POINTERS

Solutions

1. What is the output of following program?

```
# include <stdio.h>

void fun(int x)
{
x = 30;
}

int main()
{
int y = 20;
fun(y);
printf("%d", y);
return 0;
}
```

(a) 30 (b) 20
(c) Compiler Error (d) Runtime Error

Solution: Option (b)

Explanation:
Parameters are always passed by value in C. Therefore, in the above code, value of y is not modified using the function fun(). So how do we modify the value of a local variable of a function inside another function. Pointer is the solution to such problems. Using pointers, we can modify a local variable of a function inside another function. See the next question.

Note that everything is passed by value in C. We only get the effect of pass by reference using pointers.

2. Output of the following program?

```
# include <stdio.h>

void fun(int *ptr)
```

```
{
  *ptr = 30;
}

int main()
{
  int y = 20;
  fun(&y);
  printf("%d", y);
return 0;
}
```

(a) 20 (b) 30
(c) Compiler Error (d) Runtime Error

Solution: Option (b)

Explanation:
The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement ‘*ptr = 30’, value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement ‘fun(&y)’, address of y is passed so that y can be modified using its address.

3. Output of following program?

```
#include <stdio.h>

int main()
{
   int *ptr;
   int x;
   ptr = &x;
   *ptr = 0;
   printf(" x = %d\n", x);
   printf(" *ptr = %d\n", *ptr);
   *ptr += 5;
   printf(" x  = %d\n", x);
   printf(" *ptr = %d\n", *ptr);
   (*ptr)++;
```

```
    printf(" x = %d\n", x);
    printf(" *ptr = %d\n", *ptr);
    return 0;
}
```

(a) x = 0
   *ptr = 0
   x = 5
   *ptr = 5
   x = 6
   *ptr = 6

(b) x = garbage value
   *ptr = 0
   x = garbage value
   *ptr = 5
   x = garbage value
   *ptr = 6

(c) x = 0
   *ptr = 0
   x = 5
   *ptr = 5
   x = garbage value
   *ptr = garbage value

(d) x = 0
   *ptr = 0
   x = 0
   *ptr = 0
   x = 0
   *ptr = 0

Solution: Option (a)

Explanation:
See the comments below for explanation.

```
int *ptr;  /* Note: the use of * here is not for dereferencing,
              it is for data type int */

int x;
ptr = &x;   /* ptr now points to x (or ptr is equal to address of x) */
*ptr = 0;   /* set value ate ptr to 0 or set x to zero */
printf(" x = %d\n", x);   /* prints x =  0 */
printf(" *ptr = %d\n", *ptr);  /* prints *ptr =  0 */
*ptr += 5;        /* increment the value at ptr by 5 */
printf(" x  = %d\n", x);  /* prints x = 5 */
printf(" *ptr = %d\n", *ptr); /* prints *ptr =  5 */
(*ptr)++;         /* increment the value at ptr by 1 */
printf(" x  = %d\n", x);  /* prints x = 6 */
printf(" *ptr = %d\n", *ptr);  /* prints *ptr =  6 */
```

4. Consider a compiler where int takes 4 bytes, char takes 1 byte and pointer takes 4 bytes.

```
#include <stdio.h>

int main()

{
int arri[] = {1, 2 ,3};
int *ptri = arri;

char arrc[] = {1, 2 ,3};
char *ptrc = arrc;

printf("sizeof arri[] = %d ", sizeof(arri));
printf("sizeof ptri = %d ", sizeof(ptri));
printf("sizeof arrc[] = %d ", sizeof(arrc));
printf("sizeof ptrc = %d ", sizeof(ptrc));

return 0;
}
```

(a) sizeof arri[] = 3
sizeof ptri = 4
sizeof arrc[] = 3
sizeof ptrc = 4

(b) sizeof arri[] = 12
sizeof ptri = 4
sizeof arrc[] = 3
sizeof ptrc = 1

(c) sizeof arri[] = 3
sizeof ptri = 4
sizeof arrc[] = 3
sizeof ptrc = 1

(d) sizeof arri[] = 12
sizeof ptri = 4
sizeof arrc[] = 3
sizeof ptrc = 4

Solution: Option (d)

Explanation:
Size of an array is number of elements multiplied by the type of element that is why we get sizeof arri as 12 and sizeof arrc as 3. Size of a pointer is fixed for a compiler. All pointer types take same number of bytes for a compiler. That is why we get 4 for both ptri and ptrc.

5. Assume that float takes 4 bytes, predict the output of following program.

```
#include <stdio.h>

int main()
{
```

```
float arr[5] = {12.5, 10.0, 13.5, 90.5, 0.5};
float *ptr1 = &arr[0];
float *ptr2 = ptr1 + 3;
printf("%f ", *ptr2);
printf("%d", ptr2 – ptr1);
return 0;
}
```

(a) 90.500000
3

(b) 90.500000
12

(c) 10.000000
12

(d) 0.500000
3

Solution: Option (a)

Explanation:
When we add a value x to a pointer p, the value of the resultant expression is p + x*sizeof(*p) where sizeof(*p) means size of data type pointed by p. That is why ptr2 is incremented to point to arr[3] in the above code. Same rule applies for subtraction. Note that only integral values can be added or subtracted from a pointer. We can also subtract or compare two pointers of same type.

7.

```
#include<stdio.h>

int main( )
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr1 = arr;
   int *ptr2 = arr + 5;
   printf("Number of elements between two pointer are: %d.",
          (ptr2 - ptr1));
  printf("Number of bytes between two pointers are: %d",
          (char*)ptr2 - (char*) ptr1);
 return 0;
}
```

Assume that an int variable takes 4 bytes and a char variable takes 1 byte

(a) Number of elements between two pointers are: 5.
    Number of bytes between two pointers are: 20
(b) Number of elements between two pointers are: 20.
    Number of bytes between two pointers are: 20
(c) Number of elements between two pointers are: 5.
    Number of bytes between two pointers are: 5
(d) Compiler Error
(e) Runtime Error

Solution: Option (a)

Explanation:
Array name gives the address of first element in array. So when we do '*ptr1 = arr;', ptr1 starts holding the address of element 10. 'arr + 5′ gives the address of 6th element as arithmetic is done using pointers. So 'ptr2-ptr1′ gives 5. When we do '(char *)ptr2′, ptr2 is type-casted to char pointer and size of character is one byte, pointer arithmetic happens considering character pointers. So we get 5*sizeof(int)/sizeof(char) as a difference of two pointers.

7.

```
#include<stdio.h>

int main()
{
char *x;
int a = 512;
x = (char *) &a;
x[0] = 1;
x[1] = 2;
printf("%d\n",a);
return 0;
}
```

What is the output of above program?

(a) Machine dependent
(b) 513
(c) 258
(d) Compiler Error

Solution: Option (a)

Explanation:
Output is 513 in a little endian machine. To understand this output, let integers be stored using 16 bits.

In a little endian machine, when we do x[0] = 1 and x[1] = 2, the number a is changed to 0000000100000010 which is representation of 513 in a little endian machine.

8.

```
 #include<stdio.h>

int main()

{
  char *ptr = "ravindrababuravula";
  printf("%c\n", *&*&*ptr);
return 0;
}
```

(a) Compiler Error
(b) Garbage Value
(c) Runtime Error
(d) r

Solution: Option (d)

Explanation:
The operator * is used for dereferencing and the operator & is used to get the address. These operators cancel out effect of each other when used one after another. We can apply them alternatively any no. of times. In the above code, ptr is a pointer to first character of string r. *ptr gives us r, &*ptr gives address of r, *&*ptr again r, &*&ptr address of r, and finally *&*&ptr gives 'r'.

9.

```
#include<stdio.h>

void fun(int arr[])
{
int i;
int arr_size = sizeof(arr)/sizeof(arr[0]);
for (i = 0; i < arr_size; i++)
```

```
   printf("%d ", arr[i]);
}

int main()
{
  int i;
  int arr[4] = {10, 20 ,30, 40};
  fun(arr);
  }
```

(a) 10 20 30 40	(b) 10
(c) 10 20	(d) Nothing

Solution: Option (c)

Explanation:
In C, array parameters are always treated as pointers.

10. The reason for using pointers in a C-program is

(a) Pointers allow different functions to share and modify their local variables.
(b) To pass large structures so that complete copy of the structure can be avoided.
(c) Pointers enable complex “linked” data structures like linked lists and binary trees.
(d) All of the above.

Solution: Option (d)

Explanation:
See below explanation

(a) With pointers, address of variables can be passed different functions can use this address to access the variables.

(b) When large structure variables passed or returned, they are copied as everything is passed and returned by value in C. This can be costly with structure containing large data. To avoid this copying of large variables, we generally use pointer for large structures so that only address is copied.

(c) With pointers, we can implement “linked” data structures. Java uses reference variables to implement these data structures. Note that C doesn’t support reference variables.

11.

```
#include<stdio.h>

void f(int *p, int *q)
{
  p = q;
  *p = 2;
}

int i = 0, j = 1;
int main()
{
  f(&i, &j);
  printf("%d %d \n", i, j);
  getchar();
  return 0;
}
```

(a) 2 2 (b) 2 1
(c) 0 1 (d) 0 2

Solution: Option (d)

Explanation:
See below f() with comments for explanation.

```
/* p points to i and q points to j */
void f(int *p, int *q)
{
p = q;    /* p also points to j now */
*p = 2;   /* Value of j is changed to 2 now */
}
```

12. Consider this C code to swap two integers and these five statements after it:

```
void swap(int *px, int *py)
{
    *px = *px - *py;
    *py = *px + *py;
```

```
   *px = *py - *px;
}
```

S1: will generate a compilation error
S2: may generate a segmentation fault at runtime depending on the arguments passed
S3: correctly implements the swap procedure for all input pointers referring to integers stored in memory locations accessible to the process
S4: implements the swap procedure correctly for some but not all valid input pointers
S5: may add or subtract integers and pointers.

(a) S1
(b) S2 and S3
(c) S2 and S4
(d) S2 and S5

Solution: Option (c)

Explanation:
S2: May generate segmentation fault if value at pointers px or py is constant or px or py points to a memory location that is invalid

S4: May not work for all inputs as arithmetic overflow can occur

13.

```
int f(int x, int *py, int **ppz)
{
 int y, z;
 **ppz += 1;
 z  = **ppz;
 *py += 2;
 y = *py;
 x += 3;
 return x + y + z;
}

void main()
{
 int c, *b, **a;
 c = 4;
 b = &c;
 a = &b;
```

```
  printf("%d ", f(c, b, a));
  return 0;
}
```

(a) 18 (b) 19
(c) 21 (d) 22

Solution: Option (b)

Explanation:

c is an integer variable . b is a pointer to an integer(pointing to c) . a is pointer to a pointer pointing to the pointer variable b, Arguments are passed using call by reference.

14. Predict the output of following program

```
#include<stdio.h>

int main()
{
    int a = 12;
    void *ptr = (int *)&a;
    printf("%d", *ptr);
    getchar();
    return 0;
}
```

(a) 12 (b) Compiler Error
(c) Run Time Error (d) 0

Solution: Option (b)

Explanation:
There is compiler error in line “printf(“%d”, *ptr);”.
void * type pointers cannot be de-referenced. We must type cast them before de-referencing.

The following program works fine and prints 12.

```
#include<stdio.h>

int main()
{
    int a = 12;
    void *ptr = (int *)&a;
    printf("%d", *(int *)ptr);
    getchar();
    return 0;
}
```

15.

```
#include<stdio.h>

void swap (char *x, char *y)
{
    char *t = x;
    x = y;
    y = t;
}

int main()
{
  char *x = "ravindrababu";
  char *y = "ravula";
  char *t;
  swap(x, y);
  printf("(%s, %s)", x, y);
  t = x;
  x = y;
  y = t;
  printf("\n(%s, %s)", x, y);
  return 0;
}
```

(a) (ravindrababu,ravula)
(ravula,ravindrababu)

(b) (ravula,ravindrababu)
(ravindrababu,ravula)

(c) (ravindrababu,ravula)
(ravindrababu,ravula)

(d) (ravula,ravindrababu)
(ravula,ravindrababu)

Solution: Option (a)

16.

```
#include <stdio.h>

int main()
{
   int arr[] = {1, 2, 3, 4, 5};
   int *p = arr;
   ++*p;
   p += 2;
   printf("%d", *p);
   return 0;
}
```

(a) 2
(b) 3
(c) 4
(d) Compilation Error

Solution: Option (b)

Explanation:
The expression ++*p is evaluated as “++(*p)” . So it increments the value of first element of array (doesn’t change the pointer p).

When p += 2 is done, p is changed to point to third element of array.

17.

```
#include <stdio.h>

void f(char**);
int main()
{
   char *argv[] = { "ab", "cd", "ef", "gh", "ij", "kl" };
   f(argv);
   return 0;
}

void f(char **p)
{
   char *t;
   t = (p += sizeof(int))[-1];
   printf("%s\n", t);
}
```

(a) ab (b) cd
(c) ef (d) gh

Solution: Option (d)

18. What does the following C-statement declare?

```
int ( * f) (int * ) ;
```

(a) A function that takes an integer pointer as argument and returns an integer.
(b) A function that takes an integer as argument and returns an integer pointer.
(c) A pointer to a function that takes an integer pointer as argument and returns an integer.
(d) A function that takes an integer pointer as argument and returns a function pointer.

Solution: Option (c)

19.

```
#include <stdio.h>
#define print(x) printf("%d ", x)

int x;
void Q(int z)
{
   z += x;
   print(z);
  }

void P(int *y)
{
   int x = *y + 2;
   Q(x);
   *y = x - 1;
   print(x);
}

main(void)
{
   x = 5;
   P(&x);
```

```
    print(x);
}
```

The output of this program is

(a) 12 7 6
(b) 22 12 11
(c) 14 6 6
(d) 7 6 6

Solution: Option (a)

Explanation:
x is global so first x becomes 5 by the first line in main(). Then main() calls P() with address of x.

// in main(void)
x = 5 // Change global x to 5
P(&x)
P() has a local variable named 'x' that hides global variable.
P() theb calls Q() by passing value of local 'x'.
// In P(int *y)
int x = *y + 2; // Local x = 7
Q(x);
In Q(int z), z uses x which is global
// In Q(int z)
z += x; // z becomes 5 + 7
printz(); // prints 12
After end of Q(), control comes back to P().
In P(), *y (y is address of global x) is changed to x –1 (x is local to P()).
// Back in P()
*y = x - 1; // *y = 7-1
print(x); // Prints 7
After end of Q(), control comes back to main(). In main(), global x is printed.
// Back in main()
print(x); // prints 6 (updated in P()
//   by *y = x - 1)

20.

```
#include<stdio.h>
void fun(int *p)
```

```
{
  int q = 10;
  p = &q;
}

int main()
{
  int r = 20;
  int *p = &r;
  fun(p);
  printf("%d", *p);
  return 0;
}
```

(a) 10 (b) 20
(c) Compiler error (d) Runtime Error

Solution: Option (b)

Explanation:
Inside fun(), q is a copy of the pointer p. So if we change q to point something else then p remains unaffected. If we want to change a local pointer of one function inside another function, then we must pass pointer to the pointer. By passing the pointer to the pointer, we can change pointer to point to something else. See the following program as an example.

```
void fun(int **pptr)
{
  static int q = 10;
  *pptr = &q;
}

int main()
{
  int r = 20;
  int *p = &r;
  fun(&p);
  printf("%d", *p);
  return 0;
}
```

In the above example, the function fun() expects a double pointer (pointer to a pointer to an integer).

Fun() modifies the value at address pptr. The value at address pptr is pointer p as we pass address of p to fun(). In fun(), value at pptr is changed to address of q. Therefore, pointer p of main() is changed to point to a new variable q.

Also, note that the program won't cause any out of scope problem because q is a static variable. Static variables exist in memory even after functions return. For an auto variable, we might have seen some unexpected output because auto variable may not exist in memory after functions return.