23, 14 ) * f2 ( 12/4 ) ) + f3( ) ;
```

A. f1, f2, f3
B. f3, f2, f1
C. The order may vary from compiler to compiler
D. None of the above

Q 3.13

What would be the output of the following program?

```
main( )
{
    int i = -3, j = 2, k = 0, m ;
    m = ++i && ++j || ++k ;
    printf ( "\n%d %d %d %d", i, j, k, m ) ;
}
```


Q 3.14

What would be the output of the following program?

```
main( )
{
    int i = -3, j = 2, k = 0, m ;
    m = ++j && ++i || ++k ;
    printf ( "\n%d %d %d %d", i, j, k, m ) ;
}
```

Q 3.15

What would be the output of the following program?

```
main( )
{
    int i = -3, j = 2, k = 0, m ;
    m = ++i || ++j && ++k ;
    printf ( "\n%d %d %d %d", i, j, k, m ) ;
}
```

Q 3.16

What would be the output of the following program?

```
main( )
{
    int i = -3, j = 2, k = 0, m ;
    m = ++i && ++j && ++k ;
```

```
    printf ( "\n%d %d %d %d", i, j, k, m ) ;
}
```

Answers

A 3.1

0 0 1

That's what some of the compilers would give. But some other compiler may give a different answer. The reason is, when a single expression causes the same object to be modified or to be modified and then inspected the behaviour is undefined.

A 3.2

4. But basically the behaviour is undefined for the same reason as in 3.1 above.

A 3.3

True. For example if *a* is non-zero then *b* will not be evaluated in the expression *a* || *b*.

A 3.4

D. The order of evaluation of the arguments to a function call is unspecified.

A 3.5

A

A 3.6

No

A 3.7

```
*( ( a <= 20 ) ? &b : &c ) = 30 ;
```

A 3.8

According to the C standard an object's stored value can be modified only once (by evaluation of expression) between two sequence points. A sequence point occurs:

- at the end of full expression (expression which is not a sub-expression in a larger expression)
- at the &&, || and ?: operators
- at a function call (after the evaluation of all arguments, just before the actual call)

Since in the first expression *i* is getting modified twice between two sequence points the expression is undefined. Also, the second expression is legal because a sequence point is occurring at *&&* and *i* is getting modified once before and once after this sequence point.

A 3.9

The standard says that if an object is to get modified within an expression then all accesses to it within the same expression must be for computing the value to be stored in the object. The expression *a[i] = i++* is disallowed because one of the accesses of *i* (the one in *a[i]*) has nothing to do with the value that ends up being stored in *i*.

In this case the compiler may not know whether the access should take place before or after the incremented value is stored. Since there's no good way to define it, the standard declares it as undefined. As against this the expression $i = i + 1$ is allowed because i is accessed to determine i's final value.

A 3.10

No. Because here even though the value of p is accessed twice it is used to modify two different objects p and $*p$.

A 3.11

C. Here the multiplication will happen before the addition, but in which order the functions would be called is undefined.

A 3.12

C. Here the multiplication will happen before the addition, but in which order the functions would be called is undefined. In an arithmetic expression the parentheses tell the compiler which operands go with which operators but do not force the compiler to evaluate everything within the parentheses first.

A 3.13

-2 3 0 1

A 3.14

-2 3 0 1

A 3.15

-2 2 0 1

A 3.16

-2 3 1 1

Chapter 4

Floating Point Issues

Q 4.1

What would be the output of the following program?

```
main( )
{
    float a = 0.7 ;
    if ( a < 0.7 )
        printf ( "C" ) ;
    else
        printf ( "C++" ) ;
}
```

A. C
B. C++
C. Error
D. None of the above

Q 4.2

What would be the output of the following program?

```
main( )
{
```

```
    float a = 0.7 ;
    if ( a < 0.7f)
        printf ( "C" ) ;
    else
        printf ( "C++" ) ;
}
```

A. C
B. C++
C. Error
D. None of the above

Q 4.3

What would be the output of the following program?

```
main( )
{
    printf ( "%f", sqrt ( 36.0 ) ) ;
}
```

A. 6.0
B. 6
C. 6.000000
D. Some absurd result

Q 4.4

Would this program give proper results? <Yes/No>

```
main( )
{
    printf ( "%f", log ( 36.0 ) ) ;
}
```


Q 4.5

Would the following *printf( )*s print the same values for any value of *a*? <Yes/No>

```
main( )
{
    float a ;
    scanf ( "%f", &a ) ;
    printf ( "%f", a + a + a ) ;
    printf ( "%f", 3 * a ) ;
}
```

Q 4.6

We want to round off *x*, a *float*, to an *int* value. The correct way to do so would be

A. y = (int) (x + 0.5) ;
B. y = int (x + 0.5) ;
C. y = (int) x + 0.5 ;
D. y = (int) ((int) x + 0.5)

Q 4.7

Which error are you likely to get when you run the following program?

```
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 ) ;
}
```

A. Suspicious pointer conversion
B. Floating point formats not linked
C. Cannot use *scanf()* for structures
D. Strings cannot be nested inside structures

Q 4.8

What causes the error in problem 4.7 above to occur and how would you rectify the error in the above program?

Q 4.9

Which are the three different types of real data types available in C and what are the format specifiers used for for them?

Q 4.10

By default any real number is treated as

A. a *float*
B. a *double*
C. a *long double*
D. Depends upon the memory model that you are using

Q 4.11

What should you do to treat the constant 3.14 as a *float*?

Q 4.12

What should you do to treat the constant 3.14 as a *long double*?

Q 4.13

What would be the output of the following program?

```
main( )
{
    printf ( "%d %d %d", sizeof ( 3.14f ), sizeof ( 3.14 ), sizeof ( 3.14l ) ) ;
}
```

A. 4 4 4
B. 4 Garbage value Garbage value
C. 4 8 10
D. Error

Q 4.14

The binary equivalent of 5.375 is

A. 101.101110111
B. 101.011
C. 101011
D. None of the above

Q 4.15

How *float*s are stored in binary form?

Q 4.16

A float occupies 4 bytes. If the hexadecimal equivalent of each of these bytes is A, B, C and D, then when this float is stored in memory these bytes get stored in the order

A. ABCD
B. DCBA
C. 0xABCD
D. 0xDCBA

Q 4.17

If the binary equivalent of 5.375 in normalised form is 0100 0000 1010 1100 0000 0000 0000 0000, what would be the ouput of the following program?

```
main( )
{
    float a = 5.375 ;
    char *p ;
    int i ;
    p = ( char * ) &a ;
    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "%02x ", ( unsigned char ) p[i] ) ;
}
```

A. 40 AC 00 00
B. 04 C A 00 00
C. 0000 AC 40
D. 00 00 CA 04

Answers

A 4.1

A

A 4.2

B

A 4.3

D

A 4.4

No, since we have not included the header file "math.h".

A 4.5

No. For example, for 1.7 the two *printf()*s would print different values.

A 4.6

A

A 4.7

B

A 4.8

What causes the 'floating point formats not linked' 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 compilercan 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.

A 4.9

float	4 bytes	%f
double	8 bytes	%lf
long double	10 bytes	%Lf

A 4.10

B

A 4.11

Use 3.14f

A 4.12

Use 3.14l

A 4.13

C

A 4.14

B

A 4.15

Floating-point numbers are represented in IEEE format. The IEEE format for floating point storage uses a sign bit, a mantissa and an exponent for representing the power of 2. The sign bit denotes the sign of the number: a 0 represents a positive value and a 1 denotes a negative value. The mantissa is represented in binary after converting it to its normalised form. The normalised form results in a mantissa whose most significant digit is always 1. The IEEE format takes advantage of this by not storing this bit at all. The exponent is an integer stored in unsigned binary format after adding a positive integer bias. This ensures that the stored exponent is always positive. The value of the bias is 127 for *floats* and 1023 for *doubles*.

A *4.16*

B

A *4.17*

C

Chapter 5

Functions

Q *5.1*

What would be the output of the following program?

```
main()
{
    int a, b ;
    a = sumdig ( 123 ) ;
    b = sumdig ( 123 ) ;
    printf ( "%d %d", a, b ) ;
}

sumdig ( int n )
{
    static int s = 0 ;
    int d ;
    if ( n != 0 )
    {
        d = n % 10 ;
        n = ( n - d ) / 10 ;
        s = s + d ;
        sumdig ( n ) ;
    }
    else
        return ( s ) ;
}
```

Q 5.2

What error would the following function give on compilation?

```
f ( int a, int b )
{
    int a ;
    a = 20 ;
    return a ;
}
```

A. Missing parentheses in *return* statement
B. The function should be defined as *int f (int a, int b)*
C. Redeclaration of *a*
D. None of the above

Q 5.3

There is a mistake in the following code. Add a statement in it to remove it.

```
main( )
{
    int a ;
    a = f ( 10, 3.14 ) ;
    printf ( "%d", a ) ;
]
f ( int aa, float bb )
{
    return ( ( float ) aa + bb ) ;
}
```


Q 5.4

Point out the error in the following code.

```
main( )
{
    int a = 10 ;
    void f( ) ;
    a = f( ) ;
    printf ( "\n%d", a ) ;
}
void f( )
{
    printf ( "\nHi" ) ;
}
```

Q 5.5

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

```
main( )
{
    int b ;
    b = f ( 20 ) ;
    printf ( "%d", b ) ;
}
int f ( int a )
{
    a > 20 ? return ( 10 ) : return ( 20 ) ;
}
```

Q 5.6

A function cannot be defined inside another fuction. <True/False>

Q 5.7

Will the following functions work? <Yes/No>

```
f1 ( int a, int b )
{
    return ( f2 ( 20 ) ) ;
}
f2 ( int a )
{
    return ( a * a ) ;
}
```

Q 5.8

What are the following two notations of defining functions commonly known as:

```
inf f ( int a, float b )
{
    /* some code */
}

int f ( a, b )
int a ; float b ;
{
    /* some code */
}
```

Q 5.9

In a function two *return* statements should never occur. <True/False>

Q 5.10

In a function two *return* statements should never occur successively. <True/False>

Q 5.11

In C all functions except *main()* can be called recursively. <True/False>

Q 5.12

Usually recursion works slower than loops. <True/False>

Q 5.13

Is it true that too many recursive calls may result into stack overflow? <Yes/No>

Q 5.14

How many times the following program would print 'Jamboree'?

```
main()
{
    printf ( "\nJamboree" ) ;
    main( ) ;
}
```

A. Infinite number of times
B. 32767 times
C. 65535 times

D. Till the stack doesn't overflow

Answers

A *5.1*

6 12

A *5.2*

C

A *5.3*

Add the following function prototype in *main()*:

```
float f ( int, float ) ;
```

A *5.4*

In spite of defining the function *f()* as returning *void*, the program is trying to collect the value returned by *f()* in the variable *a*.

A *5.5*

return statement cannot be used as shown with the conditional operators. Instead the following statement may be used:

```
return ( a > 20 ? 10 : 20 ) ;
```


A *5.6*

True

A *5.7*

Yes

A *5.8*

The first one is known as ANSI notation and the second is known as Kernighan and Ritchie or simply, K & R notation.

A *5.9*

False

A *5.10*

True

A *5.11*

False. Any function including *main()* can be called recursively.

A *5.12*

True

A *5.13*

Yes

A *5.14*

D

Chapter 6

The C Preprocessor

Q *6.1*

If the file to be included doesn't exist, the preprocessor flashes an error message. <True/False>

Q *6.2*

The preprocessor can trap simple errors like missing declarations, nested comments or mismatch of braces. <True/False>

Q *6.3*

What would be the ouput of the following program?

```
#define SQR(x) ( x * x )
main( )
{
    int a, b = 3 ;
    a = SQR ( b + 2 ) ;
    printf ( "\n%d", a ) ;
}
```

A. 25
B. 11

C. Error
D. Garbage value

Q 6.4

How would you define the SQR macro in 6.3 above such that it gives the result of *a* as 25.

Q 6.5

What would be the output of the following program?

```
#define CUBE(x) ( x * x * x )
main( )
{
    int a, b = 3 ;
    a = CUBE ( b++ ) ;
    printf ( "\n%d %d", a, b ) ;
}
```

Q 6.6

Indicate what would the SWAP macro be expanded to on preprocessing. Would the code compile?

```
#define SWAP( a, b, c ) ( c t ; t = a, a = b, b = t ; )
main( )
{
    int x = 10, y = 20 ;
    SWAP ( x, y, int ) ;
    printf ( "%d %d", x, y ) ;
}
```


Q 6.7

How would you modify the SWAP macro in 6.6 above such that it is able to exchange two integers.

Q 6.8

What is the limitation of the SWAP macro of 6.7 above?

Q 6.9

In which line of the following program an error would be reported?

```
1.  #define CIRCUM(R) ( 3.14 * R * R ) ;
2.  main( )
3.  {
4.      float r = 1.0, c ;
5.      c = CIRCUM ( r ) ;
6.      printf ( "\n%f", c ) ;
7.      if ( CIRCUM ( r ) == 6.28 )
8.          printf ( "\nGobbledygook" ) ;
9.  }
```

Q 6.10

What is the type of the variable *b* in the following declaration?

```
#define FLOATPTR float *
FLOATPTR a, b ;
```

Q 6.11

Is it necessary that the header files should have a .h extension?

Q 6.12

What do the header files usually contain?

Q 6.13

Would it result into an error if a header file is included twice? <Yes/No>

Q 6.14

How can a header file ensure that it doesn't get included more than once?

Q 6.15

On inclusion, where are the header files searched for?

Q 6.16

Would the following *typedef* work?

```
typedef #include I ;
```


Q 6.17

Would the following code compile correctly?

```
main( )
{
    #ifdef NOTE
        /* unterminated comment
        int a ;
        a = 10 ;
    #else
        int a ;
        a = 20 ;
    #endif

    printf ( "%d", a ) ;
}
```

Q 6.18

What would be the output of the following program?

```
#define MESS Junk
main( )
{
    printf ( "MESS" ) ;
}
```

Q 6.19

Would the following program print the message infinite number of times? <Yes/No>

```
#define INFINITELOOP while ( 1 )
```

```
main( )
{
    INFINITELOOP
        printf ( "\nGrey haired" ) ;
}
```

Q 6.20

What would be the output of the following program?

```
#define MAX( a, b ) ( a > b ? a : b )
main( )
{
    int x ;
    x = MAX ( 3 + 2, 2 + 7 ) ;
    printf ( "%d", x ) ;
}
```

Q 6.21

What would be the output of the following program?

```
#define PRINT( int ) printf ( "%d ", int )
main( )
{
    int x = 2, y = 3, z = 4 ;
    PRINT ( x ) ;
    PRINT ( y ) ;
    PRINT ( z ) ;
}
```

Q 6.22

What would be the output of the following program?


```
#define PRINT( int ) printf ( "int = %d ", int )
main( )
{
    int x = 2, y = 3, z = 4 ;
    PRINT ( x ) ;
    PRINT ( y ) ;
    PRINT ( z ) ;
}
```

Q 6.23

How would you modify the macro of 6.22 such that it outputs:

```
x = 2  y = 3  z = 4
```

Q 6.24

Would the following program compile successfully? <Yes/No>

```
main( )
{
    printf ( "Tips" "Traps" ) ;
}
```

Q 6.25

Define the macro DEBUG such that the following program outputs:

```
DEBUG: x = 4
DEBUG: y = 3.140000
DEBUG: ch = A

main( )
{
```

```
    int x = 4 ;
    float a = 3.14 ;
    char ch = 'A' ;

    DEBUG ( x, %d ) ;
    DEBUG ( a, %f ) ;
    DEBUG ( ch, %c ) ;
}
```

Q 6.26

What would be the output of the following program?

```
#define str(x) #x
#define Xstr(x)  str(x)
#define oper  multiply
main( )
{
    char *opername = Xstr ( oper ) ;
    printf ( "%s", opername ) ;
}
```

Q 6.27

Write the macro PRINT for the following program such that it outputs:

```
x = 4 y = 4 z = 5
a = 1 b = 2 c = 3

main( )
{
    int x = 4, y = 4, z = 5 ;
    int a = 1, b = 2, c = 3 ;
    PRINT ( x, y, z ) ;
    PRINT ( a, b, c ) ;
}
```


Answers

A 6.1

True

A 6.2

False

A 6.3

B. Because, on preprocessing the expression becomes *a = (3 + 2 * 2 + 3)*.

A 6.4

```
#define SQR(x) ( (x) * (x) )
```

A 6.5

27 6. Though some compilers may give this as the answer, according to the ANSI C standard the expression *b++ * b++ * b++* is undefined. Refer Chapter 3 for more details on this.

A 6.6

```
( int t ; t = a, a = b, b = t ; ) ;
```

This code won't compile since declaration of *t* cannot occur within parentheses.

A 6.7

```
#define SWAP( a, b, c ) c t; t = a, a = b, b = t;
```

A 6.8

It cannot swap pointers. For example, the following code would not compile.

```
#define SWAP( a, b, c ) c t; t = a, a = b, b = t;
main( )
{
    float x = 10, y = 20;
    float *p, *q;
    p = &x; q = &y;
    SWAP ( p, q, float* );
    printf ( "%f %f", x, y );
}
```

A 6.9

Line number 7, whereas the culprit is really the semicolon in line number 1. On expansion line 7 becomes *if ((3.14 * 1.0 * 1.0) ; == 6.28)*. Hence the error.

A 6.10

float and not a pointer to a *float*, since on expansion the declaration becomes:

```
float *a, b;
```


A 6.11

No. However, traditionally they have been given a .h extension to identify them as something different than the .c program files.

A 6.12

Preprocessor directives like *#define*, *structure*, *union* and *enum* declarations, *typedef* declarations, global variable declarations and external function declarations. You should not write the actual code (i.e. function bodies) or global variable definition (that is defining or initialising instances) in header files. The *#include* directive should be used to pull in header files, not other .c files.

A 6.13

Yes, unless the header file has taken care to ensure that if already included it doesn't get included again.

A 6.14

All declarations must be written in the manner shown below. Assume that the name of the header file is FUNCS.H.

```
/* funcs.h */
#ifndef _FUNCS
    #define _FUNCS
    /* all declarations would go here */
#endif
```

Now if we include this file twice as shown below, it would get included only once.

```
#include "goto.c"
#include "goto.c"
main( )
{
    /* some code */
}
```

A 6.15

If included using < > the files get searched in the predefined (can be changed) include path. If included with the " " syntax in addition to the predefined include path the files also get searched in the current directory (usually the directory from which you invoked the compiler).

A 6.16

No. Because *typedef* goes to work after preprocessing.

A 6.17

No. Even though the *#ifdef* fails in this case (NOTE being undefined) and the *if* block doesn't go for compilation errors in it are not permitted.

A 6.18

MESS

A 6.19

Yes

A 6.20

9

A 6.21

2 3 4

A 6.22

int = 2 int = 3 int = 4

A 6.23

```
#define PRINT( int) printf ( #int " = %d ", int)
main( )
{
    int x = 2, y = 3, z = 4 ;
    PRINT ( x ) ;
    PRINT ( y ) ;
    PRINT ( z ) ;
}
```

The rule is if the parameter name is preceded by a # in the macro expansion, the combination (of # and parameter) will be expanded into a quoted string with the parameter replaced by the actual argument. This can be combined with string concatenation to print the output desired in our program. On expansion the macro becomes

```
printf ( "x" " = %d", x ) ;
```

The two strings get concatenated, so the effect is

```
printf ( "x = %d", x ) ;
```

A 6.24

Yes. The output would be TipsTraps. In fact this result has been used in 6.23 above.

A 6.25

```
#define DEBUG( var, fmt)  printf ( "DEBUG:" #var " = " #fmt "\n", var)
```

A 6.26

multiply

Here two operations are being carried out expansion and stringizing. *Xstr()* macro expands its argument, and then *str()* stringizes it.

A 6.27

```
#define PRINT( var1, var2, var3 )  printf ( "\n" #var1 " = %d " #var2 " = %d "
        #var3 " = %d ", var1, var2, var3 )
```

Chapter 7
Pointers

Q 7.1

Can you combine the following two statements into one?

```
char *p ;
p = malloc ( 100 ) ;
```

Q 7.2

Can you split the following statement into two statements?

```
char far *scr = ( char far * ) 0xB8000000L ;
```

Q 7.3

Are the expressions **ptr++* and *++*ptr* same?

Q 7.4

Can you write another expression which does the same job as *++*ptr*?

Q 7.5

What would be the equivalent pointer expression for referring the same element as *a[i][j][k][l]*?

Q 7.6

What would be the output of the following program?

```
main( )
{
    intarr[ ] = { 12, 13, 14, 15, 16 } ;
    printf ( "\n%d %d %d", sizeof ( arr ), sizeof ( *arr ), sizeof ( arr[0] ) ) ;
}
```

Q 7.7

What would be the output of the following program assuming that the array begins at location 1002?

```
main( )
{
    int a[3][4] = {
                        1, 2, 3, 4,
                        5, 6, 7, 8,
                        9, 10, 11, 12
                  } ;

    printf ( "\n%u %u %u", a[0] + 1, * ( a[0] + 1 ), * ( * ( a + 0 ) + 1 ) ) ;
}
```


Q 7.8

What would be the output of the following program assuming that the array begins at location 1002?

```
main( )
{
    int a[2][3][4] = {
                        {
                            1, 2, 3, 4,
                            5, 6, 7, 8,
                            9, 1, 1, 2
                        },
                        {
                            2, 1, 4, 7,
                            6, 7, 8, 9,
                            0,0, 0, 0
                        }
                    } ;

    printf ( "\n%u %u %u %d", a, *a, **a, ***a ) ;
}
```

Q 7.9

In the following program how would you print 50 using *p*?

```
main( )
{
    int a[ ] = { 10, 20, 30, 40, 50 } ;
    char *p ;
    p = ( char * ) a ;
}
```

Q 7.10

Where can one think of using pointers?

Q 7.11

In the following program add a statement in the function *fun()* such that address of *a* gets stored in *j*.

```
main( )
{
    int *j ;
    void fun ( int ** ) ;

    fun ( &j ) ;
}
void fun ( int **k )
{
    int a = 10 ;
    /* add statement here */
}
```

Q 7.12

How would you declare an array of three function pointers where each function receives two *int*s and returns a *float*?

Q 7.13

Would the following program give a compilation error or warning? <Yes/No>

```
main( )
{
    float i = 10, *j ;
    void *k ;
    k = &i ;
    j = k ;
    printf ( "\n%f", *j ) ;
}
```


Q 7.14

Would the following program compile?

```
main( )
{
    int a = 10, *j ;
    void *k ;
    j = k = &a;
    j++ ;
    k++ ;
    printf ( "\n%u %u", j, k ) ;
}
```

Q 7.15

Would the following code compile successfully?

```
main( )
{
    printf ( "%c", 7[ "Sundaram" ] ) ;
}
```

Answers

A 7.1

```
char *p = malloc ( 100 ) ;
```

A 7.2

```
char far *scr ;
scr = ( char far * ) 0xB8000000L ;
```

A 7.3

No. **ptr++* increments the pointer and not the value pointed by it, whereas *++*ptr* increments the value being pointed to by *ptr*.

A 7.4

```
( *ptr )++
```

A 7.5

```
*( *( *( *( a + i ) + j ) + k ) + l )
```

A 7.6

10 2 2

A 7.7

1004 2 2

A 7.8

1002 1002 1002 1

A 7.9

```
printf ( "\n%d", * ( ( int * ) p + 4 ) ) ;
```

A 7.10

At lot of places, some of which are:

- Accessing array or string elements
- Dynamic memory allocation
- Call by reference
- Implementing linked lists, trees, graphs and many other data structures

A 7.11

```
*k = &a ;
```

A 7.12

```
float ( *arr[3] ) ( int, int ) ;
```

A 7.13

No. Here no typecasting is required while assigning the value to and from *k* because conversions are applied automatically when other pointer types are assigned to and from *void **.

A *7.14*

No. An error would be reported in the statement *k++* since arithmetic on *void* pointers is not permitted unless the void pointer is appropriately typecasted.

A *7.15*

Yes. It would print m of Sundaram.

Chapter 8

More About Pointers

Q *8.1*

Is the NULL pointer same as an uninitialised pointer? <Yes/No>

Q *8.2*

In which header file is the NULL macro defined.

Q *8.3*

Why is it that for large memory models NULL has been defined as 0L and for small memory models as just 0?

Q *8.4*

What is a null pointer?

Q *8.5*

What's the difference between a null pointer, a NULL macro, the ASCII NUL character and a null string?

Q 8.6

What would be the output of the following program?

```
#include "stdio.h"
main( )
{
    int a, b = 5 ;
    a = b + NULL ;
    printf ( "%d", a ) ;
}
```

Q 8.7

Is the programming style adopted in 8.6 good?

Q 8.8

What would be the output of the following program?

```
#include "stdio.h"
main( )
{
    printf ( "%d %d", sizeof ( NULL), sizeof ( "" ) ) ;
}
```

Q 8.9

How many bytes are occupied by *near*, *far* and *huge* pointers?

Q 8.10

What does the error "Null Pointer Assignment" mean and what causes this error?

Q 8.11

How do we debug a Null Pointer Assignment error?

Q 8.12

Can anything else generate a Null Pointer Assignment error?

Q 8.13

Are the three declarations *char **apple*, *char *orange[ ]*, and *char cherry[ ][ ]* same? <Yes/No>

Q 8.14

Can two different *near* pointers contain two different addresses but refer to the same location in memory? <Yes/No>

Q 8.15

Can two different *far* pointers contain two different addresses but refer to the same location in memory? <Yes/No>

Q 8.16

Can two different *huge* pointers contain two different addresses but refer to the same location in memory? <Yes/No>

Q 8.17

Would the following program give any warning on compilation?

```
#include "stdio.h"
main( )
{
    int *p1, i = 25 ;
    void *p2 ;
    p1 = &i ;
    p2 = &i ;
    p1 = p2 ;
    p2 = p1 ;
}
```

Q 8.18

Would the following program give any warning on compilation?

```
#include "stdio.h"
main( )
{
    float *p1, i = 25.50 ;
    char *p2 ;
    p1 = &i ;
    p2 = &i ;
}
```


Q 8.19

What warning would be generated on compiling the following program?

```
main( )
{
    char far *scr ;
    scr = 0xB8000000 ;
    *scr = 'A' ;
}
```

Q 8.20

How would you eliminate the warning generated on compiling the following program?

```
main( )
{
    char far *scr ;
    scr = 0xB8000000 ;
    *scr = 'A' ;
}
```

Q 8.21

How would you obtain a *far* address from the segment and offset addresses of a memory location?

Q 8.22

How would you obtain segment and offset addresses from a *far* address of a memory location?

Q 8.23

In a large data model (compact, large, huge) all pointers to data are 32 bits long, whereas in a small data model (tiny, small, medium) all pointers are 16 bits long. <True/False>

Q 8.24

A *near* pointer uses the contents of CS register (if the pointer is pointing to code) or contents of DS register (if the pointer is pointing to data) for the segment part, whereas the offset part is stored in the 16-bit *near* pointer <True/False>.

Q 8.25

What would be the output of the following program?

```
main( )
{
    char far *a = 0x00000120 ;
    char far *b = 0x00100020 ;
    char far *c = 0x00120000 ;

    if ( a == b )
        printf ( "\nHello" ) ;
    if ( a == c )
        printf ( "\nHi" ) ;
    if ( b == c )
        printf ( "\nHello Hi" ) ;
    if ( a > b && a > c && b > c )
        printf ( "\nBye" ) ;
}
```


Q 8.26

```
main( )
{
    char huge *a = 0x00000120 ;
    char huge *b = 0x00100020 ;
    char huge *c = 0x00120000 ;

    if ( a == b )
        printf ( "\nHello" ) ;
    if ( a == c )
        printf ( "\nHi" ) ;
    if ( b == c )
        printf ( "\nHello Hi" ) ;
    if ( a > b && a > c && b > c )
        printf ( "\nBye" ) ;
}
```

Answers

A 8.1

No

A 8.2

In files "stdio.h" and "stddef.h"

A 8.3

Because in small memory models the pointer is two bytes long whereas in large memory models it is 4 bytes long.

A 8.4

For each pointer type (like say a *char* pointer) C defines a special pointer value which is guarenteed not to point to any object or function of that type. Usually, the null pointer constant used for representing a null pointer is the integer 0.

A 8.5

A null pointer is a pointer which doesn't point anywhere.

A NULL macro is used to represent the null pointer in source. It has a value 0 associated with it.

The ASCII NUL character has all its bits as 0 but doesn't have any relationship with the null pointer.

The null string is just another name for an empty string "".

A 8.6

5

A 8.7

No.

Only in context of pointers should NULL and 0 be considered equivalent. NULL should not be used when other kind of 0 is required. Even though this may work it is a bad style of programming. ANSI C permits definition of the NULL macro as *((void *) 0)*, which ensures that the NULL will not work in non-pointer contexts.

A 8.8

2 1

A 8.9

A *near* pointer is 2 bytes long whereas a *far* and a *huge* pointer are 4 bytes long.

A 8.10

The Null Pointer Assignment error is generated only in small and medium memory models. This error occurs in programs which attempt to change the bottom of the data segment.

In Borland's C or C++ compilers, Borland places four zero bytes at the bottom of the data segment, followed by the Borland copyright notice "Borland C++ - Copyright 1991 Borland Intl.". In the small and medium memory models, a null pointer points to DS:0000. Thus assigning a value to the memory referenced by this pointer will overwrite the first zero byte in the data segment. At program termination, the four zeros and the copyright banner are checked. If either has been modified, then the Null Pointer Assignment error is generated. Note that the pointer may not truly be null, but may be a wild pointer that references these key areas in the data segment.

A 8.11

In the Integrated Development Environment set two watches on the key memory locations mentioned in 8.10. These watches, and what they should display in the watch window, are:

*(char *)4,42MS Borland C++ - Copyright 1991 Borland Intl."

```
(char *)0                00 00 00 00
```

Of course, the copyright banner will vary depending on your version of the Borland C/C++ compiler.

Step through your program using F8 or F7 and monitor these values in the watch window. At the point where one of them changes, you have just executed a statement that uses a pointer that has not been properly initialized.

The most common cause of this error is probably declaring a pointer and then using it before allocating memory for it. For example, compile the following program in the small memory model and execute it:

```
#include "dos.h"
#include "stdio.h"
#include "string.h"

main( )
{
    char *ptr, *banner ;
    banner = ( char * ) MK_FP ( _DS, 4 ) ;
    printf ( "banner: %s\n", banner) ;
    strcpy ( ptr, "The world cup saga" ) ;
    printf ( "&ptr = %Fp\n", ( void far* ) &ptr[0] ) ;
    printf ( "banner: %s\n", banner) ;
}
```

One of the best debugging techniques for catching Null pointer assignment errors is to turn on all warning compiler messages. If the above program is compiled with warnings turned off, no warning messages will be generated. However, if all warnings are turned on, both the *strcpy()* and *printf()* calls using the *ptr* variable will generate warnings. You should be particularly suspicious of any warnings that a variable might be used before being initialized, or of a suspicious pointer assignment.

Note that a Null Pointer Assignment error is not generated in all models. In the compact, large and huge memory models, *far* pointers are used for data. Therefore, a null pointer will reference 0000:0000, or the base of system memory, and using it will not cause a corruption of the key values at the base of the data segment. Modifying the base of system memory usually causes a system crash, however. Although it would be possible that a wild pointer would overwrite the key values, it would not indicate a null pointer. In the tiny memory model, DS = CS = SS. Therefore, using a null pointer will overwrite the beginning of the code segment.

A 8.12

Yes, using a wild pointer that happens to reference the base area of the data segment may cause the same error since this would change the zeros or the copyright banner. Since data corruption or stack corruption could cause an otherwise-valid pointer to be corrupted and point to the base of the data segment, any memory corruption could result in this error being generated. If the pointer used in the program statement which corrupts the key values appears to have been properly initialized, place a watch on that pointer. Step through your program again and watch for its value (address) to change.

A 8.13

No

A 8.14

No

A *8.15*

Yes

A *8.16*

No

A *8.17*

No

A *8.18*

Yes. Suspicious pointer conversion in function main

A *8.19*

Non-portable pointer assignment in function main.

A *8.20*

Use the typecast *scr = (char far *) 0xB8000000;*

A *8.21*

```
#include "dos.h"
main( )
{
    char *seg = ( char * ) 0xB000 ;
    char *off = ( char * ) 0x8000 ;
    char far *p ;
    p = MK_FP ( seg, off) ;
}
```


A *8.22*

```
#include "dos.h"
main( )
{
    char far *scr = ( char far * ) 0xB8000000 ;
    char *seg, *off ;
    seg = ( char * ) FP_SEG ( scr ) ;
    off = ( char * ) FP_OFF ( scr ) ;
}
```

A *8.23*

True

A *8.24*

True

A *8.25*

Bye

Here *a*, *b* and *c* refer to same location in memory still the first three *if*s fail because while comparing the *far* pointers using == (and !=) the full 32-bit value is used and since the 32-bit values are different the *if*s fail. The last *if* however gets satisfied, because while comparing using > (and >=, <, <=) only the offset value is used for comparsion. And the offset values of *a*, *b* and *c* are such that the last condition is satisfied.

A 8.26

Hello
Hi
Hello Hi

Unlike *far* pointers, *huge* pointer are 'normalized' to avoid the strange behaviour as in 8.25 above. A normalized pointer is a 32-bit pointer which has as much of its value in the segment address as possible. Since a segment can start every 16 bytes, this means that the offset will only have a value from 0 to F. Huge pointers are always kept normalized. As a result for any given memory address there is only one possible huge address - segment:offset pair for it.

Chapter 9

Arrays

Q 9.1

What would be the output of the following program?

```
main( )
{
    char a[ ] = "Visual C++" ;
    char *b = "Visual C++" ;
    printf ( "\n%d %d", sizeof ( a ), sizeof ( b ) ) ;
    printf ( "\n%d %d", sizeof ( *a ), sizeof ( *b ) ) ;
}
```

Q 9.2

For the following statements would *arr[3]* and *ptr[3]* fetch the same character? <Yes/No>:

```
char arr[ ] = "Surprised" ;
char *ptr = "Surprised" ;
```

Q 9.3

For the statements in 9.2 does the compiler fetch the character *arr[3]* and *ptr[3]* in the same manner?

Q 9.4

What would be the output of the following program, if the array begins at address 1200?

```
main( )
{
    int arr[ ] = { 2, 3, 4, 1, 6 } ;
    printf ( "%d %d", arr, sizeof ( arr ) ) ;
}
```

Q 9.5

Does mentioning the array name gives the base address in all the contexts?

Q 9.6

What would be the output of the following program, if the array begins at address 65486?

```
main( )
{
    int arr[ ] = { 12, 14, 15, 23, 45 } ;
    printf ( "%u %u", arr, &arr ) ;
}
```

Q 9.7

Are the expressions *arr* and *&arr* same for an array of 10 integers?

Q 9.8

What would be the output of the following program, if the array begins at 65486?

```
main( )
{
    int arr[ ] = { 12, 14, 15, 23, 45 } ;
    printf ( "%u %u", arr + 1, &arr + 1 ) ;
}
```

Q 9.9

When are *char a[ ]* and *char *a* treated as same by the compiler?

Q 9.10

Would the following program compile successfully?

```
main( )
{
    char a[ ] = "Sunstroke" ;
    char *p = "Coldwave" ;
    a = "Coldwave" ;
    p = "Sunstroke" ;
    printf ( "\n%s %s", a, p ) ;
}
```

Q 9.11

What would be the output of the following program?

```
main( )
```

```
{
    float a[] = { 12.4, 2.3, 4.5, 6.7 } ;
    printf ( "\n%d", sizeof ( a ) / sizeof ( a[0] ) ) ;
}
```

Q 9.12

A pointer to a block of memory is effectively same as an array. <True/False>

Q 9.13

What would be the output of the following program if the array begins at 65472?

```
main( )
{
    int a[3][4] = {
                    1, 2, 3, 4,
                    4, 3, 2, 1,
                    7, 8, 9, 0
                  } ;
    printf ( "\n%u %u", a + 1, &a + 1 ) ;
}
```

Q 9.14

What does the following declaration mean:

```
int ( *ptr )[10] ;
```


Q 9.15

If we pass the name of a 1-D *int* array to a function it decays into a pointer to an *int*. If we pass the name of a 2-D array of integers to a function what would it decay into?

Q 9.16

How would you define the function *f( )* in the following program?

```
int arr[MAXROW][MAXCOL] ;
fun ( arr ) ;
```

Q 9.17

What would be the output of the following program?

```
main( )
{
    int a[3][4] = {
                    1, 2, 3, 4,
                    4, 3, 2, 8,
                    7, 8, 9, 0
                  } ;
    int *ptr ;
    ptr = &a[0][0] ;
    fun ( &ptr ) ;
}

fun ( int **p )
{
    printf ( "\n%d", **p ) ;
}
```

Answers

A 9.1

1 1 2
1 1

A 9.2

Yes

A 9.3

No. For *arr[3]* the compiler generates code to start at location arr, move three past it, and fetch the character there. When it sees the expression *ptr[3]* it generates the code to start at location stored in *ptr*, add three to the pointer, and finally fetch the character pointed to.

In other words, *arr[3]* is three places past the start of the object named *arr*, whereas *ptr[3]* is three places past the object pointed to by *ptr*.

A 9.4

1200 10

A 9.5

No. Whenever mentioning the array name gives its base address it is said that the array has decayed into a pointer. This decaying doesn't take place in two situations:

When array name is used with *sizeof* operator.
When the array name is an operand of the & operator.

A 9.6

65486 65486

A 9.7

No. Even though both may give the same adresses as in 9.6 they mean two different things. *arr* gives the address of the first *int*, whereas *&arr* gives the address of array of *ints*. Since these addresses happen to be same the results of the expressions are same.

A 9.8

65488 65496

A 9.9

When using them as formal parameters while defining a function.

A 9.10

No, because we may assign a new string to a pointer but not to an array.

A 9.11

4

A *9.12*

True

A *9.13*

65480 65496

A *9.14*

ptr is a pointer to an array of 10 integers.

A *9.15*

It decays into a pointer to an array and not a pointer to a pointer.

A *9.16*

```
fun ( int a[ ][MAXCOL] )
{
}
```

or

```
fun ( int ( *ptr )[ MAXCOL] )  /* ptr is pointer to an array */
{
}
```

A *9.17*

1

Chapter 10

Strings

Q *10.1*

What would be the output of the following program?

```
main( )
{
    printf ( 5 + "Fascimile" ) ;
}
```

A. Error
B. Fascimile
C. mile
D. None of the above

Q *10.2*

What would be the output of the following program?

```
main( )
{
    char str1[ ] = "Hello" ;
    char str2[ ] = "Hello" ;
    if ( str1 == str2 )
        printf ( "\nEqual" ) ;
    else
```

```
    printf ( "\nUnequal" ) ;
}
```

A. Equal
B. Unequal
C. Error
D. None of the above

Q 10.3

What would be the output of the following program?

```
main( )
{
    printf ( "%c", "abcdefgh"[4] ) ;
}
```

A. Error
B. d
C. e
D. abcdefgh

Q 10.4

What would be the output of the following program?

```
main( )
{
    char str[7] = "Strings" ;
    printf ( "%s", str ) ;
}
```

A. Error
B. Strings
C. Cannot predict

D. None of the above

Q 10.5

How would you output \n on the screen?

Q 10.6

What would be the output of the following program?

```
main( )
{
    char ch = 'A' ;
    printf ( "%d %d", sizeof ( ch ), sizeof ( 'A' ) ) ;
}
```

A. 1 1
B. 1 2
C. 2 2
D. 2 1

Q 10.7

What would be the output of the following program?

```
main( )
{
    printf ( "\n%d %d %d", sizeof ( '3' ), sizeof ( "3" ), sizeof ( 3 ) ) ;
}
```

A. 1 1 1
B. 2 2 2
C. 1 2 2
D. 1 1 1

Q 10.8

Is the following program correct? <Yes/No>

```
main( )
{
    char *str1 = "United" ;
    char *str2 = "Front" ;
    char *str3 ;
    str3 = strcat ( str1, str2 ) ;
    printf ( "\n%s", str3 ) ;
}
```

Q 10.9

How would you improve the code in 10.8 above?

Q 10.10

In the following code which function would get called, the user-defined *strcpy()* or the one in the standard library?

```
main( )
{
    char str1[ ] = "Keep India Beautiful... emigrate!" ;
    char str2[40] ;
    strcpy ( str2, str1 ) ;
    printf ( "\n%s", str2 ) ;
}

strcpy ( char *t, char *s )
{
    while ( *s )
    {
        *t = *s ;
        t++ ;
        s++ ;
    }
    *t = '\0' ;
}
```


Q 10.11

Can you compact the code in *strcpy()* into one line?

Q 10.12

What would be the output of the following program?

```
main( )
{
    char *str[ ] = { "Frogs", "Do", "Not", "Die.", "They", "Croak!" } ;
    printf ( "%d %d", sizeof ( str ), sizeof ( str[0] ) ) ;
}
```

Q 10.13

How would you find the length of each string in the program 10.12 above?

Q 10.14

What is the difference in the following declarations?

```
char *p = "Samuel" ;
char a[ ] = "Samuel" ;
```

Q *10.15*

While handling a string do we always have to process it character by character or there exists a method to process the entire string as one unit.

Answers

A *10.1*

C

A *10.2*

B

A *10.3*

C

A *10.4*

C. Here *str[ ]* has been declared as a 7 character array and into it a 8 character string has been stored. This would result into overwriting of the byte beyond the seventh byte reserved for the array with a '\0'. There is always a possibility that something important gets overwritten which would be unsafe.

A *10.5*

```
printf ( "\\n" ) ;
```


A *10.6*

B

A *10.7*

B

A *10.8*

No, since what is present in memory beyond 'United' is not known and we are attaching 'Front' at the end of 'United', thereby overwriting something, which is an unsafe thing to do.

A *10.9*

```
main( )
{
    char str1[15] = "United" ;
    char *str2 = "Front" ;
    char *str3 ;
    str3 = strcat ( str1, str2 ) ;
    printf ( "\n%s", str3 ) ;
}
```

A *10.10*

User-defined *strcpy()*

A *10.11*

```
strcpy ( char *t, char *s )
{
```

```
    while ( *t++ = *s++ ) ;
}
```

A *10.12*

12 2

A *10.13*

```
main( )
{
    char *str[ ] = { "Frogs", "Do", "Not", "Die.", "They", "Croak!" } ;
    int i ;

        for ( i = 0 ; i <= 5 ; i++ )
            printf ( "\n%s %d", str[i], strlen ( str[i] ) ) ;
}
```

A *10.14*

Here *a* is an array big enough to hold the message and the '\0' following the message. Individual characters within the array can be changed but the address of the array would remain same.

On the other hand, *p* is a pointer, initialized to point to a string constant. The pointer *p* may be modified to point to another string, but if you attempt to modify the string at which *p* is pointing the result is undefined.

A *10.15*

A string can be processed only on a character by character basis.

Chapter 11

Structures, Unions and Enumerations

Q *11.1*

What is the similarity between a structure, union and an enume

Q *11.2*

Would the following declaration work?

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

Q *11.3*

Can a structure contain a pointer to itself?

Q *11.4*

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

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

Q 11.5

How will you eliminate the problem in 11.4 above?

Q 11.6

Point out the 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 ;
}
```


Q 11.7

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 ) ;
}
```

Q 11.8

Can you suggest a better way to write the program in 11.7 above?

Q 11.9

How would you free the memory allocated in 11.8 above?

Q 11.10

Can you rewrite the program in 11.8 such that while freeing the memory only one call to *free()* would suffice?

Q 11.11

What would be the output of the following program?

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

Q 11.12

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

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


Q 11.13

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

Q 11.14

How are structure passing and returning implemented by the compiler?

Q 11.15

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

Q 11.16

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 ) ;
```

Q 11.17

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

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

Q 11.18

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 <= 9 ; i++ )
        scanf ( "%s %f", e[i].name, &e[i].sal ) ;
}
```

Q 11.19

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

Q 11.20

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

Q 11.21

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

```
struct transaction
{
    int sno ;
    char desc[30] ;
    char dc ;
    float amount ;
}
/* Here is the main program. */

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 ) ;
}
```

Q 11.22

What would be the output of the following program?

```
main( )
{
    struct a
    {
```

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

Q *11.23*

What's the difference between a structure and a union?

Q *11.24*

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

Q *11.25*

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

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

Q *11.26*

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

Q *11.27*

Since enumerations have integral type and enumeration constants are of type int can we freely intermix them with other integral types, without errors? <Yes/No>

Q *11.28*

Is there an easy way to print enumeration values symbolically?

Q *11.29*

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

Q *11.30*

Can we have an array of bit fields? <Yes/No>

Answers

A *11.1*

All of them let you define new data types.

A *11.2*

Yes

A 11.3

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

A 11.4

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 Q 11.4 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.

A 11.5

To fix this code, first give the structure a name ("struct 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 definition 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 could 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 defined yet; this you're allowed to do. It's a matter of style which of the above solutions would you prefer.

A 11.6

The *struct emp* is mentioned in the prototype of the function *modify()* before defining 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.

A 11.7

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, as the structure 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.

A 11.8

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

```
#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( )*.

A 11.9

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


A 11.10

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

main( )
{
    char newname[ ] = "Rahul" ;
    char *buf = 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 ( "\n%d %s", p->len, p->name ) ;
    free ( p ) ;
}
```

A 11.11

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*.

A 11.12

Structures can't 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 byte-by-byte comparison could fail while comparing the bits present in unused paddings in the structure (such padding is used to keep the alignment of later fields correct). A field-by-field comparison might require unacceptable amounts of repetitive code for large structures. Also, any compiler-generated comparison could not be expected to compare pointer fields appropriately in all cases; for example, it's often appropriate to compare *char* * fields with *strcmp()* rather than with ==.

A 11.13

```
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 ( "The structures are equal" ) ;
    else
        printf ( "The structures are unequal" ) ;
}

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'll have to write your own function to do so which carries out the comparison field by field.

A 11.14

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

A 11.15

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 from the address *&e*. Data files written as memory images with *fwrite()*, however, will not be portable, particularly if they contain floating-point fields or pointers. This is because memory layout of structures is machine and compiler dependent. Different compilers may use different amounts of padding, and the sizes 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.

A 11.16

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* would get written (and not the string pointed by it). When this structure is read back the address would be read back but it is quite unlikely that the desired string would be present at this address in memory.

A 11.17

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 follows 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 cannot perform unaligned accesses at all and require that all 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 a structure from largest to smallest. You can sometimes get more control over size and alignment by using bitfields, although they have their own drawbacks.

A 11.18

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 *atoff()*. 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.

A 11.19

You can use the offset macro given below. 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 ( "\t%d", offsetofage );

    offsetofsal = offset ( struct a, sal );
    printf ( "\t%d", offsetofsal );

}
```

The output of this program would be:

0 15 17

A 11.20

The entire structure itself and not its base address.

A 11.21

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

A 11.22

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 (9 bits to be precise) the size of the structure is being reported as 2 bytes.

A 11.23

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.

A 11.24

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 be increased by padding.

A 11.25

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 initialise 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.

```
union a
{
    int i;
```

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

A 11.26

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 is that we have no control over the sizes of enumeration variables.

A 11.27

Yes

A 11.28

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.

A 11.29

Bitfields 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.

A 11.30

No

Chapter 12

Input/Output

Q 12.1

What would be the output of the following program?

```
main( )
{
    int a = 250 ;
    printf ( "%1d", a ) ;
}
```

Q 12.2

What would be the output of the following program?

```
main( )
{
    float a = 3.15529 ;
    printf ( "\n%6.2f", a ) ;
    printf ( "\n%6.3f", a ) ;
    printf ( "\n%5.4f", a ) ;
    printf ( "\n%2.1f", a ) ;
    printf ( "\n%0.0f", a ) ;
}
```

Q 12.3

In the following code

```
#include <stdio.h>
main( )
{
    FILE *fp ;
    fp = fopen ( "trial", "r" ) ;
}
```

fp points to

A. The first character in the file
B. A structure which contains a *char* pointer which points to the first character in the file
C. The name of the file
D. None of the above

Q 12.4

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

```
#include "stdio.h"
main ( )
{
    unsigned char ;
    FILE *fp ;

    fp = fopen ( "trial", "r" ) ;
    while ( ( ch = getc ( fp ) ) != EOF )
        printf ( "%c", ch ) ;

    fclose ( fp ) ;
}
```


Q 12.5

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

```
#include "stdio.h"
main( )
{
    unsigned char ;
    FILE *fp ;

    fp = fopen ( "trial", "r" ) ;
    if ( !fp )
    {
        printf ( "Unable to open file" ) ;
        exit( ) ;
    }
    fclose ( fp ) ;
}
```

Q 12.6

If a file contains the line "I am a boy\r\n" then on reading this line into the array *str* using *fgets()* what would *str* contain?

A. "I am a boy\r\n\0"
B. "I am a boy\r\0"
C. "I am a boy\n\0"
D. "I am a boy"

Q 12.7

Point out the error if any in the following program.

```
#include "stdio.h"
```

```
main( )
{
    FILE *fp ;

    fp = fopen ( "trial", "r" ) ;
    fseek ( fp, 20, SEEK_SET ) ;
    fclose ( fp ) ;
}
```

Q 12.8

To print out *a* and *b* given below, which *printf()* statement would use?

```
float a = 3.14 ;
double b = 3.14 ;
```

A. printf ("%f %f", a, b) ;
B. printf ("%Lf %f", a, b) ;
C. printf ("%Lf %Lf", a, b) ;
D. printf ("%f %Lf", a, b) ;

Q 12.9

To scan *a* and *b* given below which *scanf()* statement would you use?

```
float a ;
double b ;
```

A. scanf ("%f %f", &a, &b) ;
B. scanf ("%Lf %Lf", &a, &b) ;
C. scanf ("%f %Lf", &a, &b) ;
D. scanf ("%f %lf", &a, &b) ;

Q 12.10

Point out the error in the following program.

```
#include "stdio.h"
main( )
{
    FILE *fp ;
    char str[80] ;

    fp = fopen ( "trial", "r" ) ;
    while ( !feof ( fp ) )
    {
        fgets ( str, 80, fp ) ;
        puts ( str ) ;
    }
    fclose ( fp ) ;
}
```

Q 12.11

Point out the error in the following program.

```
#include "stdio.h"
main( )
{
    char ch ;
    int i ;
    scanf ( "%c", &i ) ;
    scanf ( "%d", &ch ) ;
    printf ( "%c %d", ch, i ) ;
}
```

Q 12.12

What would be the output of the following program?

```
main( )
{
    printf ( "\n%%%%" ) ;
}
```

Q 12.13

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

```
#include "stdio.h"
main( )
{
    FILE *fp ;

    fp = fopen ( "c:\tc\trial", "w" ) ;
    if ( !fp )
        exit( ) ;
    fclose ( fp ) ;
}
```

Q 12.14

Would the follwing code work? <Yes/No> If yes, what would be the output?

```
main( )
{
    int n = 5 ;
    printf ( "\nn=%*d", n, n ) ;
}
```


Q 12.15

What is the * in the *printf()* of 12.14 indicative of?

Q 12.16

Can we specify variable field width in a *scanf()* format string? <Yes/No>

Q 12.17

To tackle a double in *printf()* we can use *%f*, whereas in *scanf()* we should use *%lf*. <True/False>

Q 12.18

Out of *fgets()* and *gets()* which function is safe to use?

Q 12.19

A file written in text mode can be read back in binary mode. <True/False>

Q 12.20

We should not read after a write to a file without an intervening call to *fflush(), fseek()* or *rewind()*. <True/False>

Answers

A 12.1

250

A 12.2

3.16
3.155
3.1553
3.2
3

A 12.3

B

A 12.4

EOF has been defined as *#define EOF -1* in the file "stdio.h" and an *unsigned char* ranges from 0 to 255 hence when EOF is read from the file it cannot be accommodated in *ch*. Solution is to declare *ch* as an *int*.

A 12.5

No error.

A 12.6

C

A 12.7

Instead of 20 use 20L since *fseek()* needs a *long* offset value.

A 12.8

A. It is possible to print a *double* using *%f*.

A 12.9

D

A 12.10

The last line from the file "trial" would be read twice. To avoid this, use:

```
while ( fgets ( str, 80, fp ) != NULL )
    puts ( str ) ;
```

A 12.11

You would not get a chance to supply a character for the second *scanf()* statement. Solution is to precede the second *scanf()* with the following statement.

```
fflush ( stdin ) ;
```

This would flush out the enter hit for the previous *scanf()* to be flushed out from the input stream, i.e. keyboard.

A 12.12

%%

A 12.13

The path of the filename should have been written as "c:\\tc\\trial".

A 12.14

Yes.

n= 5

A 12.15

It indicates that an *int* value from the argument list will be used for field width. In the argument list the width precedes the value to be printed. In this case the format specifier becomes *%5d*.

A 12.16

No. A '*' in *scanf()* format string after a % sign is used for suppression of assignment. That is, the current input field is scanned but not stored.

A 12.17

True

A 12.18

fgets(), because unlike *fgets()*, *gets()* cannot be told the size of the buffer into which the string supplied would be stored. As a result there is always a possibility of overflow of buffer.

A 12.19

False

A 12.20

True

Chapter 13

Command Line Arguments

Q *13.1*

What do the the 'c' and 'v' in *argc* and *argv* stand for?

Q *13.2.*

According to ANSI specifications which is the correct way of declaring *main()* when it receives command line arguments?

A. main (int argc, char *argv[ ])

B. main (argc, argv)
int argc ; char *argv[ ] ;

C. main( )
{
int argc ; char *argv[ ] ;
}

D. None of the above

Q *13.3*

What would be the output of the following program?

```
/* sample.c */
main ( int argc, char **argv )
```

```
{
    argc = argc - ( argc - 1 ) ;
    printf ( "%s", argv[argc - 1] ) ;
}
```

Q 13.4.

If different command line arguments are supplied at different times would the ouptut of the following program change? <Yes/No>

```
main ( int argc, char *argv[ ] )
{
    printf ( "%d", argv[argc] ) ;
}
```

Q 13.5.

If the following program (*myprog*) is run from the command line as

```
myprog 1 2 3
```

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    int i ;
    for ( i = 0 ; i < argc ; i++ )
        printf ( "%s ", argv[i] ) ;
}
```

Q 13.6.

If the following program (*myprog*) is run from the command line as


```
myprog 1 2 3
```

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    int i ;
    i = argv[1] + argv[2] + argv[3] ;
    printf ( "%d", i ) ;
}
```

A. 123
B. 6
C. Error
D. "123"

Q 13.7.

If the following program (*myprog*) is run from the command line as

```
myprog 1 2 3
```

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    int i, j = 0 ;
    for ( i = 0 ; i < argc ; i++ )
        j = j + atoi ( argv[i] ) ;

    printf ( "%d", j ) ;
}
```

A. 123
B. 6
C. Error

D. "123"

Q 13.8.

Would the following program give the same output at all times? <Yes/No>

```
main ( int argc, char *argv[ ] )
{
    strcpy ( argv[0], "hello" ) ;
    strcpy ( argv[1], "good morining" ) ;
    printf ( "%s %s", argv[0], argv[1] ) ;
}
```

Q 13.9.

If the following program (*myprog*) is run from the command line as

```
myprog one two three
```

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    printf ( "%s", *++argv ) ;
}
```

Q 13.10.

If the following program (*myprog*) is run from the command line as

```
myprog one two three
```

what would be the output?


```
main ( int argc, char *argv[ ] )
{
    printf ( "%c", ++**++argv ) ;
}
```

Q 13.11.

The variables *argc* and *argv* are always local to main <True/False>.

Q 13.12.

The maximum combined length of the command line arguments including the spaces between adjacent arguments is

A. 128 characters
B. 256 characters
C. 67 characters
D. It may vary from one operating system to another

Q 13.13.

What will the following program output?

```
main ( int argc, char *argv[ ], char *env[ ] )
{
    int i ;
    for ( i = 1 ; i < argc ; i++ )
        printf ( "%s ", env[i] ) ;
}
```

A. List of all environment variables
B. List of all command line arguments
C. Error
D. NULL

Q 13.14.

If the following program (*myprog*) is run from the command line as

```
myprog "*.c"
```

what would be the output?

```
main ( int argc, char *argv[] )
{
    int i ;
    for ( i = 1 ; i < argc ; i++ )
        printf ( "%s ", argv[i] ) ;
}
```

A. *.c
B. List of all .c files in the current directory
C. "*.c"
D. None of the above

Q 13.15.

If the following program (*myprog*) is run from the command line as

```
myprog *.c
```

what would be the output?

```
main ( int argc, char *argv[] )
{
    int i ;
    for ( i = 1 ; i < argc ; i++ )
        printf ( "%s ", argv[i] ) ;
}
```


A. *.c
B. List of all .c files in the current directory
C. "*.c"
D. None of the above

Q 13.16.

If we want that any wildcard characters in the command line arguments should be appropriately expanded, are we required to make any special provision? If yes, which?

Q 13.17.

Does there exist any way to make the command line arguments available to other functions without passing them as arguments to the function? <Yes/No>

Q 13.18.

If the following program (*myprog*) is run from the command line as

```
myprog Jan Feb Mar
```

what would be the output?

```
#include "dos.h"
main( )
{
    fun( ) ;
}
fun( )
{
    int i ;
    for ( i = 0 ; i < _argc ; i++ )
```

```
    printf ( "%s ", _argv[i] ) ;
}
```

Q 13.19.

If the following program (*myprog*) is present in the directory *c:\bc\tucs* then what would be its output?

```
main ( int argc, char *argv[ ] )
{
    printf ( "%s ", argv[0] ) ;
}
```

A. MYPROG
B. C:\BC\TUCS\MYPROG
C. Error
D. C:\BC\TUCS

Q 13.20.

Which is an easy way to extract *myprog* from the output of program 13.19 above?

Q 13.21.

Which of the following is true about *argv*?

A. It is an array of character pointers
B. It is a pointer to an array of character pointers
C. It is an array of strings
D. None of the above

Q 13.22.

If the following program (*myprog*) is run from the command line as

```
myprog monday tuesday wednesday thursday
```

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    while ( --argc > 0 )
        printf ( "%s ", *++argv ) ;
}
```

A. myprog monday tuesday wednesday thursday
B. monday tuesday wednesday thursday
C. myprog tuesday thursday
D. None of the above

Q 13.23.

If the following program (*myprog*) is run from the command line as

```
myprog friday tuesday sunday
```

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    printf ( "%c", ( *++argv )[0] ) ;
}
```

A. m
B. f
C. myprog

D. friday

Q 13.24.

If the following program (*myprog*) is run from the command line as

myprog friday tuesday sunday

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    printf ( "%c", **++argv ) ;
}
```

A. m
B. f
C. myprog
D. friday

Q 13.25.

If the following program (*myprog*) is run from the command line as

myprog friday tuesday sunday

what would be the output?

```
main ( int argc, char *argv[ ] )
{
    printf ( "%c", *++argv[1] ) ;
}
```

A. r
B. f

C. m
D. y

Q 13.26.

If the following program (*myprog*) is run from the command line as

myprog friday tuesday sunday

what would be the output?

```
main ( int sizeofargv, char *argv[] )
{
    while ( sizeofargv )
        printf ( "%s ", argv[--sizeofargv] ) ;
}
```

A. myprog friday tuesday sunday
B. myprog friday tuesday
C. sunday tuesday friday myprog
D. sunday tuesday friday

Answers

A 13.1

Count of arguments and vector (array) of arguments

A 13.2

A

A 13.3

C:\SAMPLE.EXE

A *13.4*

No

A *13.5*

C:\MYPROG.EXE 1 2 3

A *13.6*

C

A *13.7*

B. When *atoi()* tries to convert *argv[0]* to a number it cannot do so (*argv[0]* being a file name) and hence returns a zero.

A *13.8*

No

A *13.9*

one

A *13.10*

p

A *13.11*

True

A *13.12*

D

A *13.13*

B

A *13.14*

A

A *13.15*

A

A *13.16*

Yes. Compile the program as

```
tcc myprog wildargs.obj
```

This compiles the file *myprog.c* and links it with the wildcard expansion module WILDARGS.OBJ, then run the resulting executable file MYPROG.EXE

If you want the wildcard expansion to be default so that you won't have to link your program explicitly with WILDARGS.OBJ, you can modify your standard C?.LIB library files to have WILDARGS.OBJ linked automatically. To achieve this we have to remove SETARGV from the library and add WILDARGS. The following commands will invoke the Turbo Librarian to modify all the standard library files (assuming the current directory contains the standard C libraries, and WILDARGS.OBJ):

```
tlib cs -setargv +wildargs
tlib cc -setargv +wildargs
tlib cm -setargv +wildargs
tlib cl -setargv +wildargs
tlib ch -setargv +wildargs
```

A 13.17

Yes. Using the predefined variables *_argc*, *_argv*.

A 13.18

C:\MYPROG.EXE Jan Feb Mar

A 13.19

B

A 13.20

```
#include "dir.h"
main ( int argc, char *argv[ ] )
{
    char drive[3], dir[50], name[8], ext[3] ;
    printf ( "\n%s", argv[0] ) ;
    fnsplit ( argv[0], drive, dir, name, ext ) ;
    printf ( "\n%s\n%s\n%s\n%s", drive, dir, name, ext ) ;
}
```


A 13.21

A

A 13.22

B

A 13.23

B

A 13.24

B

A 13.25

A

A 13.26

C

Chapter 14

Bitwise Operators

Q 14.1

What would be the output of the following program?

```
main( )
{
    int i = 32 , j = 0x20, k, l, m ;
    k = i | j ;
    l = i & j ;
    m = k ^ l ;
    printf ( "%d %d %d %d %d", i, j, k, l, m ) ;
}
```

A. 32 32 32 32 0
B. 0 0 0 0 0
C. 0 32 32 32 32
D. 32 32 32 32 32

Q 14.2

What would be the output of the following program?

```
main( )
{
    unsigned int m = 32 ;
```

```
    printf ( "%x", ~m ) ;
}
```

A. ffff
B. 0000
C. ffdf
D. ddfd

Q 14.3

What would be the output of the following program?

```
main( )
{
    unsigned int a = 0xffff ;
    ~a ;
    printf ( "%x", a ) ;
}
```

A. ffff
B. 0000
C. 00ff
D. None of the above

Q 14.4

Point out the error in the following program.

```
main( )
{
    unsigned int a, b, c, d, e, f ;
    a = b = c = d = e = f = 32 ;
    a <<= 2 ;
    b >>= 2 ;
    c ^= 2 ;
```



```
    d |= 2 ;
    e &= 2 ;
    f ~= 2 ;
    printf ( "\n%x %x %x %x %x %x", a, b, c, d, e, f ) ;
}
```

Q 14.5

To which numbering system can the binary number 1011011111000101 be easily converted to?

Q 14.6

Which bitwise operator is suitable for checking whether a particular bit is on or off?

Q 14.7

Which bitwise operator is suitable for turning off a particular bit in a number?

Q 14.8

Which bitwise operator is suitable for putting on a particular bit in a number?

Q 14.9

On left shifting, the bits from the left are rotated and brought to the right and accommodated where there is empty space on the right? <True/False>

Q 14.10

Left shifting a number by 1 is always equivalent to multiplying it be 2. <Yes/No>

Q 14.11

Left shifting an *unsigned int* or *char* by 1 is always equivalent to multiplying it be 2. <Yes/No>

Q 14.12

What would be the output of the following program?

```
main( )
{
    unsigned char i = 0x80 ;
    printf ( "\n%d", i << 1 ) ;
}
```

A. 0
B. 256
C. 100
D. None of the above

Q 14.13

What is the following program doing?

```
main( )
{
    unsigned int m[ ] = { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 } ;
    unsigned char n, i ;
```



```
    scanf ( "%d", &n ) ;
    for ( i = 0 ; i <= 7 ; i++ )
    {
        if ( n & m[i] )
            printf ( "\nyes" ) ;
    }
}
```

A. Putting off all bits which are on in the number *n*
B. Testing whether the individual bits of *n* are on or off
C. This program would give an error
D. None of the above

Q 14.14

What does the following program do?

```
main( )
{
    char *s ;
    s = fun ( 128, 2 ) ;
    printf ( "\n%s", s ) ;
}

fun ( unsigned int num, int base )
{
    static char buff[33] ;
    char *ptr ;

    ptr = &buff [ sizeof ( buff ) - 1 ] ;
    *ptr = '\0' ;
    do
    {
        *--ptr = "0123456789abcdef"[ num % base ] ;
        num /= base ;
```

```
    } while ( num != 0 ) ;

    return ptr ;
}
```

A. It converts a number to given base
B. It gives a compilation error
C. None of the above

Q 14.15

```
#define CHARSIZE 8
#define MASK(y) ( 1 << y % CHARSIZE )
#define BITSLOT(y) ( y / CHARSIZE )
#define SET( x, y) ( x[BITSLOT(y)] |= MASK(y) )
#define TEST(x, y) ( x[BITSLOT(y)] & MASK(y) )
#define NUMSLOTS(n) ((n + CHARSIZE - 1) / CHARSIZE)
```

Given the above macros how would you

- declare an array *arr* of 50 bits
- put the 20th bit on
- test whether the 40th bit is on or off

Q 14.16

Consider the macros in Problem 14.15 above. On similar lines can you define a macro which would clear a given bit in a bit array?

Q 14.17

What does the following program do?

```
main( )
{
    unsigned int num ;
    int c = 0 ;
    scanf ( "%u", &num ) ;
    for ( ; num ; num >>= 1 )
    {
        if ( num & 1 )
            c++ ;
    }
    printf ( "%d", c ) ;
}
```


A. It counts the number of bits which are on in the number *num*.
B. It sets all bits in the number *num* to 1.
C. It sets all bits in the number *num* to 0.
D. None of the above

Q 14.18

What would be the output of the following program?

```
main( )
{
    printf ( "\n%x", -1 >> 4 ) ;
}
```

A. ffff
B. 0fff
C. 0000
D. fff0

Q 14.19

In the statement *expression1* >> *expression2* if *expression1* is a signed integer with its leftmost bit set to 1 then on right-shifting it the

result of the statement would vary from computer to computer. <True/False>

Q 14.20

What does the following program do?

```
main( )
{
    unsigned int num ;
    int i ;
    scanf ( "%u", &num ) ;
    for ( i = 0 ; i < 16 ; i++ )
        printf ( "%d", ( num << i & 1 << 15 ) ? 1 : 0 ) ;
}
```

A. It prints all even bits from *num*
B. It prints all odd bits from *num*
C. It prints binary equivalent of *num*
D. None of the above

Answers

A 14.1

A

A 14.2

C

A 14.3

A

A 14.4

Error is in *f* ~= 2, since there is no operator like ~=.

A 14.5

Hexadecimal, since each 4-digit binary represents one hexadecimal digit.

A 14.6

The & operator

A 14.7

The & operator

A 14.8

The | operator

A 14.9

False

A 14.10

No

A 14.11

Yes

A 14.12

B

A 14.13

B

A 14.14

A

A 14.15

```
char arr[NUMSLOTS(50)] ;
SET(arr, 20 ) ;
if ( TEST ( arr, 40 ) )
```

A 14.16

```
#define CLEAR(x, y) ( x[BITSLOT(y)] &= ~MASK(y) )
```


A 14.17

A

A 14.18

A. On computers which don't support sign extension you may get B.

A 14.19

True

A 14.20

C

Chapter 15

Subtleties of *typedef*

Q 15.1

Are the properties of *i*, *j* and *x*, *y* in the following program same? <Yes/No>

```
typedef unsigned long int uli ;
uli i, j ;
unsigned long int x, y ;
```

Q 15.2

What is the type of *compare* in the following code segment?

```
typedef int ( *ptrtofun )( char *, char * ) ;
ptrtofun compare ;
```

Q 15.3

What are the advantages of using *typedef* in a program?

Q 15.4

Is there any difference in the *#define* and the *typedef* in the following code? If yes, what?

```
typedef char * string_t;
#define string_d char *
string_t s1, s2;
string_d s3, s4;
```

Q 15.5

*typedef*s have the advantage that obey scope rules, that is, they can be declared local to a function or a block whereas *#define*s always have a global effect. <True/False>

Q 15.6

Point out the error in the following declaration and suggest atleast three solutions for it.

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

Q 15.7

In the following code can we declare a new *typedef* name *emp* even though *struct employee* has not been completely defined while using *typedef*? <Yes/No>


```
typedef struct employee *ptr ;
struct employee
{
    char name[20] ;
    int age ;
    ptr next ;
} ;
```

Q 15.8

There is an error in the following declarations. Can you rectify it?

```
typedef struct
{
    int data1 ;
    BPTR link1 ;
} *APTR ;

typedef struct
{
    int data2 ;
    APTR link2 ;
} *BPTR ;
```

Q 15.9

What do the following declarations mean?

```
typedef char *pc ;
typedef pc fpc( ) ;
typedef fpc *pfpc ;
typedef pfpc fpfpc( ) ;
typedef fpfpc *pfpfpc ;
pfpfpc a[N] ;
```

Q 15.10

How would you define *a[N]* in 15.9 above without using typedef?

Q 15.11

Improve the following code using typedef.

```
struct node
{
    int data1 ; float data2 ;
    struct node *left ;
    struct node *right ;
} ;
struct node *ptr ;
ptr = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
```

Q 15.12

Is the following declaration acceptable? <Yes/No>

```
typedef long no, *ptrtono ;
no n ;
ptrtono p ;
```

Q 15.13

In the following code what is constant, p or the character it is pointing to?

```
typedef char * charp ;
const charp p ;
```


Q 15.14

In the following code is p2 an integer or an integer pointer?

```
typedef int * ptr ;
ptr p1, p2 ;
```

Answers

A 15.1

Yes

A 15.2

It is a pointer to function which receives two character pointers and returns an integer.

A 15.3

There are three main reasons for using *typedefs*:

(a) It makes writing of complicated declarations a lot easier. This helps in eliminating a lot of clutter in the program.

(b) It helps in achieving portability in programs. That is, if we use *typedefs* for data types that are machine-dependent, only the *typedefs* need change when the program is moved to a new machine platform.

(c) It helps in providing a better doumentation for a program. For example, a node of a doubly linked list is better understood as *ptrtolist* rather than just a pointer to a complicated structure.

A 15.4

In these declarations, *s1*, *s2*, and *s3* are all treated as *char **, but *s4* is treated as a *char*, which is probably not the inetention.

A 15.5

True

A 15.6

A *typedef* declaration cannot be used until it is defined, and in our example it is not yet defined at the point where the *link* field is declared.

We can fix this problem in three ways:

(a) Give the structure a name, say *node* and then declare the *link* field as a simple *struct node ** as shown below:

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

(b) Keep the *typedef* declaration separate from the structure definition:

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


(c) Precede the structure declaration with *typedef*, so that you can use the NODEPTR *typedef* when declaring the *link* field:

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

A 15.7

Yes

A 15.8

The problem with the code is the compiler doesn't know about BPTR when it is used in the first structure declaration. We are violating the rule that a *typedef* declaration cannot be used until it is defined, and in our example it is not yet defined at the point where the *link1* field is declared.

To avoid this problem we can define the structures as shown below:

```
struct a
{
    int data1 ;
    struct b *link1 ;
} ;
struct b
{
    int data2 ;
    struct a *link2 ;
```

```
};

typedef struct a *APTR ;
typedef struct b *BPTR ;
```

The compiler can accept the field declaration *struct b *ptr1* within *struct a*, even though it has not yet heard of *struct b* (which is "incomplete" at that point). Occasionally, it is necessary to precede this couplet with the line

```
    struct b ;
```

This empty declaration masks the pair of structure declarations (if in an inner scope) from a different *struct b* in an outer scope. After declaring the two structures we can then declare the *typedef*s separately as shown above.

Alternatively, we can also define the *typedef*s before the structure definitions, in which case you can use them when declaring the *link* pointer fields as shown below:

```
typedef struct a *APTR;
typedef struct b *BPTR;
struct a
{
    int data1 ;
    BPTR link1 ;
};
struct b
{
    int data2 ;
    APTR link2 ;
};
```


A 15.9

pc	is a pointer to char.
fpc	is function returning pointer to char
pfpc	is pointer to a function returning pointer to char
fpfpc	is a function returning pointer to a function returning pointer to char
pfpfpc	is a pointer to function returning pointer to a function returning pointer to char
pfpfpc a[N]	is an array of N pointers to functions returning pointers to functions returning pointers to characters

A 15.10

```
char *(*(*a[N])( ))( ) ;
```

A 15.11

```
typedef struct node * treeptr
typedef struct node
{
    int data1 ;
    float data2 ;
    treeptr *left ;
    treeptr *right ;
} treenode ;
treeptr ptr ;
ptr = ( treeptr ) malloc ( sizeof ( treenode ) ) ;
```

A 15.12

Yes

A *15.13*

p is constant

A *15.14*

Integer pointer

Chapter 16

The *const* Phenomenon

Q *16.1*

Point out the error in the following program.

```
main( )
{
    const int x ;
    x = 128 ;
    printf ( "%d", x ) ;
}
```

Q *16.2*

What would be the output of the following program?

```
main( )
{
    int y = 128 ;
    const int x = y ;
    printf ( "%d", x ) ;
}
```

A. 128
B. Garbage value
C. Error

D. 0

Q 16.3

What would be the output of the following program?

```
main( )
{
    const int x = get( ) ;
    printf ( "%d", x ) ;
}
get( )
{
    return ( 20 ) ;
}
```

A. 20
B. Garbage value
C. Error
D. 0

Q 16.4

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

```
#define MAX 128
main( )
{
    const int max = 128 ;
    char array[max] ;
    char string[MAX] ;
    array[0] = string[0] = 'A' ;
    printf ( "%c %c", array[0], string[0] ) ;
}
```


Q 16.5

Point out the error in the following program.

```
main( )
{
    char mybuf[ ] = "Zanzibar" ;
    char yourbuf[ ] = "Zienckewiz" ;
    char * const ptr = mybuf ;
    *ptr = 'a' ;
    ptr = yourbuf ;
}
```

Q 16.6

Point out the error in the following program.

```
main( )
{
    char mybuf[ ] = "Zanzibar" ;
    char yourbuf[ ] = "Zienckewiz" ;
    const char *ptr = mybuf ;
    *ptr = 'a' ;
    ptr = yourbuf ;
}
```

Q 16.7

Point out the error in the following program.

```
main( )
{
    char mybuf[ ] = "Zanzibar" ;
    const char * const ptr = "Hello" ;
```

```
    ptr = mybuf ;
    *ptr = 'M' ;
}
```

Q 16.8

What does the following prototype indicate?

```
strcpy ( char *target, const char *source )
```

Q 16.9

What does the following prototype indicate?

```
const char *change ( char *, int )
```

Q 16.10

Point out the error in the following program.

```
main( )
{
    const char *fun( ) ;
    char *ptr = fun( ) ;
}
const char *fun( )
{
    return "Hello" ;
}
```

Q 16.11

Is there any error in the following program? <Yes/No>


```
main( )
{
    const char *fun( ) ;
    char *ptr = fun( ) ;
}
const char *fun( )
{
    return "Hello" ;
}
```

Q 16.12

Point out the error in the following program.

```
main( )
{
    const char *fun( ) ;
    *fun( ) = 'A' ;
}
const char *fun( )
{
    return "Hello" ;
}
```

Q 16.13

What do you mean by *const* correctness?

Q 16.14

What would be the output of the following program?

```
main( )
{
```

```
    const int x = 5 ;
    int *ptrx ;
    ptrx = &x;
    *ptrx = 10;
    printf ( "%d", x ) ;
}
```

A. 5
B. 10
C. Error
D. Garbage value

Q 16.15

What would be the output of the following program?

```
main( )
{
    const int x = 5 ;
    const int *ptrx ;
    ptrx = &x;
    *ptrx = 10;
    printf ( "%d", x ) ;
}
```

A. 5
B. 10
C. Error
D. Garbage value

Q 16.16

Point out the error in the following program.

```
main( )
```



```
{
    const int k = 7 ;
    int * const q = &k ;
    printf ( "%d", *q ) ;
}
```

Q 16.17

What is the difference in the following declarations?

```
const char *s ;
char const *s ;
```

Q 16.18

What is the difference in the following declarations?

```
const char * const s ;
char const * const s ;
```

Q 16.19

Is the following a properly written function? If not, why not?

```
int fun ( const int n )
{
    int a ;
    a = n * n ;
    return a ;
}
```

Answers

A 16.1

Nowhere other than through initialization can a program assign a value to a *const* identifier. *x* should have been initialised where it is declared.

A 16.2

A

A 16.3

A

A 16.4

The dimension of an array must always be a positive non-zero integer constant which *max* is not.

A 16.5

ptr pointer is constant. In *ptr = yourbuf* the program is trying to modify it, hence an error.

A 16.6

ptr can be modified but not the object that it is pointing to. Hence **ptr = 'A'* gives an error.

A 16.7

ptr is a constant pointer to a constant object. We can neither modify *ptr* nor the object it is pointing to.

A 16.8

We can modify the pointers *source* as well as *target*. However, the object to which *source* is pointing cannot be modified.

A 16.9

The function *change()* receives a *char* pointer and an *int* and returns a pointer to a constant *char*.

A 16.10

Warning: Suspicious pointer conversion. This occurs because caller *fun()* returns a pointer to a *const* character which is being assigned to a pointer to a non-const.

A 16.11

No

A 16.12

fun() returns a pointer to a *const* character which cannot be modified.

A 16.13

A program is 'const correct' if it never changes (a more common term is mutates) a constant object.

A 16.14

B

A 16.15

C

A 16.16

Warning: Suspicious pointer conversion. *k* can be pointed to only by a pointer to a *const*; *q* is a *const* pointer.

A 16.17

There is no difference.

A 16.18

There is no difference.

A 16.19

Since *fun()* would be called by value there is no need to declare *n* as a *const* since passing by value already prevents *fun()* from modifying *n*.

Chapter 17

Memory Allocation

Q 17.1

What would be the output of the following program?

```
main( )
{
    char *s ;
    char * fun( ) ;
    s = fun( ) ;
    printf ( "%s", s ) ;
}

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

Q 17.2

What is the solution for the problem in program 17.1 above?

Q 17.3

Does there exist any other solution for the problem in 17.1?

Q 17.4

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

Q 17.5

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

Q 17.6

How would 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]*?

Q 17.7

How would 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.

Q 17.8

How many bytes would be allocated by the following code?

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
main( )
{
    int ( *p )[MAXCOL] ;
    p = ( int ( * ) [MAXCOL] ) malloc ( MAXROW * sizeof ( *p ) ) ;
}
```


Q 17.9

What would be the output of the following program?

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
main( )
{
    int ( *p )[MAXCOL] ;
    p = ( int ( * ) [MAXCOL] ) malloc ( MAXROW * sizeof ( *p ) ) ;
    printf ( "%d %d", sizeof ( p ), sizeof ( *p ) ) ;
}
```

Q 17.10

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

```
#define MAXROW 3
#define MAXCOL 4
main( )
{
    int i, j ;
    int ( *p )[MAXCOL] ;
```

```
    p = ( int ( * )[MAXCOL] ) malloc ( MAXROW * sizeof ( *p ) ) ;
}
```

Q 17.11

How many bytes would be allocated by the following code?

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
main( )
{
    int ( *p )[MAXCOL][MAXROW] ;
    p = ( int ( * ) [MAXROW][MAXCOL] ) malloc ( sizeof ( *p ) ) ;
}
```

Q 17.12

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

Q 17.13

How many bytes of memory would the following code reserve?

```
#include "alloc.h"
main( )
{
    int *p ;
    p = ( int * ) malloc ( 256 * 256 ) ;
    if ( p == NULL)
        printf ( "Allocation failed" ) ;
}
```


Q 17.14

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

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
main( )
{
    int **p, i ;

    p = ( int ** ) malloc ( MAXROW * sizeof ( int * ) ) ;
    for ( i = 0 ; i < MAXROW ; i++ )
        p[i] = ( int * ) malloc ( MAXCOL * sizeof ( int ) ) ;
}
```

Q 17.15

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

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
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 ;
}
```

Q 17.16

Would the following code work at all times?

```
main( )
{
    char *ptr ;
    gets ( ptr ) ;
    printf ( "%s", ptr ) ;
}
```

Q 17.17

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

```
main( )
{
    char *s1 = "Cyber" ;
    char *s2 = "Punk" ;
    strcat ( s1, s2 ) ;
    printf ( "%s", s1 ) ;
}
```

Q 17.18

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

```
#include "alloc.h"
main( )
{
    int *p ;
    p = ( int * ) malloc ( 20 ) ;
    printf ( "%u", p ) ; /* suppose this prints 1314 */
    free ( p ) ;
    printf ( "%u", p ) ;
}
```


Q 17.19

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

```
#include "alloc.h"
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 ) ;
}
```

Q 17.20

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

Q 17.21

What would be the output of the following program?

```
#include "alloc.h"
main( )
{
    int *p ;
    p = ( int * ) malloc ( 20 ) ;
    printf ( "%d", sizeof ( p ) ) ;
    free ( p ) ;
}
```

Q *17.22*

Can I increase the size of a statically allocated array? <Yes/N> yes, how?

Q *17.23*

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

Q *17.24*

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

Q *17.25*

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?

Q *17.26*

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

Q *17.27*

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

Q *17.28*

malloc() allocates memory from the heap and not from the stack. <True/False>

Q *17.29*

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

Q *17.30*

What if we are to allocate 100 KB of memory?

Q *17.31*

Can we dynamically allocate arrays in expanded memory?

Q *17.32*

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

```
main()
{
    char *ptr ;
    *ptr = ( char * ) malloc ( 30 ) ;
    strcpy ( ptr, "RAM - Rarely Adequate Memory" ) ;
    printf ( "\n%s", ptr ) ;
    free ( ptr ) ;
}
```

Answers

A 17.1

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

A 17.2

```
main( )
{
    char *s ;
    char * fun( ) ;

    s = fun( ) ;
    printf ( "%s", s ) ;
}

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


A 17.3

```
#include "alloc.h"
main( )
{
    char *s ;
    char * fun( ) ;

    s = fun( ) ;
    printf ( "%s", s ) ;
    free ( s ) ;
}

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

A 17.4

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

A 17.5

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
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" ) ;
    }
}
```

A 17.6

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
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" ) ;
    }
}
```

A 17.7

```
#include "alloc.h"
#define MAXROW 3
#define MAXCOL 4
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" ) ;
    }
}
```

A 17.8

14 bytes

A 17.9

2 8

A 17.10

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] ) ;
```

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] ) ;
}
```

A 17.11

14 bytes

A 17.12

```
#include "alloc.h"
#define MAXX 3
#define MAXY 4
#define MAXZ 5
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" ) ;
    }
}
```

A 17.13

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

A 17.14

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

```
free ( p ) ;
```

A 17.15

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

A 17.16

No. Since *ptr* is an uninitialised pointer it must be pointing at some unknown location in memory. The string that we type in would get stored at the location to which *ptr* is pointing thereby overwriting whatever is present at that location.

A 17.17

```
main( )
{
    char s1[25] = "Cyber" ;
    char *s2 = "Punk" ;
    strcat ( s1, s2 ) ;
    printf ( "%s", s1 ) ;
}
```

A 17.18

1314

A 17.19

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 have used:


```
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.

A 17.20

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.

A 17.21

2

A 17.22

No

A 17.23

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

```
#include "alloc.h"
main( )
{
    int *p ;
    p = ( int * ) malloc ( 20 ) ;
    t = p ;
    t = ( int * ) realloc ( p, 40 ) ;
    if ( t == NULL )
```

```
        printf ( "Cannot reallocate, leaves previous allocated region unchanged" );
    else
    {
        if ( p == t )
            ; /* the array expanded at the same region */
        else
        {
            free ( p ) ; /* deallocate the original array */
            p = t ; /* set p to newly allocated region */
        }
    }
}
```

A 17.24

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

A 17.25

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.

A 17.26

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 ) ) ;
```


would allocate space for a 10-integer array. Additionally, *calloc()* would 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 ) ) ;
```

A 17.27

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

A 17.28

True

A 17.29

64 KB

A 17.30

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

A 17.31

Yes. Using interrupt 0x67, the details of which are beyond the scope of this book.

A *17.32*

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

Chapter 18

Variable Number of Arguments

Q *18.1*

Which header file should you include if you are to develop a function which can accept variable number of arguments?

A. vararg.h
B. stdlib.h
C. stdio.h
D. stdarg.h

Q *18.2*

What would be the output of the following program?

```
#include <stdio.h>
#include <stdarg.h>
main( )
{
    fun ( "Nothing specific", 1, 4, 7, 11, 0 ) ;
}

fun ( char *msg, ... )
{
    int tot = 0 ;
    va_list ptr ;
```

```
    int num ;

    va_start ( ptr, msg ) ;
    num = va_arg ( ptr, int ) ;
    num = va_arg ( ptr, int ) ;
    printf ( "\n%d", num ) ;
}
```

Q 18.3

Is it necessary that in a function which accepts variable argument list there should at least be one fixed argument? <Yes/No>

Q 18.4

Can the fixed arguments passed to the function which accepts variable argument list occur at the end? <Yes/No>

Q 18.5

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

```
#include <stdarg.h>
main( )
{
    varfun ( 3, 7, -11, 0 ) ;
}

varfun ( int n, ... )
{
    va_list ptr ;
    int num ;

    num = va_arg ( ptr, int ) ;
```



```
    printf ( "\n%d", num ) ;
}
```

Q 18.6

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

```
#include <stdarg.h>
main( )
{
    varfun ( 3, 7.5, -11.2, 0.66 ) ;
}

varfun ( int n, ... )
{
    float *ptr ;
    int num ;

    va_start ( ptr, n ) ;
    num = va_arg ( ptr, int ) ;
    printf ( "\n%d", num ) ;
}
```

Q 18.7

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

```
#include <stdarg.h>
main( )
{
    fun ( 1, 4, 7, 11, 0 ) ;
}

fun ( ... )
{
```

```
    va_list ptr ;
    int num ;

    va_start ( ptr, int ) ;
    num = va_arg ( ptr, int ) ;
    printf ( "%d", num ) ;
}
```

Q 18.8

The macro *va_start* is used to initialise a pointer to the beginning of the list of fixed arguments. <True/False>

Q 18.9

The macro *va_arg* is used to extract an argument from the variable argument list and advance the pointer to the next argument. <True/False>

Q 18.10

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

```
#include "stdarg.h"
main( )
{
    display ( 4, 'A', 'a', 'b', 'c' ) ;
}

display ( int num, ... )
{
    char c ; int j ;
    va_list ptr ;
```



```
    va_start ( ptr, num ) ;

    for ( j = 1 ; j <= num ; j++ )
    {
        c = va_arg ( ptr, char ) ;
        printf ( "%c ", c ) ;
    }
}
```

Q 18.11

Mention any variable argument list funciton that you have used and its prototype.

Q 18.12

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

```
#include "stdarg.h"
main( )
{
    display ( 4, 12.5, 13.5, 14.5, 44.3 ) ;
}

display ( int num, ... )
{
    float c ; int j ;
    va_list ptr ;

    va_start ( ptr, num ) ;

    for ( j = 1 ; j <= num ; j++ )
    {
        c = va_arg ( ptr, float ) ;
        printf ( "\n%f ", c ) ;
```

```
    }
}
```

Q 18.13

Point out the error in the following program.

```
#include "stdarg.h"
main( )
{
    display ( "Hello", 4, 2, 12.5, 13.5, 14.5, 44.0 ) ;
}

display ( char *s, int num1, int num2, ... )
{
    float c ; int j ;
    va_list ptr ;

    va_start ( ptr, s ) ;
    c = va_arg ( ptr, double ) ;
    printf ( "\n%f ", c ) ;
}
```

Q 18.14

Can we pass a variable argument list to a function at run-time? <Yes/No>

Q 18.15

While defining a variable argument list function can we drop the ellipsis. (...) <Yes/No>

Q 18.16

Can we write a function that takes a variable argument list and passes the list to another function (which takes a variable number of arguments) <Yes/No>

Q 18.17

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

```
#include "stdarg.h"
main( )
{
    display ( "Hello", 4, 12, 13, 14, 44 ) ;
}

display ( char *s, ... )
{
    show ( s, ... ) ;
}

show ( char *t, ... )
{
    va_list ptr ;
    int a ;
    va_start ( ptr, t ) ;
    a = va_arg ( ptr, int ) ;
    printf ( "%f", a ) ;
}
```

Q 18.18

How would you rewrite program 18.17 such that *show()* is able to handle variable argument list?

Q 18.19

If I use the following *printf()* to print a *long int* why I am not warned about the type mismatch?

```
printf ( "%d", num ) ;
```

Q 18.20

Can you write a function that works similar to *printf()*?

Q 18.21

How can a called function determine the number of arguments that have been passed to it?

Q 18.22

Can there be at least some solution to determine the number of arguments passed to a variable argument list function?

Q 18.23

Point out the error in the following program.

```
#include "stdarg.h"
main( )
{
    int ( *p1 )( ) ;
    int ( *p2 )( ) ;
    int fun1( ), fun2( ) ;
    p1 =fun1 ;
    p2 = fun2 ;
    display ( "Bye", p1, p2 ) ;
}

display ( char *s, ... )
{
    int ( *pp1 )( ) ;
    va_list ptr ;

    va_start ( ptr, s ) ;
    pp1 = va_arg ( ptr, int ( * )( ) ) ;
    ( *pp1 )( ) ;
    pp2 = va_arg ( ptr, int ( * )( ) ) ;
    ( *pp2 )( ) ;
}

fun1( )
{
    printf ( "Hello" ) ;
}

fun2( )
{
    printf ( "Hi" ) ;
}
```


Q 18.24

How would you rectify the error in the program 18.23 above?

Answers

A *18.1*

D

A *18.2*

4

A *18.3*

Yes

A *18.4*

No

A *18.5*

Since *ptr* has not been set at the beginning of the variable argument list using *va_start* it may be pointing anywhere and hence a call to *va_arg* would fetch a wrong integer.

A *18.6*

Irrespective of the type of argument passed to a function receiving variable argument list, *ptr* must be of the type *va_list*.

A *18.7*

There is no fixed argument in the definition of *fun()*.

A *18.8*

False

A *18.9*

True

A *18.10*

While extracting a *char* argument using *va_arg* we should have used *c = va_arg (ptr, int)*.

A *18.11*

printf (const char *format, ...) ;

A *18.12*

While extracting a *float* argument using *va_arg* we should have used *c = va_arg (ptr, double)*.

A *18.13*

While setting the *ptr* to the beginning of variable argument list we should have used *va_start (ptr, num2)*.

A 18.14

No. Every actual argument list must be completely known at compile time. In that sense it is not truely a variable argument list.

A 18.15

Yes

A 18.16

Yes

A 18.17

The call to *show()* is improper. This is not the way to pass variable argument list to a function.

A 18.18

```
#include "stdarg.h"
main( )
{
    display ( "Hello", 4, 12, 13, 14,44 ) ;
}

display ( char *s, ... )
{
    va_list ptr ;
    va_start ( ptr, s ) ;
    show ( s, ptr ) ;
}
```



```
show ( char *t, va_list ptr1 )
{
    int a, n, i ;
    a = va_arg ( ptr1, int ) ;
    for ( i = 0 ; i < a ; i++ )
    {
        n = va_arg ( ptr1, int ) ;
        printf ( "\n%d", n ) ;
    }
}
```

A 18.19

When a function accepts a variable number of arguments, its prototype cannot provide any information about the number of arguments and type of those variable arguments. Hence the compiler cannot warn about the mismatches. The prorgammer must make sure that arguments match or must manually insert explicit typecasts.

A 18.20

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

main( )
{
    void myprintf ( char *, ... ) ;
    char * convert ( unsigned int, int ) ;
    int i = 65 ;
    char str[ ] = "Intranets are here to stay.." ;
    myprintf ( "\nMessage = %s %d %x", str, i, i ) ;
}

void myprintf ( char *fmt, ... )
{
    char *p ;
```

```
int i ;
unsigned u ;
char *s ;
va_list argp ;

va_start ( argp, fmt ) ;

p = fmt ;
for ( p = fmt ; *p != '\0' ; p++ )
{
    if ( *p != '%' )
    {
        putchar ( *p ) ;
        continue ;
    }
    p++ ;
    switch ( *p )
    {
    case 'c' :
                    i = va_arg( argp, int ) ;
                    putchar( i ) ;
                    break ;

    case 'd' :
                    i = va_arg ( argp, int ) ;
                    if( i < 0 )
                    {
                        i = -i ;
                        putchar ( '-' ) ;
                    }
                    puts ( convert ( i, 10 ) ) ;
                    break ;
    case 'o' :
                    u = va_arg ( argp, unsigned int ) ;
                    puts ( convert ( u, 8 ) ) ;
                    break;
```



```
    case 's' :
                    s = va_arg ( argp, char * ) ;
                    puts ( s ) ;
                    break ;
    case 'u' :
                    u = va_arg ( argp, unsigned int ) ;
                    puts ( convert ( u, 10 ) ) ;
                    break ;
    case 'x' :
                    u = va_arg ( argp, unsigned int ) ;
                    puts ( convert ( u, 16 ) ) ;
                    break ;
    case '%' :
                    putchar ( '%' ) ;
                    break ;
    }
}
    va_end ( argp ) ;
}

char * convert ( unsigned int num, int base )
{
    static char buff[33] ;
    char *ptr ;

    ptr = &buff [ sizeof ( buff ) - 1 ] ;
    *ptr = '\0' ;
    do
    {
        *--ptr = "0123456789abcdef"[ num % base ] ;
        num /= base ;
    } while ( num != 0 ) ;

    return ptr ;
}
```

A 18.21

It cannot. Any function that takes a variable number of arguments must be able to determine from the arguments themselves how many of them there are. The printf() function, for example, does this by looking for format specifiers (%d etc.) in the format string. This is the reason why such functions fail badly if the format string does not match the argument list.

A 18.22

When the arguments passed are all of same type we can think of passing a sentinel value like -1 or 0 or a NULL pointer at the end of the variable argument list. Another way could be to explicitly pass the count of number of variable arguments.

A 18.23

va_arg cannot extract the function pointer from the variable argument list in the manner tried here.

A 18.24

Use *typedef* as shown below:

```
#include "stdarg.h"
main( )
{
    int ( *p1 )( ) ;
    int ( *p2 )( ) ;
    int fun1( ), fun2( ) ;

    p1 = fun1 ;
```



```
    p2 = fun2 ;
    display ( "Bye", p1, p2 ) ;
}

display ( char *s, ... )
{
    int ( *pp1 )( ), ( *pp2 )( ) ;
    va_list ptr ;
    typedef int ( *funcptr )( ) ;
    va_start ( ptr, s ) ;
    pp1 = va_arg ( ptr, funcptr ) ;
    ( *pp1 )( ) ;
    pp2 = va_arg ( ptr, funcptr ) ;
    ( *pp2 )( ) ;
}

fun1( )
{
    printf ( "\nHello" ) ;
}

fun2( )
{
    printf ( "\nHi" ) ;
}
```

Chapter 19

Complicated Declarations

Q 19.1

What do the following declarations signify?

A. `int *f( ) ;`
B. `int ( *pf)( ) ;`
C. `char **argv ;`
D. `void ( *f[10] ) ( int, int ) ;`
E. `char far *scr ;`
F. `char far *arr[10] ;`
G. `int ( *a[5] )( int far *p ) ;`
H. `char far *scr1, *scr2 ;`
I. `int ( *ftable[ ] )( void ) = { fadd, fsub, fmul, fdiv } ;`
J. `int ( *ptr )[30] ;`
K. `int *ptr[30] ;`
L. `void *cmp( ) ;`
M. `void ( *cmp )( ) ;`
N. `char ( * ( *f( ) ) [ ] ) ( )`
O. `char ( * ( *x[3] )( ) )[5] ;`
P. `void ( *f)( int, void ( * )( ) ) ;`
Q. `int ** ( *f)( int **, int ** ( * ) ( int **, int ** ) ) ;`
R. `void ( *f)( void ( * )( int *, void **), int ( * )(void **, int *)) ;`
S. `char far * far *ptr ;`
T. `char far * near *ptr ;`
U. `char far * huge *ptr ;`

Q 19.2

What would be the output of the following program?

```
main( )
{
    char near * near *ptr1 ;
    char near * far  *ptr2 ;
    char near * huge *ptr3 ;
    printf ( "%d %d %d", sizeof ( ptr1 ), sizeof ( ptr2 ), sizeof ( ptr3 ) ) ;
}
```

Q 19.3

What would be the output of the following program?

```
main( )
{
    char far * near *ptr1 ;
    char far * far  *ptr2 ;
    char far * huge *ptr3 ;
    printf ( "%d %d %d", sizeof ( ptr1 ), sizeof ( ptr2 ), sizeof ( ptr3 ) ) ;
}
```

Q 19.4

What would be the output of the following program?

```
main( )
{
    char huge * near *ptr1 ;
    char huge * far  *ptr2 ;
    char huge * huge *ptr3 ;
    printf ( "%d %d %d", sizeof ( ptr1 ), sizeof ( ptr2 ), sizeof ( ptr3 ) ) ;
}
```


Q 19.5

What would be the output of the following program?

```
main( )
{
    char huge * near * far *ptr1 ;
    char near * far  * huge *ptr2 ;
    char far  * huge * near *ptr3 ;
    printf ( "%d %d %d", sizeof ( ptr1 ), sizeof ( ptr2 ), sizeof ( ptr3 ) ) ;
}
```

Q 19.6

What would be the output of the following program?

```
main( )
{
    char huge * near * far  *ptr1 ;
    char near* far  * huge *ptr2 ;
    char far  * huge * near *ptr3 ;
    printf ( "%d %d %d", sizeof ( ptr1 ), sizeof ( *ptr2 ), sizeof ( **ptr3 ) ) ;
}
```

Q 19.7

What would be the output of the following program?

```
main( )
{
    char huge * near * far  *ptr1 ;
    char near * far  * huge *ptr2 ;
```

```
    char far * huge * near *ptr3 ;
    printf ( "%d %d %d", sizeof ( *ptr1 ), sizeof ( **ptr2 ), sizeof ( ptr3 ) ) ;
}
```

Q 19.8

What would be the output of the following program?

```
main ( )
{
    char huge * near * far  *ptr1 ;
    char near * far  * huge *ptr2 ;
    char far  * huge * near *ptr3 ;
    printf ( "%d %d %d", sizeof ( **ptr1 ), sizeof ( ptr2 ), sizeof ( *ptr3 ) ) ;
}
```

Q 19.9

Are the following two declarations same? <Yes/No>

```
char far * far *scr ;
char far far ** scr ;
```

Q 19.10

How would you declare the following:

- An array of three pointers to *chars*.
- An array of three *char* pointers.
- A pointer to an array of three chars.
- A pointer to a function which receives an *int* pointer and returns a *float* pointer.
- A pointer to a function which recevies nothing and returns nothing.

Q 19.11

Can you write a program which would implement the following declaration.

```
void ( *f )( int, void ( * )( ) ) ;
```

Answers:

A 19.1 A.

f is a function returning pointer to an *int*.

A 19.1 B.

pf is a pointer to function which returns an *int*.

A 19.1 C.

argv is a pointer to a *char* pointer.

A 19.1 D.

f is an array of 10 function pointers, where each function receives two *int*s and returns nothing.

A 19.1 E.

scr is a *far* pointer to a *char*. (*far* pointer is a pointer which contains an address which lies outside the data segment).

A *19.1 F.*

arr is an array of 10 *far* character pointers.

A *19.1 G.*

a is an array of 5 function pointers. Each of these functions receive a *far* pointer to an *int* and returns an *int*.

A *19.1 H.*

scr1 is a *far* pointer to a *char*, whereas *scr2* is a *near* pointer to a *char*.

A *19.1 I.*

ftable is an array of 4 function pointers which point to the functions *fadd( ),fsub( )* etc. Each of these functions accept nothing and return an *int*.

A *19.1 J.*

ptr is a pointer to an array of 30 integers.

A *19.1 K.*

ptr is an array of 30 pointers to integers.

A *19.1 L.*

cmp is a function returning pointer to *void*.

A *19.1 M.*

cmp is a pointer to function which returns a *void*.

A *19.1 N.*

f is a function returning pointer to *array[]* of pointer to function returning *char*.

A *19.1 O.*

x is an array of 3 pointers to functions returning pointer to an array of 5 *chars*.

A *19.1 P.*

f is a pointer to a function which returns nothing and receives as its parameter an integer and a pointer to a function which receives nothing and returns nothing.

A *19.1 Q.*

f is a pointer to a function which returns a pointer to an *int* pointer and receives two arguments: a pointer to an *int* pointer and a function pointer which points to a function which receives two pointers to *int* pointers and returns a pointer to an *int* pointer.

A *19.1 R.*

f is a pointer to a function which returns nothing and receives two arguments, both function pointers: the first function pointer points to

a function which returns nothing but receives two arguments - an *int* pointer and a pointer to a *void* pointer; the second function pointer points to a function which returns an *int* pointer and receives a pointer to a *void* pointer and an *int* pointer.

A *19.1 S.*

ptr is a *far* pointer to a *far* pointer to a *char*, or in easier words, *ptr* contains a *far* address of a *far* pointer to a *char*.

A *19.1 T.*

ptr is a *near* pointer to a *far* pointer to a *char*, or in easier words, *ptr* contains a *near* address of a *far* pointer to a *char*.

A *19.1 U.*

ptr is a *huge* pointer to a *far* pointer to a *char*, or in easier words, *ptr* contains a *huge* address of a *far* pointer to a *char*.

A *19.2*

244

A *19.3*

244

A *19.4*

244

A *19.5*

442

A *19.6*

444

A *19.7*

222

A *19.8*

444

A *19.9*

No

A *19.10*

```
char *ptr[3];
char *ptr[3];
char ( *ptr )[3];
float * ( *ptr )( int * );
void ( *ptr )( );
```

A *19.11*

```
main( )
{
```

```
    void ( *f )( int, void ( * )( ) ) ;
    void fun ( int, void ( * )( ) ) ;
    void fun1( ) ;
    void ( *p )( ) ;
    f = fun ;
    p = fun1 ;
    ( *f )( 23, p ) ;
}
void fun ( int i, void ( *q )( ) )
{
    printf ( "Hello" ) ;
}
void fun1( )
{
    ;
}
```

Chapter 20

Library Functions

Q *20.1*

What do the functions *atoi()*, *itoa()* and *gcvt()* do? Show how would you use them in a program.

Q *20.2*

Does there exist any other function which can be used to convert an integer or a float to a string? If yes, show how you would use it.

Q *20.3*

How would you use *qsort()* function to sort an array of structures?

Q *20.4*

How would you use *qsort()* function to sort the names stored in an array of pointers to strings?

Q 20.5

How would you use *bsearch()* function to search a name stored in an array of pointers to strings?

Q 20.6

How would you use the functions *sin(), pow(), sqrt()*?

Q 20.7

How would you use the function *memcpy()*?

Q 20.8

How would you use the function *memset()*?

Q 20.9

How would you use the function *memmove()*?

Q 20.10

How would you use the functions *fseek(), fread(), fwrite()* and *ftell()*?

Q 20.11

How would you obtain the current time and difference between two times?

Q 20.12

How would you use the function *randomize()* and *random()*?

Q 20.13

Would the following program always output 'Banglore'?

```
main( )
{
    char str1[ ] = "Banglore - 440010" ;
    char str2[10] ;

    strncpy ( str2, str1, 8 ) ;
    printf ( "\n%s", str2 ) ;
}
```

Q 20.14

Can you shorten this code?

```
main( )
{
    char str1[ ] = "Banglore - 440010" ;
    char str2[10] ;

    str2[0] = '\0' ;
    strncat ( str2, str1, 8 ) ;
    printf ( "%s", str2 ) ;
}
```

Q 20.15

How would you implement a *substr()* function that extracts a substring from a given string?

Q 20.16

Given a sentence containing words separated by spaces how would you construct an array of pointers to strings containing addresses of each word in the sentence?

Q 20.17

Write the comparison function *qs_compare()* for the following code.

```
struct date
{
    int d, m, y ;
} ;
qs_compare ( const void *, const void * ) ;

main( )
{

    struct date dd[ ] = {
                          { 17, 11, 62 },
                          { 24, 8,  78 },
                          { 17, 11, 62 },
                          { 16, 12, 76 },
                          { 19,  2, 94 }
                        } ;
    int i, w ;
    clrscr( ) ;
```



```
    w = sizeof ( struct date ) ;
    qsort ( dd, 5, w, qs_compare ) ;

    for ( i = 0 ; i < 4 ; i++ )
        printf ( "\n%d %d %d", dd[i].d, dd[i].m, dd[i].y ) ;
}
```

Q 20.18

How should we sort a linked list?

Q 20.19

What's the difference between the functions *rand()*, *random()*, *srand()* and *randomize()*?

Q 20.20

What's the difference between the function *memmove()* and *memcpy()*?

Q 20.21

How would you print a string on the printer?

Q 20.22

Can you use the function *fprintf()* to display the output on the screen?

Answers

A 20.1

atoi() Converts a string to an integer.
itoa() Converts an integer to a string.
gcvt() Converts a floating-point number to a string.

```
#include "stdlib.h"
main()
{
    char s[ ] = "12345" ;
    char buffer[15], string[20] ;
    int i ;

    i = atoi ( s ) ;
    printf ( "\n%d", i ) ;

    gcvt ( 20.141672, 4, buffer ) ;
    printf ( "\n%s", buffer ) ;

    itoa ( 15, string, 2 ) ;
    printf ( "\n%s",string ) ;
}
```

A 20.2

The function *sprintf()* can be used for this purpose. This function also has the ability to format the numbers as they are converted to strings. The following program shows how to use this function.

```
#include "stdio.h"
main( )
{
    int a = 25 ;
```



```
    float b = 3.14 ;
    char str[40] ;

    sprintf ( str, "a = %d b = %f", a, b ) ;
    puts ( str ) ;
}
```

A 20.3

```
#include "string.h"
#include "stdlib.h"

struct stud
{
    int rollno ;
    int marks ;
    char name[30] ;
} ;
int sort_rn ( struct stud *, struct stud * ) ;
int sort_name ( struct stud *, struct stud * ) ;
int sort_marks ( struct stud *, struct stud * ) ;

main( )
{
    static struct stud ss[ ] = {
                                { 15, 96, "Akshay" },
                                { 2, 97, "Madhuri" },
                                { 8, 85, "Aishvarya" },
                                { 10, 80, "Sushmita" }
                              } ;
    int x, w ;

    clrscr( ) ;
    w = sizeof ( struct stud ) ;

    printf ( "\nIn order of roll numbers: " ) ;
    qsort ( ss, 4, w, sort_rn ) ;
```

```
    for ( x = 0 ; x < 4 ; x++ )
        printf ( "\n%d %s %d", ss[x].rollno, ss[x].name, ss[x].marks ) ;

    printf ( "\n\nIn order of names: " ) ;
    qsort ( ss, 4, w, sort_name ) ;

    for ( x = 0 ; x < 4 ; x++ )
        printf ( "\n%d %s %d", ss[x].rollno, ss[x].name, ss[x].marks ) ;
    printf ( "\n\nIn order of marks: " ) ;
    qsort ( ss, 4, w, sort_marks ) ;

    for ( x = 0 ; x < 4 ; x++ )
        printf ( "\n%d %s %d", ss[x].rollno, ss[x].name, ss[x].marks ) ;
}

int sort_m ( struct stud *t1, struct stud *t2 )
{
    return ( t1->rollno - t2->rollno ) ;
}

int sort_name ( struct stud *t1, struct stud *t2 )
{
    return ( strcmp ( t1->name, t2->name ) ) ;
}

int sort_marks ( struct stud *t1, struct stud *t2 )
{
    return ( t2->marks - t1->marks ) ;
}
```

A 20.4

```
#include "string.h"
#include "stdlib.h"

int sort_name ( const void *, const void * ) ;
```



```
main( )
{
    char *names[ ] = {
                        "Akshay",
                        "Madhuri",
                        "Aishvarya",
                        "Sushmita",
                        "Sudeepta"
                    } ;

    int i ;

    qsort ( names, 5, sizeof ( char * ), sort_name ) ;
    for ( i = 0 ; i < 5 ; i++ )
        printf ( "\n%s", names[i] ) ;
}

int sort_name ( const void *t1, const void *t2 )
{
    /* t1 and t2 are always pointers to objects being compared */

    char **t11, **t22 ;

    /* cast appropriately */
    t11 = ( char ** ) t1 ;
    t22 = ( char ** ) t2 ;
    return ( strcmp ( *t11, *t22 ) ) ;
}
```

A 20.5

```
#include "string.h"
#include "stdlib.h"

int sort_name ( const void *, const void * ) ;
int bs_compare ( char**, char ** ) ;
main( )
```

```
{
    char *names[ ] = {
                        "Akshay",
                        "Madhuri",
                        "Aishvarya",
                        "Sushmita",
                        "Sudeepta"
                    } ;

    int i, wide, nel ;
    char *s = "aMadhuri", **b ;

    qsort ( names, 5, sizeof ( char * ), sort_name ) ;
    clrscr( ) ;
    for ( i = 0 ; i < 5 ; i++ )
        printf ( "\n%s", names[i] ) ;

    wide = sizeof ( names[0] ) ;
    nel  = sizeof ( names ) / wide ;
    b = bsearch ( &s, names, nel, wide, bs_compare ) ;

    if ( b == NULL)
        printf ( "Not found" ) ;
    else
        printf ( "\n\n%s", *b ) ;

}

int sort_name ( const void *t1, const void *t2 )
{
    /* t1 and t2 are always pointers to objects being compared */

    char **t11, **t22 ;

    /* cast appropriately */
    t11 = ( char** ) t1 ;
    t22 = ( char** ) t2 ;
```



```
    return ( strcmp ( *t11, *t22 ) ) ;
}

int bs_compare ( char **s1, char **s2 )
{
   return ( strcmp ( *s1, *s2 ) ) ;
}
```

A 20.6

```
#include "math.h"
main( )
{
    int ang ;
    float angrad, x, y, a, b ;

    printf ( "\nEnter the angle in degrees" ) ;
    scanf ( "%d", &ang ) ;

    angrad = ang * 3.14 / 180 ;
    x = sin ( angrad ) ;
    a = pow ( x, 2 ) ;
    b = 1 - a ;
    y = sqrt ( b ) ;

    printf ( "\ncosine of angle %d is = %f", ang, y ) ;
}
```

A 20.7

```
#include "mem.h"
#include "alloc.h"

main( )
{
    int area ;
    char src[ ] = "Pray, not for lighter load, but for stronger back" ;
```

```
    char *dest ;

    area = sizeof ( src ) ;
    dest = malloc ( area ) ;
    memcpy ( dest, src, area ) ;

    printf ( "\n%s", src ) ;
    printf ( "\n%s", dest ) ;
}
```

A 20.8

```
#include "mem.h"
main( )
{
    int  area ;
    char src[ ] = "Bon jour, Madam" ;

    area = sizeof ( src ) ;
    memset ( src, '!', area - 7 ) ;

    printf ( "\n%s", src ) ;
}
```

A 20.9

```
#include "mem.h"
#include "alloc.h"

main( )
{
    int area ;
    char *dest ;
    char src[ ] = "Life is the camera and you are the target "
                  "so keep smiling always" ;

    area = sizeof ( src ) ;
```



```
    dest = malloc ( area ) ;
    memmove ( dest, src, area ) ;

    printf ( "\n%s", dest ) ;
    printf ( "\n%s", src ) ;
}
```

A 20.10

```
#include "stdio.h"

struct stud
{
    int rollno ;
    char name[10] ;
    float per ;
} e ;

FILE *fs ;

main( )
{
    long position = 0L ;
    int rollno;
    charch;
    float temp ;

    fs = fopen ( "stud.dat", "rb+" ) ;

    if ( fs == NULL)
    {
        puts ( "Unable to open file" ) ;
        exit ( 1 ) ;
    }

    do
    {
```

```
        printf ( "\nEnter code no. to modify: " ) ;
        scanf ( "%d", &rollno ) ;

        while ( fread ( &e, sizeof ( e ) , 1, fs ) == 1 )
        {
            if ( e.rollno == rollno )
            {
                    printf ( "\nEnter the new record" ) ;
                    scanf ( "%s %f", e.name, &temp ) ;
                    e.per = temp ;
                    fseek ( fs, position, SEEK_SET ) ;
                    fwrite ( &e, sizeof ( e ) , 1, fs ) ;
                    break ;
            }

            position = ftell ( fs ) ;
        }

        puts ( "You want to modify records" ) ;
        ch = getche( ) ;
    } while ( ch == 'Y' ) ;
}
```

A 20.11

```
#include "time.h"
#include "dos.h"

main( )
{
    time_t t1, t2 ;
    double diff, f ;
    int i = 2 ;

    time ( &t1 ) ;
    sleep ( i ) ;
    time ( &t2 ) ;
```



```
    diff = difftime ( t2, t1 ) ;

    printf ( "\nProgram was active for %lf seconds", diff ) ;
}
```

A 20.12

```
#include "dos.h"
#include "stdlib.h"

main( )
{
    randomize( ) ;

    printf ( "\nPress any key to stop." ) ;
    while ( !kbhit( ) )
    {
        sound ( random ( 500 ) ) ;
        delay ( random ( 100 ) ) ;
        nosound( ) ;
    }
}
```

A 20.13

No. Because after copying the source string into the target string *strncpy()* doesn't terminate the target string with a '\0'. A better way of copying would be:

```
str2[0] = '\0' ;
strncat ( str2, str1, 8 ) ;
```

strncat() always terminates the target string with a '\0'.

A 20.14

Yes, using *sprintf()* as shown below:

```
main()
{
    char str1[] = "Banglore - 440010" ;
    char str2[10] ;

    sprintf ( str2, "%.*s", 8, str1 ) ;
    printf ( "%s", str2 ) ;
}
```

A 20.15

```
main( )
{
    char str1[ ] = "Banglore" ;
    char str2[5] ;

    /* extract 3 characters beginning with first character */
    substr ( str2, str1, 1, 3 ) ;
    printf ( "%s", str2 ) ;
}

substr ( char *t, char *s, int pos, int len )
{
    t[0] = '\0' ;
    strncat ( t, s + pos, len ) ;
}
```

A 20.16

```
#include "stdio.h"
#include "string.h"
main( )
```



```
{
    char str[ ] = "This is a test" ;
    char *ptr[10] ;
    char *p ;
    int i = 1, j ;

    p = strtok ( str, " " ) ;
    if ( p != NULL )
    {
        ptr[0] = p ;
        while ( 1 )
        {
            p = strtok ( NULL, " " ) ;
            if ( p == NULL )
                break ;
            else
            {
                ptr[i] = p ;
                i++ ;
            }
        }
    }

    for ( j = 0 ; j < i ; j++ )
        printf ( "\n%s", ptr[j] ) ;
}
```

A 20.17

```
int qs_compare ( const void *p1, const void *p2 )
{
    const struct date *sp1 = p1 ;
    const struct date *sp2 = p2 ;

    if ( sp1->y < sp2->y )
        return ( -1 ) ;
    else if ( sp1->y > sp2->y )
```

```
        return ( 1 ) ;
    else if ( sp1->m < sp2->m )
        return ( -1 ) ;
    else if ( sp1->m > sp2->m )
        return ( 1 ) ;
    else if ( sp1->d < sp2->d )
        return ( -1 ) ;
    else if ( sp1->d > sp2->d )
        return ( 1 ) ;
    else
        return ( 0 ) ;
}
```

A 20.18

Often it's easier to keep the list in order as we build it rather than sorting it later. Still if we want to sort the list then we can allocate a temporary array of pointers, fill it with pointers to the various nodes in the list, call *qsort()* and finally rebuild the list pointers based on the sorted array.

A 20.19

rand() returns a random number.
random() returns a random number in a specified range.
srand() initialises a random number generator with a given seed value.
randomize() initialises a random number generator with a random value based on time.

A 20.20

Both the functions copy a block of bytes from source to destination. However, if source and destination overlaps the behaviour of

memcpy() is undefined whereas *memmove()* carries out the copying correctly. *memcpy()* is more efficient, whereas *memmove()* is safer to use.

A 20.21

```
#include "stdio.h"
main( )
{
    char str[ ] = "There wouldn't have been COBOL without C" ;
    fprintf ( stdprn, "\n%s\n", str ) ;
}
```

A 20.22

Yes, by replacing the usual file pointer with *stdout* as shown below:

```
fprintf ( stdout, "%s %d %f", str, i, a ) ;
```