C Questions

Note : All the programs are tested under Turbo C/C++ compilers.
It is assumed that,

- Programs run under DOS environment,
- The underlying machine is an x86 system,
- Program is compiled using Turbo C/C++ compiler.

The program output may depend on the information based on this assumptions (for example sizeof(int) == 2 may be assumed).

Predict the output or error(s) for the following:

1. ```
   void main()
   {
       int  const * p=5;
       printf("%d",++(*p));
   }
   ```
   Answer:
   Compiler error: Cannot modify a constant value.
   Explanation:
   p is a pointer to a "constant integer". But we tried to change the value of the "constant integer".

2. ```
   main()
   {
       char s[ ]="man";
       int i;
       for(i=0;s[ i ];i++)
       printf("\n%c%c%c%c",s[ i ],*(s+i),*(i+s),i[s]);
   }
   ```
   Answer:
   mmmm
   aaaa
   nnnn
   Explanation:
   s[i], *(i+s), *(s+i), i[s] are all different ways of expressing the same idea. Generally array name is the base address for that array. Here **s** is the base address. **i** is the index number/displacement from the base address. So, indirecting it with * is same as s[i]. i[s] may be surprising. But in the case of C it is same as s[i].

3. ```
   main()
   {
       float me = 1.1;
       double you = 1.1;
       if(me==you)
       printf("I love U");
   ```
   ```

```
    else
        printf("I hate U");
}
```

Answer:

I hate U

Explanation:

For floating point numbers **(**float, double, long double**)** the values cannot be predicted exactly. Depending on the number of bytes, the precession with of the value represented varies. Float takes 4 bytes and long double takes 10 bytes. So float stores 0.9 with less precision than long double.

Rule of Thumb:

Never compare or at-least be cautious when using floating point numbers with relational operators **(**== , >, <, <=, >=,!= **)** .

4.
```
main()
{
static int var = 5;
printf("%d ",var--);
if(var)
    main();
}
```

Answer:

5 4 3 2 1

Explanation:

When *static* storage class is given, it is initialized once. The change in the value of a *static* variable is retained even between the function calls. Main is also treated like any other ordinary function, which can be called recursively.

5.
```
main()
{
 int c[ ]={2.8,3.4,4,6.7,5};
 int j,*p=c,*q=c;
 for(j=0;j<5;j++) {
        printf(" %d ",*c);
        ++q;    }
 for(j=0;j<5;j++){
printf(" %d ",*p);
++p;    }
}
```

Answer:

2 2 2 2 2 2 3 4 6 5

Explanation:

Initially pointer c is assigned to both **p** and **q**. In the first loop, since only **q** is incremented and not c , the value 2 will be printed 5 times. In second loop **p** itself is incremented. So the values 2 3 4 6 5 will be printed.

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

Answer:

Linker Error : Undefined symbol '_i'

Explanation:

extern storage class in the following declaration,

extern int i;

specifies to the compiler that the memory for **i** is allocated in some other program and that address will be given to the current program at the time of linking. But linker finds that no other variable of name **i** is available in any other program with memory space allocated for it. Hence a linker error has occurred .

7. *main()*
```
{
    int i=-1,j=-1,k=0,l=2,m;
    m=i++&&j++&&k++||l++;
    printf("%d %d %d %d %d",i,j,k,l,m);
}
```

Answer:

0 0 1 3 1

Explanation :

Logical operations always give a result of **1 or 0** . And also the logical AND (&&) operator has higher priority over the logical OR (||) operator. So the expression '**i++ && j++ && k++'** is executed first. The result of this expression is 0 (-1 && -1 && 0 = 0). Now the expression is 0 || 2 which evaluates to 1 (because OR operator always gives 1 except for '0 || 0' combination- for which it gives 0). So the value of m is 1. The values of other variables are also incremented by 1.

8. *main()*
```
{
    char *p;
    printf("%d %d ",sizeof(*p),sizeof(p));
}
```

Answer:

1 2

Explanation:

The sizeof() operator gives the number of bytes taken by its operand. P is a character pointer, which needs one byte for storing its value (a character). Hence sizeof(*p) gives a value of 1. Since it needs two bytes to store the address of the character pointer sizeof(p) gives 2.

9.
```
main()
{
    int i=3;
    switch(i)
     {
       default:printf("zero");
       case 1: printf("one");
                break;
      case 2:printf("two");
               break;
     case 3: printf("three");
               break;
     }
}
```

Answer :

three

Explanation :

The default case can be placed anywhere inside the loop. It is executed only when all other cases doesn't match.

10.
```
main()
{
     printf("%x",-1<<4);
}
```

Answer:

fff0

Explanation :

-1 is internally represented as all 1's. When left shifted four times the least significant 4 bits are filled with 0's.The %x format specifier specifies that the integer value be printed as a hexadecimal value.

11.
```
main()
{
    char string[]="Hello World";
    display(string);
}
void display(char *string)
{
    printf("%s",string);
}
```

Answer:

Compiler Error : Type mismatch in redeclaration of function display

Explanation :

In third line, when the function **display** is encountered, the compiler doesn't know anything about the function display. It assumes the arguments and

return types to be integers, (which is the default type). When it sees the actual function **display,** the arguments and type contradicts with what it has assumed previously. Hence a compile time error occurs.

12. main()

```
{
    int c=- -2;
    printf("c=%d",c);
}
```

Answer:

c=2;

Explanation:

Here unary minus (or negation) operator is used twice. Same maths rules applies, ie. minus * minus= plus.

Note:

However you cannot give like --2. Because -- operator can only be applied to variables as a **decrement** operator (eg., i--). 2 is a constant and not a variable.

13. #define int char

```
main()
{
    int i=65;
    printf("sizeof(i)=%d",sizeof(i));
}
```

Answer:

sizeof(i)=1

Explanation:

Since the #define replaces the string **int** by the macro **char**

14. main()

```
{
    int i=10;
    i=!i>14;
    Printf ("i=%d",i);
}
```

Answer:

i=0

Explanation:

In the expression **!i>14** , NOT (!) operator has more precedence than ‘ >’ symbol. **!** is a unary logical operator. !i (!10) is 0 (not of true is false). 0>14 is false (zero).

15. #include<stdio.h>

```
main()
{
    char s[]={'a','b','c','\n','c','\0'};
    char *p,*str,*str1;
    p=&s[3];
    str=p;
    str1=s;
    printf("%d",++*p + ++*str1-32);
}
```

Answer:

77

Explanation:

p is pointing to character '\n'. str1 is pointing to character 'a' ++*p. "p is pointing to '\n' and that is incremented by one." the ASCII value of '\n' is 10, which is then incremented to 11. The value of ++*p is 11. ++*str1, str1 is pointing to 'a' that is incremented by 1 and it becomes 'b'. ASCII value of 'b' is 98.

Now performing (11 + 98 – 32), we get 77("M");

So we get the output 77 :: "M" (Ascii is 77).

16.
```
#include<stdio.h>
main()
{
    int a[2][2][2] = { {10,2,3,4}, {5,6,7,8}  };
    int *p,*q;
    p=&a[2][2][2];
    *q=***a;
    printf("%d----%d",*p,*q);
}
```

Answer:

SomeGarbageValue---1

Explanation:

p=&a[2][2][2] you declare only two 2D arrays, but you are trying to access the third 2D(which you are not declared) it will print garbage values. *q=***a starting address of a is assigned integer pointer. Now q is pointing to starting address of a. If you print *q, it will print first element of 3D array.

17.
```
#include<stdio.h>
main()
{
    struct xx
    {
        int x=3;
        char name[]="hello";
     };
    struct xx *s;
    printf("%d",s->x);
```

```
    printf("%s",s->name);
}
```

Answer:

Compiler Error

Explanation:

You should not initialize variables in declaration

18.
```
#include<stdio.h>
main()
{
    struct xx
    {
int x;
        struct yy
        {
            char s;
            struct xx *p;
        };
        struct yy *q;
    };
}
```

Answer:

Compiler Error

Explanation:

The structure yy is nested within structure xx. Hence, the elements are of yy are to be accessed through the instance of structure xx, which needs an instance of yy to be known. If the instance is created after defining the structure the compiler will not know about the instance relative to xx. Hence for nested structure yy you have to declare member.

19.
```
main()
{
    printf("\nab");
    printf("\bsi");
    printf("\rha");
}
```

Answer:

hai

Explanation:

\n - newline
\b - backspace
\r - linefeed

20.
```
main()
{
    int i=5;
```

```
    printf("%d%d%d%d%d%d",i++,i--,++i,--i,i);
}
```

Answer:

45545

Explanation:

The arguments in a function call are pushed into the stack from left to right. The evaluation is by popping out from the stack. and the evaluation is from right to left, hence the result.

21.
```
#define square(x) x*x
main()
{
    int i;
    i = 64/square(4);
    printf("%d",i);
}
```

Answer:

64

Explanation:

the macro call square(4) will substituted by 4*4 so the expression becomes i = 64/4*4 . Since / and * has equal priority the expression will be evaluated as (64/4)*4 i.e. 16*4 = 64

22.
```
main()
{
char *p="hai friends",*p1;
p1=p;
while(*p!='\0') ++*p++;
printf("%s   %s",p,p1);
}
```

Answer:

ibj!gsjfoet

Explanation:

++*p++ will be parse in the given order

- *p that is value at the location currently pointed by p will be taken
- ++*p the retrieved value will be incremented
- when ; is encountered the location will be incremented that is p++ will be executed

Hence, in the while loop initial value pointed by p is ‘h’, which is changed to ‘i’ by executing ++*p and pointer moves to point, ‘a’ which is similarly changed to ‘b’ and so on. Similarly blank space is converted to ‘!’. Thus, we obtain value in p becomes “ibj!gsjfoet” and since p reaches ‘\0’ and p1 points to p thus p1doesnot print anything.

23.
```
#include <stdio.h>
#define a 10
main()
```

```
{
    #define a 50
    printf("%d",a);
}
```

Answer:

50

Explanation:

The preprocessor directives can be redefined anywhere in the program. So the most recently assigned value will be taken.

24.
```
#define clrscr() 100
main()
{
    clrscr();
    printf("%d\n",clrscr());
}
```

Answer:

100

Explanation:

Preprocessor executes as a seperate pass before the execution of the compiler. So textual replacement of clrscr() to 100 occurs.The input program to compiler looks like this :

```
main()
{
    100;
    printf("%d\n",100);
}
```

Note:

100; is an executable statement but with no action. So it doesn't give any problem

25.
```
main()
{
    41printf("%p",main);
}8
```
Answer:

Some address will be printed.

Explanation:

Function names are just addresses (just like array names are addresses).

main() is also a function. So the address of function main will be printed. %p in printf specifies that the argument is an address. They are printed as hexadecimal numbers.

27)
```
main()
{
clrscr();
}
```

```
clrscr();
```

Answer:
No output/error
Explanation:
The first clrscr() occurs inside a function. So it becomes a function call. In the second clrscr(); is a function declaration (because it is not inside any function).

28)
```
enum colors {BLACK,BLUE,GREEN}
 main()
{

 printf("%d..%d..%d",BLACK,BLUE,GREEN);

 return(1);
}
```
Answer:
0..1..2
Explanation:
enum assigns numbers starting from 0, if not explicitly defined.

29)
```
void main()
{
 char far *farther,*farthest;

 printf("%d..%d",sizeof(farther),sizeof(farthest));

 }
```
Answer:
4..2
Explanation:
the second pointer is of char type and not a far pointer

30)
```
main()
{
 int i=400,j=300;
 printf("%d..%d");
}
```
Answer:
400..300
Explanation:
printf takes the values of the first two assignments of the program. Any number of printf's may be given. All of them take only the first two values. If more number of assignments given in the program,then printf will take garbage values.

31)
```
main()
```

```
{
 char *p;
 p="Hello";
 printf("%c\n",*&*p);
}
```

Answer:

H

Explanation:

* is a dereference operator & is a reference operator. They can be applied any number of times provided it is meaningful. Here p points to the first character in the string "Hello". *p dereferences it and so its value is H. Again & references it to an address and * dereferences it to the value H.

32)
```
main()
{
   int i=1;
   while (i<=5)
   {
     printf("%d",i);
     if (i>2)
          goto here;
     i++;
   }
}
fun()
{
  here:
    printf("PP");
}
```

Answer:

Compiler error: Undefined label 'here' in function main

Explanation:

Labels have functions scope, in other words the scope of the labels is limited to functions. The label 'here' is available in function fun() Hence it is not visible in function main.

33)
```
main()
{
  static char names[5][20]={"pascal","ada","cobol","fortran","perl"};
   int i;
   char *t;
   t=names[3];
   names[3]=names[4];
   names[4]=t;
   for (i=0;i<=4;i++)
         printf("%s",names[i]);
```

```
}
```

Answer:

Compiler error: Lvalue required in function main

Explanation:

Array names are pointer constants. So it cannot be modified.

34)
```
void main()
{
        int i=5;
        printf("%d",i++ + ++i);
}
```

Answer:

Output Cannot be predicted exactly.

Explanation:

Side effects are involved in the evaluation of i

35)
```
void main()
{
        int i=5;
        printf("%d",i+++++i);
}
```

Answer:

Compiler Error

Explanation:

The expression i+++++i is parsed as i ++ ++ + i which is an illegal combination of operators.

36)
```
#include<stdio.h>
main()
{
int i=1,j=2;
switch(i)
 {
 case 1:  printf("GOOD");
            break;
 case j:  printf("BAD");
            break;
 }
}
```

Answer:

Compiler Error: Constant expression required in function main.

Explanation:

The case statement can have only constant expressions (this implies that we cannot use variable names directly so an error).

Note:

Enumerated types can be used in case statements.

37)
```
main()
{
int i;
printf("%d",scanf("%d",&i));  // value 10 is given as input here
}
```
Answer:

1

Explanation:

Scanf returns number of items successfully read and not 1/0. Here 10 is given as input which should have been scanned successfully. So number of items read is 1.

38)
```
#define f(g,g2) g##g2
main()
{
int var12=100;
printf("%d",f(var,12));
}
```
Answer:

100

39)
```
main()
{
int i=0;

for(;i++;printf("%d",i)) ;
printf("%d",i);
}
```
Answer:

1

Explanation:

before entering into the for loop the checking condition is "evaluated". Here it evaluates to 0 (false) and comes out of the loop, and i is incremented (note the semicolon after the for loop).

40)
```
#include<stdio.h>
main()
{
  char s[]={'a','b','c','\n','c','\0'};
  char *p,*str,*str1;
  p=&s[3];
  str=p;
  str1=s;
  printf("%d",++*p + ++*str1-32);
}
```
Answer:

M

Explanation:

p is pointing to character '\n'.str1 is pointing to character 'a' ++*p meAnswer:"p is pointing to '\n' and that is incremented by one." the ASCII value of '\n' is 10. then it is incremented to 11. the value of ++*p is 11. ++*str1 meAnswer:"str1 is pointing to 'a' that is incremented by 1 and it becomes 'b'. ASCII value of 'b' is 98. both 11 and 98 is added and result is subtracted from 32.

i.e. (11+98-32)=77("M");

41)
```
#include<stdio.h>
main()
{
 struct xx
  {
    int x=3;
    char name[]="hello";
  };
struct xx *s=malloc(sizeof(struct xx));
printf("%d",s->x);
printf("%s",s->name);
}
```

Answer:

Compiler Error

Explanation:

Initialization should not be done for structure members inside the structure declaration

42)
```
#include<stdio.h>
main()
{
struct xx
{
 int x;
 struct yy
  {
   char s;
   struct xx *p;
  };
        struct yy *q;
 };
}
```

Answer:

Compiler Error

Explanation:

in the end of nested structure yy a member have to be declared.

43)
```
main()
{
 extern int i;
 i=20;
 printf("%d",sizeof(i));
}
```
Answer:
Linker error: undefined symbol '_i'.
Explanation:
extern declaration specifies that the variable i is defined somewhere else. The compiler passes the external variable to be resolved by the linker. So compiler doesn't find an error. During linking the linker searches for the definition of i. Since it is not found the linker flags an error.

44)
```
main()
{
printf("%d", out);
}
int out=100;
```
Answer:
Compiler error: undefined symbol out in function main.
Explanation:
The rule is that a variable is available for use from the point of declaration. Even though a is a global variable, it is not available for main. Hence an error.

45)
```
main()
{
 extern out;
 printf("%d", out);
}
 int out=100;
```
Answer:
100
Explanation:
This is the correct way of writing the previous program.

46)
```
main()
{
 show();
}
void show()
{
 printf("I'm the greatest");
}
```
Answer:
Compier error: Type mismatch in redeclaration of show.

Explanation:

When the compiler sees the function show it doesn't know anything about it. So the default return type (ie, int) is assumed. But when compiler sees the actual definition of show mismatch occurs since it is declared as void. Hence the error.

The solutions are as follows:

1. declare void show() in main() .
2. define show() before main().
3. declare extern void show() before the use of show().

47)
```
main( )
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{{2,2},{2,3},{3,4}}};
 printf("%u %u %u %d \n",a,*a,**a,***a);
 printf("%u %u %u %d \n",a+1,*a+1,**a+1,***a+1);
}
```

Answer:

100, 100, 100, 2
114, 104, 102, 3

Explanation:

The given array is a 3-D one. It can also be viewed as a 1-D array.

2	4	7	8	3	4	2	2	2	3	3	4
100	102	104	106	108	110	112	114	116	118	120	122

thus, for the first printf statement a, *a, **a give address of first element . since the indirection ***a gives the value. Hence, the first line of the output.

for the second printf a+1 increases in the third dimension thus points to value at 114, *a+1 increments in second dimension thus points to 104, **a +1 increments the first dimension thus points to 102 and ***a+1 first gets the value at first location and then increments it by 1. Hence, the output.

48)
```
main( )
{
 int a[ ] = {10,20,30,40,50},j,*p;
for(j=0; j<5; j++)
 {
 printf("%d" ,*a);
 a++;
  }
  p = a;
 for(j=0; j<5; j++)
{
 printf("%d " ,*p);
 p++;
}
```

```
}
```

Answer:

Compiler error: lvalue required.

Explanation:

Error is in line with statement a++. The operand must be an lvalue and may be of any of scalar type for the any operator, array name only when subscripted is an lvalue. Simply array name is a non-modifiable lvalue.

**49)

```
main( )
{
 static int  a[ ]  = {0,1,2,3,4};
 int  *p[ ] = {a,a+1,a+2,a+3,a+4};
 int  **ptr =  p;
 ptr++;
 printf("\n %d  %d  %d", ptr-p, *ptr-a, **ptr);
 *ptr++;
 printf("\n %d  %d  %d", ptr-p, *ptr-a, **ptr);
 *++ptr;
printf("\n %d  %d  %d", ptr-p, *ptr-a, **ptr);
++*ptr;
 printf("\n %d  %d  %d", ptr-p, *ptr-a, **ptr);
}
```

Answer:

111
222
333
344

Explanation:

Let us consider the array and the two pointers with some address

a

0	1	2	3	4
100	102	104	106	108

p

100	102	104	106	108
1000	1002	1004	1006	1008

ptr

1000
2000

After execution of the instruction ptr++ value in ptr becomes 1002, if scaling factor for integer is 2 bytes. Now ptr – p is value in ptr – starting location of array p, (1002 – 1000) / (scaling factor) = 1, *ptr – a = value at address pointed by ptr – starting value of array a, 1002 has a value 102 so the value is (102 – 100)/(scaling factor) = 1, **ptr is the value stored in the location pointed by the pointer of ptr = value pointed by value pointed by 1002 = value pointed by 102 = 1. Hence the output of the firs printf is 1, 1, 1.

After execution of *ptr++ increments value of the value in ptr by scaling factor, so it becomes1004. Hence, the outputs for the second printf are ptr – p = 2, *ptr – a = 2, **ptr = 2.
After execution of *++ptr increments value of the value in ptr by scaling factor, so it becomes1004. Hence, the outputs for the third printf are ptr – p = 3, *ptr – a = 3, **ptr = 3.
After execution of ++*ptr value in ptr remains the same, the value pointed by the value is incremented by the scaling factor. So the value in array p at location 1006 changes from 106 10 108,. Hence, the outputs for the fourth printf are ptr – p = 1006 – 1000 = 3, *ptr – a = 108 – 100 = 4, **ptr = 4.

50)
```
main( )
{
 char  *q;
 int  j;
 for (j=0; j<3; j++) scanf("%s" ,(q+j));
 for (j=0; j<3; j++) printf("%c" ,*(q+j));
 for (j=0; j<3; j++) printf("%s" ,(q+j));
}
```

Explanation:

Here we have only one pointer to type char and since we take input in the same pointer thus we keep writing over in the same location, each time shifting the pointer value by 1. Suppose the inputs are MOUSE, TRACK and VIRTUAL. Then for the first input suppose the pointer starts at location 100 then the input one is stored as

M	O	U	S	E	\0

When the second input is given the pointer is incremented as j value becomes 1, so the input is filled in memory starting from 101.

M	T	R	A	C	K	\0

The third input starts filling from the location 102

M	T	V	I	R	T	U	A	L	\0

This is the final value stored .

The first printf prints the values at the position q, q+1 and q+2 = M T V

The second printf prints three strings starting from locations q, q+1, q+2

i.e MTVIRTUAL, TVIRTUAL and VIRTUAL.

51)
```
main( )
{
 void *vp;
 char ch = 'g', *cp = "goofy";
 int j = 20;
 vp = &ch;
 printf("%c", *(char *)vp);
 vp = &j;
 printf("%d",*(int *)vp);
 vp = cp;
 printf("%s",(char *)vp + 3);
```

```
}
```

Answer:

g20fy

Explanation:

Since a void pointer is used it can be type casted to any other type pointer. vp = &ch stores address of char ch and the next statement prints the value stored in vp after type casting it to the proper data type pointer. the output is 'g'. Similarly the output from second printf is '20'. The third printf statement type casts it to print the string from the 4th value hence the output is 'fy'.

52)
```
main ( )
{
 static char *s[ ]  = {"black", "white", "yellow", "violet"};
 char **ptr[ ] = {s+3, s+2, s+1, s}, ***p;
 p = ptr;
 **++p;
 printf("%s",*--*++p + 3);
}
```

Answer:

ck

Explanation:

In this problem we have an array of char pointers pointing to start of 4 strings. Then we have ptr which is a pointer to a pointer of type char and a variable p which is a pointer to a pointer to a pointer of type char. p hold the initial value of ptr, i.e. p = s+3. The next statement increment value in p by 1 , thus now value of p = s+2. In the printf statement the expression is evaluated *++p causes gets value s+1 then the pre decrement is executed and we get s+1 – 1 = s . the indirection operator now gets the value from the array of s and adds 3 to the starting address. The string is printed starting from this position. Thus, the output is 'ck'.

53)
```
main()
{
 int  i, n;
 char *x = "girl";
 n = strlen(x);
 *x = x[n];
 for(i=0; i<n; ++i)
   {
printf("%s\n",x);
x++;
   }
 }
```

Answer:

(blank space)
irl
rl

l

Explanation:

Here a string (a pointer to char) is initialized with a value "girl". The strlen function returns the length of the string, thus n has a value 4. The next statement assigns value at the nth location ('\0') to the first location. Now the string becomes "\0irl" . Now the printf statement prints the string after each iteration it increments it starting position. Loop starts from 0 to 4. The first time x[0] = '\0' hence it prints nothing and pointer value is incremented. The second time it prints from x[1] i.e "irl" and the third time it prints "rl" and the last time it prints "l" and the loop terminates.

54)
```
int i,j;
for(i=0;i<=10;i++)
{
j+=5;
assert(i<5);
}
```
Answer:

Runtime error: Abnormal program termination.
assert failed (i<5), <file name>,<line number>

Explanation:

asserts are used during debugging to make sure that certain conditions are satisfied. If assertion fails, the program will terminate reporting the same. After debugging use,

#undef NDEBUG

and this will disable all the assertions from the source code. Assertion is a good debugging tool to make use of.

55)
```
main()
{
int i=-1;
+i;
printf("i = %d, +i = %d \n",i,+i);
}
```
Answer:

i = -1, +i = -1

Explanation:

Unary + is the only dummy operator in C. Where-ever it comes you can just ignore it just because it has no effect in the expressions (hence the name dummy operator).

56) What are the files which are automatically opened when a C file is executed?

Answer:

stdin, stdout, stderr (standard input,standard output,standard error).

57) what will be the position of the file marker?

a: fseek(ptr,0,SEEK_SET);

b: fseek(ptr,0,SEEK_CUR);

Answer :

a: The SEEK_SET sets the file position marker to the starting of the file.

b: The SEEK_CUR sets the file position marker to the current position of the file.

58)
```
main()
{
char name[10],s[12];
scanf(" \"%[^\"]\"",s);
}
```
How scanf will execute?

Answer:

First it checks for the leading white space and discards it.Then it matches with a quotation mark and then it reads all character upto another quotation mark.

59) What is the problem with the following code segment?
```
while ((fgets(receiving array,50,file_ptr)) != EOF)
                ;
```
Answer & Explanation:

fgets returns a pointer. So the correct end of file check is checking for != NULL.

60)
```
main()
{
main();
}
```
Answer:

Runtime error : Stack overflow.

Explanation:

main function calls itself again and again. Each time the function is called its return address is stored in the call stack. Since there is no condition to terminate the function call, the call stack overflows at runtime. So it terminates the program and results in an error.

61)
```
main()
{
char *cptr,c;
void *vptr,v;
c=10;  v=0;
cptr=&c; vptr=&v;
printf("%c%v",c,v);
}
```
Answer:

Compiler error (at line number 4): size of v is Unknown.

Explanation:

You can create a variable of type void * but not of type void, since void is an empty type. In the second line you are creating variable vptr of type void * and v of type void hence an error.

62)
```
main()
{
char *str1="abcd";
char str2[]="abcd";
printf("%d %d %d",sizeof(str1),sizeof(str2),sizeof("abcd"));
}
```
Answer:

2 5 5

Explanation:

In first sizeof, str1 is a character pointer so it gives you the size of the pointer variable. In second sizeof the name str2 indicates the name of the array whose size is 5 (including the '\0' termination character). The third sizeof is similar to the second one.

63)
```
main()
{
char not;
not=!2;
printf("%d",not);
}
```
Answer:

0

Explanation:

! is a logical operator. In C the value 0 is considered to be the boolean value FALSE, and any non-zero value is considered to be the boolean value TRUE. Here 2 is a non-zero value so TRUE. !TRUE is FALSE (0) so it prints 0.

64)
```
#define FALSE -1
#define TRUE   1
#define NULL   0
main() {
  if(NULL)
        puts("NULL");
  else if(FALSE)
        puts("TRUE");
  else
        puts("FALSE");
  }
```
Answer:

TRUE

Explanation:

The input program to the compiler after processing by the preprocessor is,

```
main(){
if(0)
        puts("NULL");
else if(-1)
        puts("TRUE");
else
        puts("FALSE");
}
```

Preprocessor doesn't replace the values given inside the double quotes. The check by if condition is boolean value false so it goes to else. In second if -1 is boolean value true hence "TRUE" is printed.

65)
```
main()
{
int k=1;
printf("%d==1 is ""%s",k,k==1?"TRUE":"FALSE");
}
```
Answer:

1==1 is TRUE

Explanation:

When two strings are placed together (or separated by white-space) they are concatenated (this is called as "stringization" operation). So the string is as if it is given as "%d==1 is %s". The conditional operator(?:) evaluates to "TRUE".

66)
```
main()
{
int y;
scanf("%d",&y); // input given is 2000
if( (y%4==0 && y%100 != 0) || y%100 == 0 )
    printf("%d is a leap year");
else
    printf("%d is not a leap year");
}
```
Answer:

2000 is a leap year

Explanation:

An ordinary program to check if leap year or not.

67)
```
#define max 5
#define int arr1[max]
main()
{
typedef char arr2[max];
arr1 list={0,1,2,3,4};
arr2 name="name";
printf("%d %s",list[0],name);
```

```
}
```

Answer:

Compiler error (in the line arr1 list = {0,1,2,3,4})

Explanation:

arr2 is declared of type array of size 5 of characters. So it can be used to declare the variable name of the type arr2. But it is not the case of arr1. Hence an error.

Rule of Thumb:

#defines are used for textual replacement whereas typedefs are used for declaring new types.

68)
```
int i=10;
main()
{
 extern int i;
  {
    int i=20;
          {
           const volatile unsigned i=30;
           printf("%d",i);
          }
     printf("%d",i);
  }
printf("%d",i);
}
```

Answer:

30,20,10

Explanation:

'{' introduces new block and thus new scope. In the innermost block i is declared as,

```
const volatile unsigned
```

which is a valid declaration. i is assumed of type int. So printf prints 30. In the next block, i has value 20 and so printf prints 20. In the outermost block, i is declared as extern, so no storage space is allocated for it. After compilation is over the linker resolves it to global variable i (since it is the only variable visible there). So it prints i's value as 10.

69)
```
main()
{
   int *j;
   {
    int i=10;
    j=&i;
    }
    printf("%d",*j);
}
```

Answer:

10

Explanation:
The variable i is a block level variable and the visibility is inside that block only. But the lifetime of i is lifetime of the function so it lives upto the exit of main function. Since the i is still allocated space, *j prints the value stored in i since j points i.

70)
```
main()
{
int i=-1;
-i;
printf("i = %d, -i = %d \n",i,-i);
}
```
Answer:
i = -1, -i = 1
Explanation:
-i is executed and this execution doesn't affect the value of i. In printf first you just print the value of i. After that the value of the expression -i = -(-1) is printed.

71)
```
#include<stdio.h>
main()
 {
  const int i=4;
  float j;
  j = ++i;
  printf("%d  %f", i,++j);
 }
```
Answer:
Compiler error
Explanation:
i is a constant. you cannot change the value of constant

72)
```
#include<stdio.h>
main()
{
  int a[2][2][2] = { {10,2,3,4}, {5,6,7,8}  };
  int *p,*q;
p=&a[2][2][2];
  *q=***a;
  printf("%d..%d",*p,*q);
}
```
Answer:
garbagevalue..1
Explanation:
p=&a[2][2][2] you declare only two 2D arrays. but you are trying to access the third 2D(which you are not declared) it will print garbage values. *q=***a starting address of a is assigned integer pointer. now q is pointing to starting address of a.if you print *q meAnswer:it will print first element of 3D array.

73)
```
#include<stdio.h>
main()
  {
    register i=5;
    char j[]= "hello";
     printf("%s  %d",j,i);
}
```
Answer:

hello 5

Explanation:

if you declare i as register compiler will treat it as ordinary integer and it will take integer value. i value may be stored either in register or in memory.

74)
```
main()
{
  int i=5,j=6,z;
  printf("%d",i+++j);
 }
```
Answer:

11

Explanation:

the expression i+++j is treated as (i++ + j)

76)
```
struct aaa{
struct aaa *prev;
int i;
struct aaa *next;
};
main()
{
 struct aaa abc,def,ghi,jkl;
 int x=100;
 abc.i=0;abc.prev=&jkl;
 abc.next=&def;
 def.i=1;def.prev=&abc;def.next=&ghi;
 ghi.i=2;ghi.prev=&def;
 ghi.next=&jkl;
 jkl.i=3;jkl.prev=&ghi;jkl.next=&abc;
 x=abc.next->next->prev->next->i;
 printf("%d",x);
}
```
Answer:

2

Explanation:

above all statements form a double circular linked list;

abc.next->next->prev->next->i
this one points to "ghi" node the value of at particular node is 2.

77)
```
struct point
 {
 int x;
 int y;
 };
struct point origin,*pp;
main()
{
pp=&origin;
printf("origin is(%d%d)\n",(*pp).x,(*pp).y);
printf("origin is (%d%d)\n",pp->x,pp->y);
}
```

Answer:
origin is(0,0)
origin is(0,0)
Explanation:
pp is a pointer to structure. we can access the elements of the structure either with arrow mark or with indirection operator.

Note:
Since structure point is globally declared x & y are initialized as zeroes

78)
```
main()
{
int i=_l_abc(10);
 printf("%d\n",--i);
}
int _l_abc(int i)
{
 return(i++);
}
```

Answer:
9
Explanation:
return(i++) it will first return i and then increments. i.e. 10 will be returned.

79)
```
main()
{
 char *p;
 int *q;
 long *r;
 p=q=r=0;
 p++;
```

```
 q++;
 r++;
 printf("%p...%p...%p",p,q,r);
}
```

Answer:

0001...0002...0004

Explanation:

++ operator when applied to pointers increments address according to their corresponding data-types.

80)
```
main()
{
 char c=' ',x,convert(z);
 getc(c);
 if((c>='a') && (c<='z'))
 x=convert(c);
 printf("%c",x);
}
convert(z)
{
  return z-32;
}
```

Answer:

Compiler error

Explanation:

declaration of convert and format of getc() are wrong.

81)
```
main(int argc, char **argv)
{
 printf("enter the character");
 getchar();
 sum(argv[1],argv[2]);
}
sum(num1,num2)
int num1,num2;
{
 return num1+num2;
}
```

Answer:

Compiler error.

Explanation:

argv[1] & argv[2] are strings. They are passed to the function sum without converting it to integer values.

82)
```
# include <stdio.h>
int one_d[]={1,2,3};
```

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

Answer:
garbage value
Explanation:
ptr pointer is pointing to out of the array range of one_d.

83)
```
# include<stdio.h>
aaa() {
  printf("hi");
 }
bbb(){
 printf("hello");
 }
ccc(){
 printf("bye");
 }
main()
{
  int (*ptr[3])();
  ptr[0]=aaa;
  ptr[1]=bbb;
  ptr[2]=ccc;
  ptr[2]();
}
```

Answer:
bye
Explanation:
ptr is array of pointers to functions of return type int.ptr[0] is assigned to address of the function aaa. Similarly ptr[1] and ptr[2] for bbb and ccc respectively. ptr[2]() is in effect of writing ccc(), since ptr[2] points to ccc.

85)
```
#include<stdio.h>
main()
{
FILE *ptr;
char i;
ptr=fopen("zzz.c","r");
while((i=fgetch(ptr))!=EOF)
printf("%c",i);
}
```

Answer:
contents of zzz.c followed by an infinite loop
Explanation:
The condition is checked against EOF, it should be checked against NULL.

86)
```
main()
{
 int i =0;j=0;
 if(i && j++)
        printf("%d..%d",i++,j);
printf("%d..%d,i,j);
}
```
Answer:
0..0
Explanation:
The value of i is 0. Since this information is enough to determine the truth value of the boolean expression. So the statement following the if statement is not executed. The values of i and j remain unchanged and get printed.

87)
```
main()
{
 int i;
 i = abc();
 printf("%d",i);
}
abc()
{
 _AX = 1000;
}
```
Answer:
1000
Explanation:
Normally the return value from the function is through the information from the accumulator. Here _AH is the pseudo global variable denoting the accumulator. Hence, the value of the accumulator is set 1000 so the function returns value 1000.

88)
```
int i;
main(){
int t;
for ( t=4;scanf("%d",&i)-t;printf("%d\n",i))
                printf("%d--",t--);
        }
// If the inputs are 0,1,2,3 find the o/p
```
Answer:
```
        4--0
        3--1
```

2--2

Explanation:

Let us assume some x= scanf("%d",&i)-t the values during execution will be,

t	i	x
4	0	-4
3	1	-2
2	2	0

89)
```
main(){
 int a= 0;int b = 20;char x =1;char y =10;
 if(a,b,x,y)
      printf("hello");
 }
```

Answer:

hello

Explanation:

The comma operator has associativity from left to right. Only the rightmost value is returned and the other values are evaluated and ignored. Thus the value of last variable y is returned to check in if. Since it is a non zero value if becomes true so, "hello" will be printed.

90)
```
main(){
 unsigned int i;
 for(i=1;i>-2;i--)
         printf("c aptitude");
}
```

Explanation:

i is an unsigned integer. It is compared with a signed value. Since the both types doesn't match, signed is promoted to unsigned value. The unsigned equivalent of -2 is a huge value so condition becomes false and control comes out of the loop.

91) In the following pgm add a stmt in the function fun such that the address of 'a' gets stored in 'j'.

```
main(){
 int * j;
 void fun(int **);
 fun(&j);
 }
 void fun(int **k) {
 int a =0;
 /* add a stmt here*/
 }
```

Answer:

*k = &a

Explanation:

The argument of the function is a pointer to a pointer.

92) What are the following notations of defining functions known as?

i.
```
int abc(int a,float b)
    {
    /* some code */
}
```

ii.
```
int abc(a,b)
 int a; float b;
    {
    /* some code*/
    }
```

Answer:

i. ANSI C notation

ii. Kernighan & Ritche notation

93)
```
main()
{
char *p;
p="%d\n";
p++;
p++;
printf(p-2,300);
}
```

Answer:

300

Explanation:

The pointer points to % since it is incremented twice and again decremented by 2, it points to '%d\n' and 300 is printed.

94)
```
main(){
 char a[100];
 a[0]='a';a[1]]='b';a[2]='c';a[4]='d';
 abc(a);
}
abc(char a[]){
 a++;
 printf("%c",*a);
 a++;
 printf("%c",*a);
}
```

Explanation:

The base address is modified only in function and as a result a points to 'b' then after incrementing to 'c' so bc will be printed.

95) func(a,b)

```
int a,b;
{
 return( a= (a==b) );
}
main()
{
int process(),func();
printf("The value of process is %d !\n ",process(func,3,6));
}
process(pf,val1,val2)
int (*pf) ();
int val1,val2;
{
return((*pf) (val1,val2));
 }
```

Answer:

The value if process is 0 !

Explanation:

The function 'process' has 3 parameters - 1, a pointer to another function 2 and 3, integers. When this function is invoked from main, the following substitutions for formal parameters take place: func for pf, 3 for val1 and 6 for val2. This function returns the result of the operation performed by the function 'func'. The function func has two integer parameters. The formal parameters are substituted as 3 for a and 6 for b. since 3 is not equal to 6, a==b returns 0. therefore the function returns 0 which in turn is returned by the function 'process'.

96)

```
void main()
{
        static int i=5;
        if(--i){
                main();
                printf("%d ",i);
        }
}
```

Answer:

0 0 0 0

Explanation:

The variable "I" is declared as static, hence memory for I will be allocated for only once, as it encounters the statement. The function main() will be called recursively unless I becomes equal to 0, and since main() is recursively called, so the value of static I ie., 0 will be printed every time the control is returned.

97)

```
void main()
{
        int k=ret(sizeof(float));
        printf("\n here value is %d",++k);
```

```
}
int ret(int ret)
{
        ret += 2.5;
        return(ret);
}
```

Answer:

Here value is 7

Explanation:

The int ret(int ret), ie., the function name and the argument name can be the same.

Firstly, the function ret() is called in which the sizeof(float) ie., 4 is passed, after the first expression the value in ret will be 6, as ret is integer hence the value stored in ret will have implicit type conversion from float to int. The ret is returned in main() it is printed after and preincrement.

98)

```
void main()
{
        char a[]="12345\0";
        int i=strlen(a);
        printf("here in 3 %d\n",++i);
}
```

Answer:

here in 3 6

Explanation:

The char array 'a' will hold the initialized string, whose length will be counted from 0 till the null character. Hence the 'I' will hold the value equal to 5, after the pre-increment in the printf statement, the 6 will be printed.

99)

```
void main()
{
        unsigned giveit=-1;
        int gotit;
        printf("%u ",++giveit);
        printf("%u \n",gotit=--giveit);
}
```

Answer:

0 65535

Explanation:

100)

```
void main()
{
        int i;
        char a[]="\0";
        if(printf("%s\n",a))
                printf("Ok here \n");
```

```
        else
                printf("Forget it\n");
}
```

Answer:

Ok here

Explanation:

Printf will return how many characters does it print. Hence printing a null character returns 1 which makes the if statement true, thus "Ok here" is printed.

```
101)    void main()
{
        void *v;
        int integer=2;
        int *i=&integer;
        v=i;
        printf("%d",(int*)*v);
}
```

Answer:

Compiler Error. We cannot apply indirection on type void*.

Explanation:

Void pointer is a generic pointer type. No pointer arithmetic can be done on it. Void pointers are normally used for,

1. Passing generic pointers to functions and returning such pointers.
2. As a intermediate pointer type.
3. Used when the exact pointer type will be known at a later point of time.

```
102)    void main()
{
        int i=i++,j=j++,k=k++;
        printf("%d%d%d",i,j,k);
}
```

Answer:

Garbage values.

Explanation:

An identifier is available to use in program code from the point of its declaration.

So expressions such as i = i++ are valid statements. The i, j and k are automatic variables and so they contain some garbage value. *Garbage in is garbage out (GIGO)*.

```
103)    void main()
{
        static int i=i++, j=j++, k=k++;
        printf("i = %d j = %d k = %d", i, j, k);
```

```
}
```

Answer:

i = 1 j = 1 k = 1

Explanation:

Since static variables are initialized to zero by default.

```
104)    void main()
{
        while(1){
                if(printf("%d",printf("%d")))
                        break;
                else
                        continue;
        }
}
```

Answer:

Garbage values

Explanation:

The inner printf executes first to print some garbage value. The printf returns no of characters printed and this value also cannot be predicted. Still the outer printf prints something and so returns a non-zero value. So it encounters the break statement and comes out of the while statement.

```
104)    main()
{
        unsigned int i=10;
        while(i-->=0)
                printf("%u ",i);

}
```

Answer:

10 9 8 7 6 5 4 3 2 1 0 65535 65534…..

Explanation:

Since i is an unsigned integer it can never become negative. So the expression i-- >=0 will always be true, leading to an infinite loop.

```
105)    #include<conio.h>
main()
{
        int x,y=2,z,a;
        if(x=y%2) z=2;
        a=2;
        printf("%d %d ",z,x);
}
```

Answer:

Garbage-value 0

Explanation:

The value of y%2 is 0. This value is assigned to x. The condition reduces to if (x) or in other words if(0) and so z goes uninitialized.

Thumb Rule: **C**heck all control paths to write bug free code.

```
106)    main()
{
        int a[10];
        printf("%d",*a+1-*a+3);
}
```

Answer:

4

Explanation:

*a and -*a cancels out. The result is as simple as 1 + 3 = 4 !

```
107)    #define prod(a,b) a*b
main()
{
        int x=3,y=4;
        printf("%d",prod(x+2,y-1));
}
```

Answer:

10

Explanation:

The macro expands and evaluates to as:

x+2*y-1 => x+(2*y)-1 => 10

```
108)    main()
{
        unsigned int i=65000;
        while(i++!=0);
        printf("%d",i);
}
```

Answer:

1

Explanation:

Note the semicolon after the while statement. When the value of i becomes 0 it comes out of while loop. Due to post-increment on i the value of i while printing is 1.

```
109)    main()
{
        int i=0;
        while(+(+i--)!=0)
                i-=i++;
        printf("%d",i);
}
```

Answer:

-1

Explanation:

Unary + is the only dummy operator in C. So it has no effect on the expression and now the while loop is, while(i--!=0) which is false and so breaks out of while loop. The value –1 is printed due to the post-decrement operator.

```
113)    main()
{
        float f=5,g=10;
        enum{i=10,j=20,k=50};
        printf("%d\n",++k);
        printf("%f\n",f<<2);
        printf("%lf\n",f%g);
        printf("%lf\n",fmod(f,g));
}
```

Answer:

Line no 5: Error: Lvalue required
Line no 6: Cannot apply leftshift to float
Line no 7: Cannot apply mod to float

Explanation:

Enumeration constants cannot be modified, so you cannot apply ++.
Bit-wise operators and % operators cannot be applied on float values.
fmod() is to find the modulus values for floats as % operator is for ints.

```
110)    main()
{
        int i=10;
        void pascal f(int,int,int);
        f(i++,i++,i++);
        printf(" %d",i);
}
void pascal f(integer :i,integer:j,integer :k)
        {
        write(i,j,k);
        }
```

Answer:

Compiler error: unknown type integer
Compiler error: undeclared function write

Explanation:

Pascal keyword doesn't mean that pascal code can be used. It means that the function follows Pascal argument passing mechanism in calling the functions.

```
111)    void pascal f(int i,int j,int k)
        {
        printf("%d %d %d",i, j, k);
```

```
}
void cdecl f(int i,int j,int k)
{
printf("%d %d %d",i, j, k);
}
main()
{
int i=10;
f(i++,i++,i++);
printf(" %d\n",i);
i=10;
f(i++,i++,i++);
printf(" %d",i);
}
```

Answer:

10 11 12 13

12 11 10 13

Explanation:

Pascal argument passing mechanism forces the arguments to be called from left to right. cdecl is the normal C argument passing mechanism where the arguments are passed from right to left.

112). What is the output of the program given below

```
main()
   {
     signed char i=0;
     for(;i>=0;i++) ;
     printf("%d\n",i);
   }
```

Answer

-128

Explanation

Notice the semicolon at the end of the for loop. THe initial value of the i is set to 0. The inner loop executes to increment the value from 0 to 127 (the positive range of char) and then it rotates to the negative value of -128. The condition in the for loop fails and so comes out of the for loop. It prints the current value of i that is -128.

113)

```
main()
   {
     unsigned char i=0;
     for(;i>=0;i++) ;
     printf("%d\n",i);
   }
```

Answer

infinite loop

Explanation

The difference between the previous question and this one is that the char is declared to be unsigned. So the i++ can never yield negative value and i>=0 never becomes false so that it can come out of the for loop.

114) main()

```
{
    char i=0;
    for(;i>=0;i++) ;
    printf("%d\n",i);

}
```

Answer:

Behavior is implementation dependent.

Explanation:

The detail if the char is signed/unsigned by default is implementation dependent. If the implementation treats the char to be signed by default the program will print –128 and terminate. On the other hand if it considers char to be unsigned by default, it goes to infinite loop.

Rule:

You can write programs that have implementation dependent behavior. But dont write programs that depend on such behavior.

115) Is the following statement a declaration/definition. Find what does it mean?

int (*x)[10];

Answer

Definition.

x is a pointer to array of(size 10) integers.

Apply clock-wise rule to find the meaning of this definition.

116). What is the output for the program given below

```
typedef enum errorType{warning, error, exception,}error;
main()
{
  error g1;
  g1=1;
  printf("%d",g1);
}
```

Answer

Compiler error: Multiple declaration for error

Explanation

The name error is used in the two meanings. One means that it is a enumerator constant with value 1. The another use is that it is a type name (due to typedef) for

enum errorType. Given a situation the compiler cannot distinguish the meaning of error to know in what sense the error is used:

```
        error g1;
g1=error;
        // which error it refers in each case?
```

When the compiler can distinguish between usages then it will not issue error (in pure technical terms, names can only be overloaded in different namespaces).

Note: the extra comma in the declaration,

enum errorType{warning, error, exception,}

is not an error. An extra comma is valid and is provided just for programmer's convenience.

117)
```
typedef struct error{int warning, error, exception;}error;
main()
{
  error g1;
  g1.error =1;
  printf("%d",g1.error);
}
```

Answer

1

Explanation

The three usages of name errors can be distinguishable by the compiler at any instance, so valid (they are in different namespaces).

Typedef struct error{int warning, error, exception;}error;

This error can be used only by preceding the error by struct kayword as in:

struct error someError;

typedef struct error{int warning, error, exception;}error;

This can be used only after . (dot) or -> (arrow) operator preceded by the variable name as in :

```
g1.error =1;
        printf("%d",g1.error);
        typedef struct error{int warning, error, exception;}error;
```

This can be used to define variables without using the preceding struct keyword as in:

error g1;

Since the compiler can perfectly distinguish between these three usages, it is perfectly legal and valid.

Note

This code is given here to just explain the concept behind. In real programming don't use such overloading of names. It reduces the readability of the code. Possible doesn't mean that we should use it!

118) #ifdef something

```
int some=0;
#endif

main()
{
int thing = 0;
printf("%d %d\n", some ,thing);
}
```

Answer:

Compiler error : undefined symbol some

Explanation:

This is a very simple example for conditional compilation. The name something is not already known to the compiler making the declaration

```
int some = 0;
```

effectively removed from the source code.

119)

```
#if something == 0
int some=0;
#endif

main()
{
int thing = 0;
printf("%d %d\n", some ,thing);
}
```

Answer

0 0

Explanation

This code is to show that preprocessor expressions are not the same as the ordinary expressions. If a name is not known the preprocessor treats it to be equal to zero.

120). What is the output for the following program

```
main()
        {
    int arr2D[3][3];
     printf("%d\n", ((arr2D==* arr2D)&&(* arr2D == arr2D[0])) );
  }
```

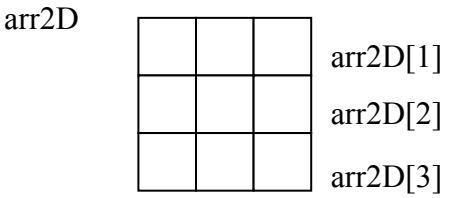
Answer

1

Explanation

This is due to the close relation between the arrays and pointers. N dimensional arrays are made up of (N-1) dimensional arrays.

arr2D is made up of a 3 single arrays that contains 3 integers each .



The name arr2D refers to the beginning of all the 3 arrays. *arr2D refers to the start of the first 1D array (of 3 integers) that is the same address as arr2D. So the expression (arr2D == *arr2D) is true (1).
Similarly, *arr2D is nothing but *(arr2D + 0), adding a zero doesn't change the value/meaning. Again arr2D[0] is the another way of telling *(arr2D + 0). So the expression (*(arr2D + 0) == arr2D[0]) is true (1).
Since both parts of the expression evaluates to true the result is true(1) and the same is printed.

121) void main()
```
{
  if(~0 == (unsigned int)-1)
printf("You can answer this if you know how values are represented in memory");
}
```
Answer
You can answer this if you know how values are represented in memory
Explanation
~ (tilde operator or bit-wise negation operator) operates on 0 to produce all ones to fill the space for an integer. –1 is represented in unsigned value as all 1's and so both are equal.

122) int swap(int *a,int *b)
```
{
 *a=*a+*b;*b=*a-*b;*a=*a-*b;
}
main()
{
        int x=10,y=20;
        swap(&x,&y);
        printf("x= %d y = %d\n",x,y);
}
```
Answer
x = 20 y = 10
Explanation
This is one way of swapping two values. Simple checking will help understand this.

123) main()
```
{
```

```
char *p = “ayqm”;
printf(“%c”,++*(p++));
}
```

Answer:
b

124)
```
main()
{
 int i=5;
 printf("%d",++i++);
}
```

Answer:

Compiler error: Lvalue required in function main

Explanation:

++i yields an rvalue. For postfix ++ to operate an lvalue is required.

125)
```
main()
{
char *p = “ayqm”;
char c;
c = ++*p++;
printf(“%c”,c);
}
```

Answer:
b

Explanation:

There is no difference between the expression ++*(p++) and ++*p++. Parenthesis just works as a visual clue for the reader to see which expression is first evaluated.

126)
```
int aaa() {printf(“Hi”);}
int bbb(){printf(“hello”);}
iny ccc(){printf(“bye”);}

main()
{
int ( * ptr[3]) ();
ptr[0] = aaa;
ptr[1] = bbb;
ptr[2] =ccc;
ptr[2]();
}
```

Answer:
bye
Explanation:

int (* ptr[3])() says that ptr is an array of pointers to functions that takes no arguments and returns the type int. By the assignment ptr[0] = aaa; it means that the first function pointer in the array is initialized with the address of the function aaa. Similarly, the other two array elements also get initialized with the addresses of the functions bbb and ccc. Since ptr[2] contains the address of the function ccc, the call to the function ptr[2]() is same as calling ccc(). So it results in printing "bye".

127)

```
main()
{
int i=5;
printf("%d",i=++i ==6);
}
```

Answer:

1

Explanation:

The expression can be treated as i = (++i==6), because == is of higher precedence than = operator. In the inner expression, ++i is equal to 6 yielding true(1). Hence the result.

128)

```
main()
{
        char p[ ]="%d\n";
p[1] = 'c';
printf(p,65);
}
```

Answer:

A

Explanation:

Due to the assignment p[1] = 'c' the string becomes, "%c\n". Since this string becomes the format string for printf and ASCII value of 65 is 'A', the same gets printed.

129) void (* abc(int, void (*def) ())) ();

Answer::

abc is a ptr to a function which takes 2 parameters .(a). an integer variable.(b). a ptrto a funtion which returns void. the return type of the function is void.

Explanation:

Apply the clock-wise rule to find the result.

130)

```
main()
{
while (strcmp("some","some\0"))
```

```
printf("Strings are not equal\n");
}
```

Answer:

No output

Explanation:

Ending the string constant with \0 explicitly makes no difference. So "some" and "some\0" are equivalent. So, strcmp returns 0 (false) hence breaking out of the while loop.

131)
```
main()
{
char str1[] = {'s','o','m','e'};
char str2[] = {'s','o','m','e','\0'};
while (strcmp(str1,str2))
printf("Strings are not equal\n");
}
```

Answer:

"Strings are not equal"
"Strings are not equal"
….

Explanation:

If a string constant is initialized explicitly with characters, '\0' is not appended automatically to the string. Since str1 doesn't have null termination, it treats whatever the values that are in the following positions as part of the string until it randomly reaches a '\0'. So str1 and str2 are not the same, hence the result.

132)
```
main()
{
int i = 3;
for (;i++=0;) printf("%d",i);
}
```

Answer:

Compiler Error: Lvalue required.

Explanation:

As we know that increment operators return rvalues and hence it cannot appear on the left hand side of an assignment operation.

133)
```
void main()
{
int *mptr, *cptr;
mptr = (int*)malloc(sizeof(int));
printf("%d",*mptr);
int *cptr = (int*)calloc(sizeof(int),1);
printf("%d",*cptr);
}
```

Answer:
garbage-value 0
Explanation:
The memory space allocated by malloc is uninitialized, whereas calloc returns the allocated memory space initialized to zeros.

134)
```
void main()
{
static int i;
while(i<=10)
(i>2)?i++:i--;
        printf("%d", i);
}
```
Answer:
32767
Explanation:
Since i is static it is initialized to 0. Inside the while loop the conditional operator evaluates to false, executing i--. This continues till the integer value rotates to positive value (32767). The while condition becomes false and hence, comes out of the while loop, printing the i value.

135)
```
main()
{
        int i=10,j=20;
        j = i, j?(i,j)?i:j:j;
        printf("%d %d",i,j);
}
```

Answer:
10 10
Explanation:
The Ternary operator (? :) is equivalent for if-then-else statement. So the question can be written as:
```
        if(i,j)
            {
if(i,j)
             j = i;
        else
           j = j;
        }
  else
        j = j;
```

136)
```
1. const char *a;
2. char* const a;
```

3. char const *a;

-Differentiate the above declarations.

Answer:

1. 'const' applies to char * rather than 'a' (pointer to a constant char)

*a='F' : illegal

a="Hi" : legal

2. 'const' applies to 'a' rather than to the value of a (constant pointer to char)

*a='F' : legal

a="Hi" : illegal

3. Same as 1.

137)
```
main()
{
    int i=5,j=10;
    i=i&=j&&10;
    printf("%d %d",i,j);
}
```

Answer:

1 10

Explanation:

The expression can be written as i=(i&=(j&&10)); The inner expression (j&&10) evaluates to 1 because j==10. i is 5. i = 5&1 is 1. Hence the result.

138)
```
main()
{
    int i=4,j=7;
    j = j || i++ && printf("YOU CAN");
    printf("%d %d", i, j);
}
```

Answer:

4 1

Explanation:

The boolean expression needs to be evaluated only till the truth value of the expression is not known. j is not equal to zero itself means that the expression's truth value is 1. Because it is followed by || and *true || (anything) => true where (anything) will not be evaluated.* So the remaining expression is not evaluated and so the value of i remains the same.

Similarly when && operator is involved in an expression, when any of the operands become false, the whole expression's truth value becomes false and hence the remaining expression will not be evaluated.

false && (anything) => false where (anything) will not be evaluated.

139)
```
main()
{
        register int a=2;
        printf("Address of a = %d",&a);
        printf("Value of a   = %d",a);
}
```
Answer:
Compier Error: '&' on register variable
Rule to Remember:
& (address of) operator cannot be applied on register variables.

140)
```
main()
{
        float i=1.5;
        switch(i)
        {
                case 1: printf("1");
                case 2: printf("2");
                default : printf("0");
        }
}
```
Answer:
Compiler Error: switch expression not integral
Explanation:
Switch statements can be applied only to integral types.

141)
```
main()
{
        extern i;
        printf("%d\n",i);
        {
                int i=20;
                printf("%d\n",i);
        }
}
```
Answer:
Linker Error : Unresolved external symbol i
Explanation:
The identifier i is available in the inner block and so using extern has no use in resolving it.

142)
```
main()
{
        int a=2,*f1,*f2;
        f1=f2=&a;
```

```
        *f2+=*f2+=a+=2.5;
        printf("\n%d %d %d",a,*f1,*f2);
}
```

Answer:

16 16 16

Explanation:

f1 and f2 both refer to the same memory location a. So changes through f1 and f2 ultimately affects only the value of a.

143)
```
main()
{
        char *p="GOOD";
        char a[ ]="GOOD";
printf("\n sizeof(p) = %d, sizeof(*p) = %d, strlen(p) = %d", sizeof(p), sizeof(*p), strlen(p));
        printf("\n sizeof(a) = %d, strlen(a) = %d", sizeof(a), strlen(a));
}
```

Answer:

sizeof(p) = 2, sizeof(*p) = 1, strlen(p) = 4
sizeof(a) = 5, strlen(a) = 4

Explanation:

sizeof(p) => sizeof(char*) => 2
sizeof(*p) => sizeof(char) => 1
Similarly,
sizeof(a) => size of the character array => 5

When sizeof operator is applied to an array it returns the sizeof the array and it is not the same as the sizeof the pointer variable. Here the sizeof(a) where a is the character array and the size of the array is 5 because the space necessary for the terminating NULL character should also be taken into account.

144)
```
#define DIM( array, type) sizeof(array)/sizeof(type)
main()
{
int arr[10];
printf("The dimension of the array is %d", DIM(arr, int));
}
```

Answer:

10

Explanation:

The size of integer array of 10 elements is 10 * sizeof(int). The macro expands to sizeof(arr)/sizeof(int) => 10 * sizeof(int) / sizeof(int) => 10.

145)
```
int DIM(int array[])
{
return sizeof(array)/sizeof(int );
}
```

```
main()
{
int arr[10];
printf("The dimension of the array is %d", DIM(arr));
}
```

Answer:

1

Explanation:

Arrays cannot be passed to functions as arguments and only the pointers can be passed. So the argument is equivalent to int * array (this is one of the very few places where [] and * usage are equivalent). The return statement becomes, sizeof(int *)/ sizeof(int) that happens to be equal in this case.

146)
```
main()
{
        static int a[3][3]={1,2,3,4,5,6,7,8,9};
        int i,j;
        static *p[]={a,a+1,a+2};
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                printf("%d\t%d\t%d\t%d\n",*(*(p+i)+j),
                *(*(j+p)+i),*(*(i+p)+j),*(*(p+j)+i));
        }
}
```

Answer:

1	1	1	1
2	4	2	4
3	7	3	7
4	2	4	2
5	5	5	5
6	8	6	8
7	3	7	3
8	6	8	6
9	9	9	9

Explanation:

((p+i)+j) is equivalent to p[i][j].

147)
```
main()
{
        void swap();
        int x=10,y=8;
        swap(&x,&y);
        printf("x=%d y=%d",x,y);
}
void swap(int *a, int *b)
```

```
{
  *a ^= *b,  *b ^= *a, *a ^= *b;
}
```

Answer:

x=10 y=8

Explanation:

Using ^ like this is a way to swap two variables without using a temporary variable and that too in a single statement.

Inside main(), void swap(); means that swap is a function that may take any number of arguments (not no arguments) and returns nothing. So this doesn't issue a compiler error by the call swap(&x,&y); that has two arguments.

This convention is historically due to pre-ANSI style (referred to as Kernighan and Ritchie style) style of function declaration. In that style, the swap function will be defined as follows,

```
void swap()
int *a, int *b
{
  *a ^= *b,  *b ^= *a, *a ^= *b;
}
```

where the arguments follow the (). So naturally the declaration for swap will look like, void swap() which means the swap can take any number of arguments.

148)

```
main()
{
        int i = 257;
        int *iPtr = &i;
        printf("%d %d", *((char*)iPtr), *((char*)iPtr+1) );
}
```

Answer:

1 1

Explanation:

The integer value 257 is stored in the memory as, 00000001 00000001, so the individual bytes are taken by casting it to char * and get printed.

149)

```
main()
{
        int i = 258;
        int *iPtr = &i;
        printf("%d %d", *((char*)iPtr), *((char*)iPtr+1) );
}
```

Answer:

2 1

Explanation:

The integer value 257 can be represented in binary as, 00000001 00000001. Remember that the INTEL machines are 'small-endian' machines. *Small-endian means that the lower order bytes are stored in the higher memory addresses and the*

higher order bytes are stored in lower addresses. The integer value 258 is stored in memory as: 00000001 00000010.

150)
```
main()
{
        int i=300;
        char *ptr = &i;
        *++ptr=2;
        printf("%d",i);
}
```
Answer:
556
Explanation:
The integer value 300 in binary notation is: 00000001 00101100. It is stored in memory (small-endian) as: 00101100 00000001. Result of the expression *++ptr = 2 makes the memory representation as: 00101100 00000010. So the integer corresponding to it is 00000010 00101100 => 556.

151)
```
#include <stdio.h>
main()
{
char * str = "hello";
char * ptr = str;
char least = 127;
while (*ptr++)
     least = (*ptr<least ) ?*ptr :least;
printf("%d",least);
}
```
Answer:
0
Explanation:
After 'ptr' reaches the end of the string the value pointed by 'str' is '\0'. So the value of 'str' is less than that of 'least'. So the value of 'least' finally is 0.

152) Declare an array of N pointers to functions returning pointers to functions returning pointers to characters?
Answer:
```
(char*(*)( )) (*ptr[N])( );
```

153)
```
main()
{
struct student
{
char name[30];
struct date dob;
}stud;
```

```
struct date
      {
      int day,month,year;
      };
 scanf("%s%d%d%d", stud.rollno, &student.dob.day, &student.dob.month,
&student.dob.year);
}
```

Answer:

Compiler Error: Undefined structure date

Explanation:

Inside the struct definition of 'student' the member of type struct date is given. The compiler doesn't have the definition of date structure (forward reference is not allowed in C in this case) so it issues an error.

154)

```
main()
{
struct date;
struct student
{
char name[30];
struct date dob;
}stud;
struct date
      {
      int day,month,year;
 };
scanf("%s%d%d%d", stud.rollno, &student.dob.day, &student.dob.month,
&student.dob.year);
}
```

Answer:

Compiler Error: Undefined structure date

Explanation:

Only declaration of struct date is available inside the structure definition of 'student' but to have a variable of type struct date the definition of the structure is required.

155) There were 10 records stored in "somefile.dat" but the following program printed 11 names. What went wrong?

```
void main()
{
struct student
{
char name[30], rollno[6];
}stud;
FILE *fp = fopen("somefile.dat","r");
while(!feof(fp))
 {
```

```
                fread(&stud, sizeof(stud), 1 , fp);
        puts(stud.name);
}
        }
```

Explanation:

fread reads 10 records and prints the names successfully. It will return EOF only when fread tries to read another record and fails reading EOF (and returning EOF). So it prints the last record again. After this only the condition feof(fp) becomes false, hence comes out of the while loop.

156) Is there any difference between the two declarations,

1. int foo(int *arr[]) and
2. int foo(int *arr[2])

Answer:

No

Explanation:

Functions can only pass pointers and not arrays. The numbers that are allowed inside the [] is just for more readability. So there is no difference between the two declarations.

157) What is the subtle error in the following code segment?

```
        void fun(int n, int arr[])
        {
        int *p=0;
        int i=0;
        while(i++<n)
                        p = &arr[i];
        *p = 0;
}
```

Answer & Explanation:

If the body of the loop never executes p is assigned no address. So p remains NULL where *p =0 may result in problem (may rise to runtime error "NULL pointer assignment" and terminate the program).

158) What is wrong with the following code?

```
        int *foo()
        {
        int *s = malloc(sizeof(int)100);
        assert(s != NULL);
        return s;
        }
```

Answer & Explanation:

assert macro should be used for debugging and finding out bugs. The check s != NULL is for error/exception handling and for that assert shouldn't be used. A plain if and the corresponding remedy statement has to be given.

159) What is the hidden bug with the following statement?

```
assert(val++ != 0);
```

Answer & Explanation:

Assert macro is used for debugging and removed in release version. In assert, the experssion involves side-effects. So the behavior of the code becomes different in case of debug version and the release version thus leading to a subtle bug.

Rule to Remember:

Don't use expressions that have side-effects in assert statements.

160)
```
void main()
{
int *i = 0x400;  // i points to the address 400
*i = 0;             // set the value of memory location pointed by i;
}
```

Answer:

Undefined behavior

Explanation:

The second statement results in undefined behavior because it points to some location whose value may not be available for modification. *This type of pointer in which the non-availability of the implementation of the referenced location is known as 'incomplete type'.*

161)
```
#define assert(cond) if(!(cond)) \
  (fprintf(stderr, "assertion failed: %s, file %s, line %d \n",#cond,\
  __FILE__,__LINE__), abort())

void main()
{
int i = 10;
if(i==0)
   assert(i < 100);
else
   printf("This statement becomes else for if in assert macro");
}
```

Answer:

No output

Explanation:

The else part in which the printf is there becomes the else for if in the assert macro. Hence nothing is printed.

The solution is to use conditional operator instead of if statement,

```
#define assert(cond) ((cond)?(0): (fprintf (stderr, "assertion failed: \ %s, file %s, line %d \n",#cond, __FILE__,__LINE__), abort()))
```

Note:

However this problem of "matching with nearest else" cannot be solved by the usual method of placing the if statement inside a block like this,

```
#define assert(cond) { \
if(!(cond)) \
  (fprintf(stderr, "assertion failed: %s, file %s, line %d \n",#cond,\
 __FILE__,__LINE__), abort()) \
}
```

162) Is the following code legal?

```
struct a
   {
int x;
 struct a b;
   }
```

Answer:

No

Explanation:

Is it not legal for a structure to contain a member that is of the same type as in this case. Because this will cause the structure declaration to be recursive without end.

163) Is the following code legal?

```
struct a
   {
int x;
struct a *b;
   }
```

Answer:

Yes.

Explanation:

*b is a pointer to type struct a and so is legal. The compiler knows, the size of the pointer to a structure even before the size of the structure is determined(as you know the pointer to any type is of same size). This type of structures is known as 'self-referencing' structure.

164) Is the following code legal?

```
typedef struct a
   {
int x;
 aType *b;
   }aType
```

Answer:

No

Explanation:

The typename aType is not known at the point of declaring the structure (forward references are not made for typedefs).

165) Is the following code legal?

```
typedef struct a aType;
struct a
{
int x;
aType *b;
};
```

Answer:

Yes

Explanation:

The typename aType is known at the point of declaring the structure, because it is already typedefined.

166) Is the following code legal?

```
void main()
{
typedef struct a aType;
aType someVariable;
struct a
{
int x;
    aType *b;
 };
}
```

Answer:

No

Explanation:

When the declaration,

```
typedef struct a aType;
```

is encountered body of struct a is not known. This is known as 'incomplete types'.

167)

```
void main()
{
printf("sizeof (void *) = %d \n", sizeof( void *));
printf("sizeof (int *)    = %d \n", sizeof(int *));
printf("sizeof (double *)  = %d \n", sizeof(double *));
printf("sizeof(struct unknown *) = %d \n", sizeof(struct unknown *));
}
```

Answer :

```
sizeof (void *) = 2
sizeof (int *)    = 2
sizeof (double *)  =  2
sizeof(struct unknown *) =  2
```

Explanation:

The pointer to any type is of same size.

168) char inputString[100] = {0};
To get string input from the keyboard which one of the following is better?
1) gets(inputString)
2) fgets(inputString, sizeof(inputString), fp)

Answer & Explanation:

The second one is better because gets(inputString) doesn't know the size of the string passed and so, if a very big input (here, more than 100 chars) the charactes will be written past the input string. When fgets is used with stdin performs the same operation as gets but is safe.

169) Which version do you prefer of the following two,
1) printf("%s",str); // or the more curt one
2) printf(str);

Answer & Explanation:

Prefer the first one. If the str contains any format characters like %d then it will result in a subtle bug.

170)
```
void main()
{
int i=10, j=2;
int *ip= &i, *jp = &j;
int k = *ip/*jp;
printf("%d",k);
}
```

Answer:

Compiler Error: "Unexpected end of file in comment started in line 5".

Explanation:

The programmer intended to divide two integers, but by the "maximum munch" rule, the compiler treats the operator sequence / and * as /* which happens to be the starting of comment. To force what is intended by the programmer,

```
int k = *ip/ *jp;
// give space explicity separating / and *
//or
int k = *ip/(*jp);
// put braces to force the intention
```

will solve the problem.

171)
```
void main()
{
char ch;
for(ch=0;ch<=127;ch++)
printf("%c   %d \n", ch, ch);
}
```

Answer:

Implementaion dependent

Explanation:

The char type may be signed or unsigned by default. If it is signed then ch++ is executed after ch reaches 127 and rotates back to -128. Thus ch is always smaller than 127.

172) Is this code legal?

```
int *ptr;
ptr = (int *) 0x400;
```

Answer:

Yes

Explanation:

The pointer ptr will point at the integer in the memory location 0x400.

173)
```
main()
{
char a[4]="HELLO";
printf("%s",a);
}
```

Answer:

Compiler error: Too many initializers

Explanation:

The array a is of size 4 but the string constant requires 6 bytes to get stored.

174)
```
main()
{
char a[4]="HELL";
printf("%s",a);
}
```

Answer:

HELL%@!~@!@???@~~!

Explanation:

The character array has the memory just enough to hold the string “HELL” and doesnt have enough space to store the terminating null character. So it prints the HELL correctly and continues to print garbage values till it accidentally comes across a NULL character.

175)
```
main()
{
        int a=10,*j;
        void *k;
        j=k=&a;
        j++;
        k++;
        printf("\n %u %u ",j,k);
}
```

Answer:

Compiler error: Cannot increment a void pointer

Explanation:

Void pointers are generic pointers and they can be used only when the type is not known and as an intermediate address storage type. No pointer arithmetic can be done on it and you cannot apply indirection operator (*) on void pointers.

176)
```
main()
        {
                extern int i;
        {       int i=20;
         {
          const volatile unsigned i=30; printf("%d",i);
         }
                printf("%d",i);
        }
         printf("%d",i);
        }
int i;
```

177) Printf can be implemented by using _____ list.

Answer:

Variable length argument lists

178)
```
char *someFun()
{
char *temp = "string constant";
return temp;
}
int main()
{
puts(someFun());
}
```

Answer:

string constant

Explanation:

The program suffers no problem and gives the output correctly because the character constants are stored in code/data area and not allocated in stack, so this doesn't lead to dangling pointers.

179)
```
char *someFun1()
{
char temp[ ] = "string";
return temp;
}
char *someFun2()
{
char temp[ ] = {'s', 't','r','i','n','g'};
```

```
return temp;
}
int main()
{
puts(someFun1());
puts(someFun2());
}
```

Answer:

Garbage values.

Explanation:

Both the functions suffer from the problem of dangling pointers. In someFun1() temp is a character array and so the space for it is allocated in heap and is initialized with character string "string". This is created dynamically as the function is called, so is also deleted dynamically on exiting the function so the string data is not available in the calling function main() leading to print some garbage values. The function someFun2() also suffers from the same problem but the problem can be easily identified in this case.