**C Sample Question**

**Note :** All the C sample 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).

Following are some C sample questions.

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

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".

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].

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 (== , >, <, <=, >=,!= ).

 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.

 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.

 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 .

 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

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.

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.

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.

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.

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.

#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

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 â€˜ >â€™ symbol. ! is a unary logical operator. !i (!10) is 0 (not of true is false). 0>14 is false (zero).

#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).

#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.

#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

#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.

main()

{

printf("\nab");

printf("\bsi");

printf("\rha");

}

**Answer:**

hai

**Explanation:**

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

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.

#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

 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 â€œibj!gsjfoetâ€ and since p reaches '\0' and p1 points to p thus p1doesnot print anything.

 #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.

 #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

 main()

{

printf("%p",main);

}

**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.

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).

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.

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

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.

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.

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.

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.

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

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.

#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.

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.

#define f(g,g2) g##g2

main()

{

int var12=100;

printf("%d",f(var,12));

 }

**Answer:**

100

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).

#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");

#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

#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.

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.

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.

main()

{

 extern out;

 printf("%d", out);

}

 int out=100;

**Answer:**

100

**Explanation:**

This is the correct way of writing the previous program.

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:

declare void show() in main() .

define show() before main().

declare extern void show() before the use of show().

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.

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.

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.

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.

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'.

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'.

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.