RRB-JE 2024

Railway Recruitment Board

Junior Engineer Examination

Electronics Engineering

Computer Programming

Well Illustrated **Theory** *with*
Solved Examples and **Practice Questions**

MADE EASY PUBLICATIONS

Computer Programming

Contents

❍❍❍❍

Chapter 1

Programming Methodology

1.1 C Variable

A variable is named location of data. In other word we can say variable is container of data. In real world you have used various type of containers for specific purpose. For *example* you have used suitcase to store clothes, match box to store match sticks etc. In the same way variables of different data type is used to store different types of data. For *example* integer variables are used to store integers char variables is used to store characters etc. On the basis of type of data a variable will store, we can categorize the all C variable in three groups.

(a) Variables which can store only one data at time.
Example*:* Integer variables, char variables, pointer variables etc.

(b) Variables which can store more than one data of similar type at a time.
Example: Array variables

(c) Variables, which can store more than one value of dissimilar type at a time.
Example: Structure or union variables.

Properties of C Variable

Every variable in C have three most fundamental attributes. They are Name, Value and Address.

Name of a Variable: Every variable in C has its own name. A variable without any name is not possible in C. Most important properties of variables name are its unique names.

- No two variables in C can have same name with same visibility.
- It is possible that two variable with same name but different visibility. In this case variable name can access only that variable which is more local. In C there is not any way to access global variable if any local variable is present of same name.

Example - 1.1 What will be output of following program?

Code	Solution

`#include <stdio.h>`
`int main(){`
`  auto int a=5;  //Visibility is within main block`
`  static int a=10;  //Visibility is within main block`
`  /* Two variables of same name */`
`  printf("%d",a);`
`  return 0;`
`}`
	Output: compilation error
	Here variable 'a' is declared twice in main block. So it generate compile time error.

Example - 1.2 What will be output of following program?

Code	Solution

`#include <stdio.h>`
`int a=50;  //Visibility is in whole program`
`int main()`
`{  int a=10;  //Visibility within main block`
`  printf("%d",a);`
`  return 0;`
`}`
	Output: 10
	Here variable 'a' has different visibility at two different places. So printf will print main block variable value.

NOTE

In C any name is called identifier. This name can be variable name, function name, enum constant name, micro constant name, goto label name, any other data type name like structure, union, enum names or typedef name.

Identifier Naming Rule in C

In C any name is called identifier. This name can be variable name, function name, enum constant name, micro constant name, goto label name, any other data type name like structure, union, enum names or typedef name.

Rule 1: Name of identifier includes alphabets, digit and underscore.

Valid name: world, addition23, sum_of_number etc.

Invalid name: factorial#, avg value, display*number etc.

Rule 2: First character of any identifier must be either alphabets or underscore.

Valid name: _calculate, _5,a_, __ etc.

Invalid name: 5_, 10_function, 123 etc.

Rule 3: Name of identifier cannot be any keyword of C program.

Valid name: INT, FLOAT, etc.

Invalid name: int, float, enum etc.

Rule 4: Name of function cannot be global identifier.

Valid name: __TOTAL__, __NAME__ , __TINY__etc.

Invalid name: __TIME__ ,__DATE__ , __FILE__ ,__LINE__ ,__STDC__, __TINY__, __SMALL__, __COMPACT__, __LARGE__, __HUHE__, __CDECL__, __PASCAL__, __MSDOS__, __TURBOC__

Rule 5: Name of identifier cannot be register Pseudo variables

Rule 6: Name of identifier cannot be exactly same as of name of function within the scope of the function.

Rule 7: Name of identifier is case sensitive i.e. num and Num are two different variables.

Rule 8: Only first 32 characters are significant of identifier name.

Example: abcdefghijklmnopqrstuvwxyz123456aaa.

Rule 9: Identifier name cannot be exactly same as constant name which have been declared in header file of C and you have included that header files.

- Variable name cannot be exactly same as function name which have been declared any header file of C and we have included that header file in our program and used that function also in the program.
- Identifier name in C can be exactly same as data type which has been declared in any header of C.

Example - 1.3 What will be output of following program?

Code	Solution

`#include <stdio.h>`
`int main()`
`{`
`  int float=5;`
`  printf("%d",float);`
`return 0;`
`}`
	Output: compilation error
	Here we have used float as variable name. So it will give compile time error.

1.2 Operators in C

Bitwise Operators in C

In C bitwise operators work at bit-level. They are as follows:

(*i*) **& (bitwise AND):** '&' operator takes two numbers as input as input and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

(*ii*) **| (bitwise OR):** 'l' operator takes two numbers as input and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

(*iii*) **∧ (bitwise XOR):** '&' operator takes two numbers as input and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

(*iv*) **<< (left shift):** '<<' operator takes two numbers as input, left shifts the bits of first number, the second number decides the number of places to shift.

(*v*) **>> (right shift):** '<<' operator takes two numbers as input, right shifts the bits of first number, the second number decides the number of places to shift.

(*vi*) **~(bitwise NOT):** '~' operator takes one number as input and inverts all bits of it.

Example - 1.4 What will be output of following C code?

Code	Solution

`#include<stdio.h>`
`int main()`
`{`
`  char a = 7, b = 11;`
`  printf("a = %d, b = %d\n", a, b);`
`  printf("a & b = %d\n", a & b);`
`  printf("a|b = %d\n", a|b);`
`  printf("a ∧b = %d\n", a ∧b);`
`  printf("~a = %d\n", a = ~a);`
`  printf("a << 1 = %d\n", a << 1);`
`  printf("b >> 1 = %d\n", b >> 1);`
`  return 0;`
`}`
	Output :
	a = 7, b = 11
	a & b = 3
	a|b = 15
	a ∧ b = 12
	~a = 248
	a << 1 = 22
	b >> 1 = 5

Some properties of bitwise operators are as follows:

1. The **left shift and right shift operators should not be used for negative numbers**. The result of '<<' and '>>' is undefined behaviour if any of the operands is a negative number.

2. If the number is shifted more than the size of integer, the behaviour is undefined. For example, 2 << 33 is undefined if integers are stored using 32 bits.
3. **The bitwise operators should not be used in-place of logical operators.**
   The result of logical operators (&&, II and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1.

Example - 1.5 What will be output of following C code?

Code	Solution

`int main()`
`{ int x = 2, y = 5;`
`(x&y)? printf("True "):printf("False ");`
`(x&&y)? printf("True "):printf("False ");`
`return 0;`
`}`
	Output:
	False True

4. The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively but it will works only if numbers are positive.

Example - 1.6 What will be output of following C code?

Code	Solution

`int main()`
`{ int x = 21;`
`printf ("x << 1 = %d\n", x << 1);`
`printf ("x >> 1 = %d\n", x >> 1);`
`return 0;`
`}`
	Output : 42 10

5. **The & operator can be used to quickly check if a number is odd or even**
   The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

Example - 1.7 What will be output of following C code?

Code	Solution

`int main()`
`{ int x = 21;`
`(x & 1)? printf("Odd"): printf("Even");`
`return 0;`
`}`
	Output: Odd

Properties of Operator Precedence and Associativity in C

Precedence: In an expression there are multiple operators. Precedence plays most important role, in order to decide the evaluation of there operators. **Example:** $1 + 2 \times 3$ is calculated as $1+(2 \times 3)$ and not as $(1 + 2) \times 3$.

Associativity: In an expression some times two or more operators of same precedence appear, order of evaluation is decided by associativity. Associativity can be either Left to Right or Right to Left. **Example** '×' and '/' have same precedence and their associativity is Left to Right, so the expression "1000/100 × 10" is treated as "(1000/100) × 10".

1. Associativity is compiler dependent.

Code	

`int x = 0;`
`int main()`
`{ int p = f1() + f2();`
`printf("%d ", x);`
`return 0;`
`}`
	`int f1()`
	`{ x = 5;`
	`return x;`
	`}`
	`int f2()`
	`{ x = 10;`
	`return x;`
	`}`

2. **Precedence and associativity of postfix ++ and prefix ++ are different:**
   Precedence of postfix ++ is more than prefix ++, their associativity is also different. Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left.
3. Comma has the least precedence among all operators.

Example - 1.8 What will be output of following C code?

Code	Solution

`#include<stdio.h>`
`int main()`
`{ int a;`
`a = 1, 2, 3;`
`printf("%d", a);`
`return 0;`
`}`
	Output :
	Evaluated as (a = 1), 2, 3

Operators Precedence Table

S. No.	Operator	Associativity
1.	(,), [,] , → , ·	Left to right
2.	!, ~, ++, – –, + (unary), – (unary), *, &, sizeof	Right to left
3.	*, / , %	Left to right
4.	+ , –	Right to left
5.	<< , >>	Left to right
6.	<, <=, >, >=	Right to left
7.	==, !=	Left to right
8.	&	Right to left
9.	^	Left to right
10.	|	Right to left
11.	&&	Left to right
12.	||	Right to left
13.	? :	Left to right
14.	=, +=, –=, *=, /=, %=, &=, ^=, |=, <<=, >>=	Right to left
15.	,	Left to right

(Highest) ↕ (Lowest)

Arithmetic Operators

Arithmetic Operators perform arithmetic operations on operands.

- **Addition:** The '+' operator does addition of two operands i.e., $x + y$.
- **Subtraction:** The '–' operator does subtraction of two operands i.e., $x - y$.
- **Multiplication:** The '*' operator does multiplication of two operands i.e., $x \times y$.
- **Division:** The '/' operator does division of the first operand by the second i.e., x/y.
- **Modulus:** The '%' operator returns the remainder when first operand is divided by the second i.e., $x\%y$.

Example - 1.9 What will be output of following C code?

Code	Solution

`#include<stdio.h>`
`int main()`
`{ int a = 10, b = 4, op;`
`//printing a and b`
`printf("a is %d and b is %d\n", a, b);`
`op = a+b; printf("a+b is %d\n", op);`
`op = a-b; printf("a-b is %d\n", op);`
`op = a*b; printf("a*b is %d\n", op);`
`op = a/b; printf("a/b is %d\n", op);`
`op = a%b; printf("a%b is %d\n", op);`
`return 0;`
`}`
	Output :
	a is 10 and b is 4
	a + b is 14
	a – b is 6
	a*b is 40
	a/b is 2
	a%b is 2

Unary Arithmetic Operators

- **Increment:** The '++' operator is used to increment the value of an integer. When placed before the variable name called **pre-increment** i.e., ++ a (value increment and update immediately).
  When it is placed after the variable name called **post-increment** i.e., a++. Its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement.
- **Decrement:** The '––' operator is used to decrement the value of an integer. When placed before the variable name called **pre-decrement** i.e., –– a (value decremented and update immediately).
  When it is placed after the variable name called **post-decrement** i.e., a ––. Its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement.

Example - 1.10 What will be output of following C code?

Code	Solution

`#include<stdio.h>`
`int main()`
`{ int a = 10, op;`
`op = a++;`
`printf("a is %d and op is %d\n", a, op);`
`op = a--;`
`printf("a is %d and op is %d\n", a, op);`
`op = ++a;`
`printf("a is %d and op is %d\n", a, op);`
`op = --a;`
`printf("a is %d and op is %d\n", a, op);`
`return 0;`
`}`
	Output :
	a is 11 and op is 10
	a is 10 and op is 11
	a is 11 and op is 11
	a is 10 and op is 10

Relational Operators

Relational operators are used for comparison of two values.

(*i*) **'=='** operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false i.e., 5 == 5 will return true. But 5 == 6 returns false.

(*ii*) **'!='** operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise it returns false i.e., 5! = 5 will return false. But 5! = 6 returns true.

(*iii*) **'>'** operator checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise it returns false i.e., 11 > 10 will return true. But 9 > 10 returns false.

(*iv*) **'<'** operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise it returns false i.e., 11 < 10 will return false. But 9 < 10 returns true.

(*v*) **'>='** operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true. Otherwise it returns false i.e., 10 >= 10 will return true. But 9 >= 10 returns false.

(*vi*) **'<='** operator checks whether the first operand is lesser than or equal to the second operand. If so, it returns true. Otherwise it returns false i.e., 10 <= 10 will also return true. But 11 <= 10 returns false.

Example - 1.11 What will be output of following C code?

Code	Solution

`#include <stdio.h>`
`int main()`
`{`
`  int a=6, b=5;`
`  if (a > b)`
`  printf("a is greater than b\n");`
`  else printf("a is less than or equal to b\n");`
`  if (a >= b)`
`  printf("a is greater than or equal to b\n");`
`  else printf("a is lesser than b\n");`
`  if (a < b)`
`  printf("a is less than b\n");`
`  else printf("a is greater than or equal to b\n");`
`  if (a <= b)`
`  printf("a is lesser than or equal to b\n");`
`  else printf("a is greater than b\n");`
`  if (a == b)`
`  printf("a is equal to b\n");`
`  else printf("a and b are not equal\n'');`
`  if (a! = b)`
`  printf("a is not equal to b\n");`
`  else printf("a is equal b\n");`
`  return 0;`
`}`
	Output :
	a is greater than b
	a is greater than or equal to b
	a is greater than or equal to b
	a is greater than b
	a and b are not equal
	a is not equal to b

Logical Operators

They are used to combine two or more conditions and produce output in form of true (1) and false (0).

(*i*) **Logical AND:** The '&&' operator returns true when both the conditions are satisfied. Otherwise it returns false i.e., a && b returns true when both a and b are true.

(*ii*) **Logical OR:** The 'll' operator returns true when one (or both) of the conditions is satisfied. Otherwise it returns false i.e., a || b returns true if one of a or b is true. It returns true when both a and b are true.

(*iii*) **Logical NOT:** The '!' operator returns true if the condition is not satisfied. Otherwise it returns false i.e., !a returns true if a is false, i.e. when a = 0.

Example - 1.12 What will be output of following C code?

Code	Solution

`#include <stdio.h>`
`int main()`
`{ int a = 5, b = 2, c = 5, d = 10;`
`if (a>b && c == d)`
`printf("a is greater than b AND c is equal to d\n");`
`else printf("Logical AND condition not satisfied\n");`
`if (a>b || c==d)`
`printf("a is greater than b Logical OR c is equal to d\n");`
`else printf("Neither a is greater than b nor c is equal " " to d\n");`
`if (!a)`
`printf("a is zero\n");`
`else printf("a is not zero");`
`return 0;`
`}`
	Output
	Logical AND condition not satisfied
	a is greater than b Logical OR c is equal to d
	a is not zero.

Short-Circuiting in Logical Operators

In case of **logical AND**, the second operand is not evaluated if first operand is false.

Example - 1.13 What will be output of following C code?

Code	Solution

`#include <stdio.h>`
`#include <stdbool.h>`
`int main()`
`{ int a = 5, b = 4;`
`bool op = ((a == b) && printf("MADEEASY"));`
`return 0;`
`}`
	Output :
	bool op will return 0 because of (a == b) is false.
	So printf statement is not evaluated.

`#include <stdio.h>`
`#include <stdbool.h>`
`int main()`
`{ int a = 5, b = 4;`
`bool op = ((a! = b) && printf("MADEEASY"));`
`return 0;`
`}`
	Output :
	bool op will return 1 because of (a != b) is true.
	So printf statement is evaluated.

In case of **logical OR**, the second operand is not evaluated if first operand is true.

Example - 1.14 What will be output of following C code?

Code	Solution

`#include <stdio.h>`
`#include <stdbool.h>`
`int main()`
`{ int a = 5, b = 4;`
`bool op = ((a! = b)|| printf("MADEEASY"));`
`return 0;`
`}`
	Output :
	bool op will return 1 because of (a != b) is true.
	So printf statement is not evaluated.

`#include <stdio.h>`
`#include <stdbool.h>`
`int main()`
`{ int a = 5, b = 4;`
`bool op = ((a == b)|| printf("MADEEASY"));`
`return 0;`
`}`
	Output :
	bool op will return 0 because of (a != b) is true.
	So printf statement is evaluated.

1.3 Flow Control in C

Looping is the process of repeating of same code until a specific condition doesn't satisfy. In C there are three types of loop: (a) for loop, (b) while loop and (c) do while.

For Loop

This loop is used when we have to execute a part of code in finite times. It is per tested loop.

Syntax of for loop:

```
for (Expression 1; Expression 2; Expression 3) {
    Loop body
}
```

Order of movement of control in for loop:

First time: Expression 1→ Expression 2 → Loop body → Expression 3.

Second time and onward: Expression 2 → Loop body → Expression 3.

Expression 1: Only executes in the first iteration. From second iteration and onward control doesn't go to the Expression 1.

Example - 1.15 What will be output of following program?

Code	Solution

`#include <stdio.h>`
`int main(){`
`int i;`
`for(i=0;i<=4;i++){`
`printf("%d ",i);`
`}`
`return 0;`
`}`
	Output: 0 1 2 3 4
	Expression 1 is called initialization expression. Task of this expression is to initialize the looping variables.

Properties of Expression 1:

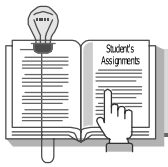
1. Expression 1 can initialize the more than one variable.
2. Expression 1 is optional.
3. Unlike Java, in C we cannot declare the variable at the expression 1.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

- **Polymorphism:** The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.
- **Overloading:** The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

Summary

- **'Auto' Storage Class:** The auto storage class is implicitly the default storage calls used simply specifies a normal local variable which is visible within its own code block only and which is created and destroyed automatically upon entry and exit respectively from the code block.
- **'Static' Storage Class:** The static storage class causes a local variable to become permanent within its own code block i.e. it retains its memory space and hence its value between function calls.
- **'Register' Storage Class:** The register storage class also specifies a normal local variable but it also requests that the compiler store a variable so that it may be accessed as quickly as possible, possible from a CPU register.
- **'Extern' Storage Class:** An extern class which informs the compiler for the existence of the global variable which enables it can be accessed in more that one source file.
- **Control Statements:** 1. **if** statement, 2. Nested **if** statement, 3. Conditional operator?, 4. **switch** statement, 5. **for** statement, 6. **while** statement, 7. **do-while** statement, 8. **break** statement and 9. **continue** statement.
- **Call by value:** "Actuals copied to formal", but not formals to actuals.
- **Call by reference:** "Actuals and formals uses same address space".
- **Call by value result (Restore):** "Actuals copied to formals and formals copied to actuals".
- **Call by Result:** "Actuals not copied to formals", but formals copied to actuals.
- **Call by Constant:** Formal values are never changed.
- **Call by Name:** The textual substitution of every occurrence of a formal parameter in the called routines body by the corresponding actual parameter.
- **Call by Text:** Same as call by name, but if variables are named same as local variables then it prefers local.
- **Call by Need:** Memory allocates only if formals are used in function.
- **Static Scoping:** The method of binding names to non local variables called static scoping. Scope of variable can be statically determined prior to execution.
- **Dynamic Scoping:** It is based on the calling sequence of subprograms. Scope can be determined at runtime.
- **Pointer:** A pointer is a variable that is used to store a memory address of another variable in memory. If one variable holds the address of another then it is said to the second variable.



Student's Assignments

Q.1 In 'C' language, the scope of the _____ variable extends from the point of definition throughout the remainder of the program.

(a) External (b) Static
(c) Automatic (d) Register

Q.2 What is the meaning of the following declaration?
int * p(char *a[]);

(a) P is a pointer to a function that accepts an argument which is a pointer to a character array and returns an integer quantity.
(b) P is a function that accepts an argument which is an array of pointers to characters and returns a pointer to an integer quantity.
(c) P is a function that accepts an argument which is an array of pointers to characters and returns an integer quantity.
(d) P is a pointer to a function that accepts an argument which is an array of pointers to characters and returns a pointer to an integer quantity.

Q.3 Consider the following program:

```
#define funct(x) x*x+x
int main()
{
  int x;
  x=36+funct(5)*funct(3);
  printf("%d",x);
  return 0;
}
```

What will be the output of the above program?

(a) 73 (b) 396
(c) 109 (d) 360

Q.4 Consider the following function given below:

```
int function(int n){
  if(n-1)
  return 2*function(n-1)+n;
  else
  return 0;
}
```

What is the value returned by function (5)?

(a) 33 (b) 41
(c) 57 (d) 65

Q.5 What is the output of the following program if dynamic scoping is used?

```
int a, b, c;
void func1(){
  int a,b;
  a=6;
  b=8;
  func2();
  a=a+b+c;
  print(a);
}
  void func2(){
  int b,c;
  b=4;
  c=a+b;
  a+= 11;
  print(c);
}
  void main(){
  a=3;
  b=5;
  c=7;
  func1();
}
```

Output of program:

(a) 7 19 (b) 10 1
(c) 10 23 (d) 10 32

Q.6 Output of the following program will be

```
int main
{
  int i,a[8]=000, 001, 010, 011, 100,
  101, 110, 111;
  for(i=0;i=8;i++)
  printf("%d",a,i);
  return 0;
}
```

(a) 0,1,2,3,4,5,6,7
(b) 0,1,10,11,100,101,110,110
(c) 0,1,8,9,100,101,110,111
(d) None of these

Q.7 Assuming only numbers and letters given in the input, what does the following program do?

```
#include<stdio.h>
int main()
{
  int i,j, ascii[128];
  char ip[30];
  printf("Enter Input string: ");
  scanf("%s",ip);
  for(i=0;i<128;i++)
{
  ascii[i]=0;
}
  i=0;
  while(ip[i]!='\0')
{
  j=(int)ip[i];
  ascii[j]++;
  if(ascii[j]>1)
{
  printf("%c",ip[i]);
  return 0;
}
   i++;
}
  return 0;
}
```

(a) Prints the position of first repeated character in the string
(b) Prints the first repeated character in the string
(c) Prints the position of last repeated character in the string
(d) Prints the last repeated character in the string

Q.8 Consider the following code segment.

```
void foo(int x, int y){
  X+=y;
  Y+=x;
}
  main(){
  int x=4.5;
  foo(x,x);
}
```

What is the final value of x in both call by value and call by reference respectively?

(a) 4 and 12 (b) 5 and 12
(c) 12 and 16 (d) 4 and 16

Q.9 What will be the output of following C-code?

```
int main(){
  int a[3]={67,43,23};
  int *p=a;
  printf("%d",++*p);
  printf("%d",*++p);
  printf("%d",*p++);
  return 0;
}
```

(a) 68 43 43 (b) 37 43 43
(c) 67 43 23 (d) 68 43 23

Q.10 Consider following C-code.
What is the value returned by fun (10)?

```
int fun (int n){
  Static int i=10;
  if (n>=200)
  return (n+i);
  else{
      n=n+i;
      i=n+i;
      return fun(n);
    }
}
```

(a) 890 (b) 210
(c) 340 (d) None of these

Q.11 What will be the output of following C-code?

```
void fun(void *p);
static int i;
int main()
{
  void *ptr;
  ptr &i;
  fun ptr ;
  return 0;
}
  void fun( void *p)
{
  int **q;
  q=(int**)&p;
  printf("%d",**q);
```

(a) Garbage value (b) 0
(c) 1 (d) Compile-time error

Q.12 Output of the following program will be:

```
int main()
{
  char *p1="Graduate";
  char *p2=p1;
  printf ("%c,%d",*++p2,sizeof(p2));
}
```

(a) G, 8 (b) r, 7
(c) r, 4 (d) r, 8

Q.13 What is the output of following program?

```
int f(int a)
{  printf("%d", a++);
   return(++a);
}
   main()
{
   int b=1;
   b=f(b);
   b=f(b);
   b=f(1+f(b));
}
```

(a) 1 3 3 5 (b) 1 3 5 7
(c) 2 3 3 5 (d) 2 4 6 8

Q.14 Consider the following C code:

```
int*P,A[3]={0,1,2};
P=A;
*(P+2)=5;
P=A++;
*P=7;
```

What are the values stored in the array A from index 0 to index 2 after execution of the above code?

(a) 7, 5, 2 (b) 7, 1, 5
(c) 0, 7, 5 (d) None of these

Q.15 Which of the following is not a valid decimal integer constant in 'C'?

(a) 10001 (b) 9999
(c) 2019 (d) 02019

Q.16 What is the associativity of the comma operator in 'C'?

(a) Left-to-right (b) Right-to-left
(c) Bottom-to-top (d) Top-to-bottom

Q.17 Comments can be placed anywhere in a 'C' program, as long as they are placed within the delimiters:

(a) */ and */ (b) /* and /*
(c) /* and // (d) /* and */

Q.18 In 'C' language, the gets( ) function reads string from keyboard and terminates reading after encountering a _____.

(a) Carriage return (b) Newline
(c) Space (d) Tab

Q.19 In 'C' language, an identifier name CANNOT start with

(a) Underscore (_) (b) Numeric digit
(c) Uppercase letter (d) Lowercase letter

Q.20 The 'C' language was originally developed at

(a) Bell labs (b) Intel
(c) Microsoft (d) IBM

Q.21 Consider the 'C' language variables x and y with initial values as 100 and 500, respectively. What will be the value of y after executing the following expression?

((x >= 100) || (y += 100));

(a) 100 (b) 600
(c) 200 (d) 500

Q.22 Consider the 'C' language variables x and y with initial values as 0 and 1, respectively. What will be the value of y after executing the following expression?

((x > 0) && (y = y + 1));

(a) 2 (b) 0
(c) 1 (d) 3

Q.23 Each 'C' program consists of one or more functions, one of which must be named as

(a) main (b) start
(c) primary (d) head

STUDENT'S ASSIGNMENTS ANSWER KEY

1. (a)	**2.** (b)	**3.** (c)	**4.** (b)	**5.** (a)
6. (c)	**7.** (b)	**8.** (d)	**9.** (a)	**10.** (a)
11. (c)	**12.** (c)	**13.** (b)	**14.** (d)	**15.** (d)
16. (a)	**17.** (d)	**18.** (b)	**19.** (b)	**20.** (a)
21. (d)	**22.** (c)	**23.** (a)		