**Operators**

**Writer:** Sayak Haldar

**Theory:**

**Operator Precedence Table:**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Associativity** |
| ( ) [ ] . -> ++ -- | Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2) | left-to-right |
| ++ -- + - ! ~ (*type*) \* & sizeof | Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation | right-to-left |
| \*  /  % | Multiplication/division/modulus | left-to-right |
| +  - | Addition/subtraction | left-to-right |
| <<  >> | Bitwise shift left, Bitwise shift right | left-to-right |
| <  <= >  >= | Relational less than/less than or equal to Relational greater than/greater than or equal to | left-to-right |
| ==  != | Relational is equal to/is not equal to | left-to-right |
| & | Bitwise AND | left-to-right |
| ^ | Bitwise exclusive OR | left-to-right |
| | | Bitwise inclusive OR | left-to-right |
| && | Logical AND | left-to-right |
| | | | Logical OR | left-to-right |
| ? : | Ternary conditional | right-to-left |
| = +=  -= \*=  /= %=  &= ^=  |= <<=  >>= | Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment | right-to-left |
| , | Comma (separate expressions) | left-to-right |
| **Note 1:**  Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.  **Note 2:**  Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x \* z++;**the current value of **z** is used to evaluate the expression (*i.e.,***z++**evaluates to **z**) and **z** only incremented after all else is done. See **[postinc.c](http://www.difranco.net/compsci/postinc.c)**for another example. | | |

**Sequence Point And Undefined Behaviour:**

**What are Sequence Points?**

At certain specified points in the execution sequence called sequence points , all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

**What are side effects?**

Evaluation of an expression produces something and if in addition there is a change in the state of the execution environment it is said that the expression (its evaluation) has some side effect(s).

**For example:**

int x = y++; //where y is also an int

In addition to the initialization operation the value of y gets changed due to the side effect of ++ operator. (since, it changes the value of y before assigning the value of y to x. So, there is a change in execution environment.

**What are the common sequence points listed in the C++ Standard?**

Those are:

•at the end of the evaluation of full expression (§1.9/16) (A full-expression is an expression that is not a sub expression of another expression.)1

**Example:**

int a = 5; // ; is a sequence point here

because, in the point of ';' all side effects of previous evaluation is complete (i.e. previous evaluation is a=5 and it is complete at this point) and no side effects of subsequent expression take place.

•in the evaluation of each of the following expressions after the evaluation of the first expression(§1.9/18)

**1.a && b**

**2.a || b**

1. **a,b**  (when comma acts as an operator)  
     
     
   (in func(a,a++) , is not a comma operator, it's merely a separator between the arguments a and a++. **The behaviour is undefined in that case if a is considered to be a primitive type)**

The comma separator’s behaviour is undefined when it is applied to primitive type

**4.a ? b : c**

•at a function call (whether or not the function is inline), after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body .

**(**

**Before a function is entered in a function call. The order in which the arguments are evaluated is not specified, but this sequence point means that all of their side effects are complete before the function is entered. In the expression f(i++) + g(j++) + h(k++), f is called with a parameter of the original value of i, but i is incremented before entering the body of f. Similarly, j and k are updated before entering g and h respectively. However, it is not specified in which order f(), g(), h() are executed, nor in which order i, j, k are incremented. If the body of f accesses the variables j and k, it might find both, neither, or just one of them to have been incremented. (The function call f(a,b,c) is not a use of the comma operator; the order of evaluation for a, b, and c is unspecified.**

**)**

**5.The end of a full expression. This category includes following expression statements**

a) Any full statement ended with semicolon like “a = b;”

b) return statements

c) The controlling expressions of if, switch, while, or do-while statements.

d) All three expressions in a for statement.

6. Between each declarator in each declarator sequence; for example, between the two evaluations of a++ in int x = a++, y = a++

1. At the end of an initializer; for example, after the evaluation of 5 in the declaration int a = 5;.
2. After each conversion associated with an input/output format specifier. For example, in the expression printf("foo %n %d", &a, 42), there is a sequence point after the %n is evaluated and before printing 42.

**What is undefined behaviour?**

behaviour, such as might arise upon use of an erroneous program construct or erroneous data, for which this International Standard imposes no requirements

Undefined behaviour may also be expected when this International Standard omits the description of any explicit definition of behaviour.

Relation between undefined behaviour and sequence points:

Before this , we must know the difference(s) between Undefined Behaviour, Unspecified Behaviour and Implementation defined Behaviour

We must also know the order of evaluation of operands of individual operators and sub expressions of individual expressions, and the order in which side effects take place, is unspecified.

**For example:**

int x = 5, y = 6;

int z = x++ + y++; //it is unspecified whether x++ or y++ will be evaluated first.

However, will it invoke undefined behaviour?

No, because, though we are not sure about the order, the value of z will be same.

**Now, it is said: (in the standard)**

**1.Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression.**

Informally it means that between two sequence points a variable must not be modified more than once. In an expression statement, the next sequence point is usually at the terminating semicolon, and the previous sequence point is at the end of the previous statement. An expression may also contain intermediate sequence points.

From the above sentence the following expressions invoke Undefined Behaviour.

i++ \* ++i; // i is modified more than once

++++++i; //parsed as (++(++(++i)))

**2. Furthermore, the prior value shall be accessed only to determine the value to be stored.**

What does it mean? It means if an object is written to within a full expression, any and all accesses to it within the same expression must be directly involved in the computation of the value to be written.

For example in i = i +1 all the access of i (in L.H.S and in R.H.S) are directly involved in computation of the value to be written. So it is fine.

This rule effectively constrains legal expressions to those in which the accesses demonstrably precede the modification.

i = (i,++i,++i); // undefined behaviour due to rule 2.

, operator will choose the rightmost expression. But, all accesses to i are not directly involved in computation of the value to be written.

**Example 1:**

a[i] = i++ // or a[++i] = i or a[i++] = ++i etc

It is disallowed because one of the accesses of i (the one in a[i]) has nothing to do with the value which ends up being stored in i (which happens over in i++), and so there's no good way to define--either for our understanding or the compiler's --whether the access should take place before or after the incremented value is stored. So the behaviour is undefined.

**Example 2:**

int x = i + i++ ;// Similar to above

**(again, all accesses to I are not involved in the computation of the value to be written in the main memory)**

i += ++i;

This invokes undefined behaviour if i is of predefined or built-in datatype.

However, in case i is of user defined datatype and we perform operator overloading to perform += (this is one type of assignment operator) as well as ++ (increment operator: one type of unary operator) then it will not invoke undefined behaviour.

class Index

{

int state;

public:

Index(int s) : state(s) {}

Index& operator++()

{

state++;

return \*this;

}

Index& operator+=(const Index & index)

{

state+= index.state;

return \*this;

}

operator int()

{

return state;

}

Index & add(const Index & index)

{

state += index.state;

return \*this;

}

Index & inc()

{

state++;

return \*this;

}

};

and then if we perform i.operator+=(i.operator ++());

then it will not invoke undefined behaviour. It will work perfectly fine with regards to sequence points. Section 1.9.17 of the C++ ISO standard says this about sequence points and function evaluation:

**When calling a function (whether or not the function is inline), there is a sequence point after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body. There is also a sequence point after the copying of a returned value and before the execution of any expressions outside the function.**

This would indicate, for example, that the i.operator ++() as the parameter to operator += has a sequence point after its evaluation. In short, because overloaded operators are functions, the normal sequencing rules apply.

expressions outside the function.

**Some examples Of Undefined Behaviour in Case Of Function Calling:**

**PROGRAM 1**

#include <cstdio>

int f1() { printf ("Geeks"); return 1;}

int f2() { printf ("forGeeks"); return 1;}

int main()

{

int p = f1() + f2();

return 0;

}

**PROGRAM 2**

#include <cstdio>

int x = 20;

int f1() { x = x+10; return x;}

int f2() { x = x-5; return x;}

int main()

{

int p = f1() + f2();

printf ("p = %d", p);

return 0;

}

**PROGRAM 3**

#include <cstdio>

int main()

{

int i = 8;

int p = i++\*i++;

printf("%d\n", p);

return 0;

}

The output of all of the above programs is undefined or unspecified. The output may be different with different compilers and different machines. It is like asking the value of undefined automatic variable.

The reason for undefined behaviour in PROGRAM 1 is, the operator ‘+’ doesn’t have standard defined order of evaluation for its operands. Either f1() or f2() may be executed first. So output may be either “GeeksforGeeks” or “forGeeksGeeks”.

Similar to operator ‘+’, most of the other similar operators like ‘-‘, ‘/’, ‘\*’, Bitwise AND &, Bitwise OR |, .. etc doesn’t have a standard defined order for evaluation for its operands.

Evaluation of an expression may also produce side effects. For example, in the above program 2, the final values of p are ambiguous. Depending on the order of expression evaluation, if f1() executes first, the value of p will be 55, otherwise 40.

The output of program 3 is also undefined. It may be 64, 72, or may be something else. The sub expression i++ causes a side effect, it modifies i’s value, which le

**But the output of the following programs are well defined**

/Following 3 lines are common in all of the below programs #include <cstdio>

int f1() { printf ("Geeks"); return 1;} int f2() { printf ("forGeeks"); return 1;}

**PROGRAM 4**

int main()

{

//Since && defines a sequence point after first operand, it is

//guaranteed that f1() is completed first.

int p = f1() && f2();

return 0;

}

**PROGRAM 5**

int main()

{

//Since comma operator defines a sequence point after first operand, it is

//guaranteed that f1() is completed first.

int p = (f1(), f2());

return 0;

}

**PROGRAM 6**

int main()

{

//Since ? operator defines a sequence point after first operand, it is

//guaranteed that f1() is completed first.

int p = f1()? f2(): 3;

return 0;

}

**Questions:**

**1. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 1, y = 0, z = 5;

int a = x && y || z++;

printf("%d", z);

}

a) 6

b) 5

c) 0

d) Varies

**Answer: a**

Now, two logical operators are used. Now, && has higher precedence than ||.

Now, && and || also acts as a sequence point.

So, the expression will be evaluated as:

(x&&y)||z++

((x)&&(y))||z++

Now,

|  |  |  |  |
| --- | --- | --- | --- |
| op-1 | op-2 | op-1 && op-2 | op-1 ||op-2 |
| Non-zero | Non-zero | 1 | 1 |
| 0 | Non-zero | 0 | 1 |
| Non-zero | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

Now, x && y will be evaluated as 0.

O||5

=1

a will be 1 and z will be 6.

(since, right side of || is evaluated since, left side of it evaluates to 0)

**2. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 1, y = 0, z = 5;

int a = x && y && z++;

printf("%d", z);

}

a) 6

b) 5

c) 0

d) Varies

**Answer: b**

Now, here, since both operators are &&, associativity will matter. Now, &&’s associativity is from left to right.

So, x&&y will be evaluated first. Now, since, it is evaluated to 0, the right side expression z++ will not be evaluated. Hence, b.

**3. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 1, y = 0, z = 3;

x > y ? printf("%d", z) : return z;

}

a) 3

b) 1

c) Compile time error

d) Run time error

**Answer: c**

Because of return z.

Compiler will say, expected expression before ‘return’ x > y ? printf("%d", z) : return z

**4. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 1, z = 3;

int y = x << 3;

printf(" %d\n", y);

}

a) -2147483648

b) -1

c) Run time error

d) 8

**Answer: d**

Now, << left bit shifting operator has higher precedence than assignment operator.

**5. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 0, y = 2, z = 3;

int a = x & y | z;

printf("%d", a);

}

a) 3

b) 0

c) 2

d) Run time error

**Answer: a**

Now, a=x & y|z has no sequence points. So, the total expression would be evaluated based on the precedence rules.

**a=x & y|z**

**a= (x&y)|z** //since bitwise and operation has higher precedence than bitwise or operation then (x & y)'s value would be bitwise or-ed to get a.

Now, even if x&y would be evaluated to 0 the right side of | will be evaluated.

**6. What is the final value of j in the below code?**

#include <stdio.h>

int main()

{

int i = 0, j = 0;

if (i && (j = i + 10))

//do something

;

}

a) 0

b) 10

c) Depends on the compiler

d) Depends on language standard

**Answer: a**Now, Since, && acts as a sequence point and here && is used and it’s left side value is 0 **j=i+10** wont execute.

Hence, j will remain as 0

**7. What is the final value of j in the below code?**

#include <stdio.h>

int main()

{

int i = 10, j = 0;

if (i || (j = i + 10))

//do something

;

}

a) 0

b) 20

c) Compile time error

d) Depends on language standard

**Answer: a**

Now, the left side expression of || will be evaluated first as || acts as a sequence point. Now, I is 10. Hence, result of the || operator is already determined. Hence, right side expression j=i+10 will not be evaluated. Hence, result is a.

**8. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 1;

if (i++ && (i == 1))

printf("Yes\n");

else

printf("No\n");

}

a) Yes

b) No

c) Depends on the compiler

d) Depends on the standard

**Answer: b**

Now, i++ will be evaluated first. as && acts as sequence point. Now, Since, post increment operator is applied, I’s incremented value will be available in the memory or cache after the execution of nearest sequence point, which is && here.

So, when (i==1) is executed, i’s value is 2. Hence, (i==1) will return 0. Hence, the total expression i++ && (i == 1) will return 0. Hence, if block will not be executed and it’s corresponding else block will be executed. Hence, **No will be printed.**

**9. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 0;

int x = i++, y = ++i;

printf("%d % d\n", x, y);

return 0;

}

a) 0, 2

b) 0, 1

c) 1, 2

d) Undefined

**Answer: a**

Now, if you think this will will invoke undefined behaviour, read the following rule about sequence point:

In C and C++, sequence points occur in the following places:

Between each declarator in each declarator sequence; for example, between the two evaluations of a++ in int x = a++, y = a++

It is one of the rule. Hence, it is safe.

**10. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 10;

int \*p = &i;

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

}

a) 10

b) 11

c) Garbage value

d) Address of i

**Answer) a) 10**

Now, ++ post increment operator has higher precedence has \* (dereference) operator. Hence, the memory location pointed by p will be incremented first. But, since, it is post incrementation, the incremented value will be applicable after the next sequence point. Which is ‘;’ line 6. Hence, the result will be 10.

**11. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 97;

int y = sizeof(x++);

printf("X is %d", x);

}

a) X is 97

b) X is 98

c) X is 99

d) Run time error

**Answer) a) X is 97.**

Now, x++ will be calculated in compile time. Since, sizeof() is a compile time operator. Now, in runtime, x’s value will remain same.

**12. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 4, y, z;

y = --x;

z = x--;

printf("%d%d%d", x, y, z);

}

a) 3 2 3

b) 2 3 3

c) 3 2 2

d) 2 3 4

**Answer) is b) 2 3 3**

Now, y=--x here, -- pre increment operator has higher precedence than assignment operator. Now, after line 3, y has value 3 x has value 3.

Now, consider line 4. Again, -- (post increment operator) has higher precedence over = operator. So, x is post incremented. But, x’s updated value will be available after the execution of next sequence point. Hence, 3 is assigned to z.

**13. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 4;

int \*p = &x;

int \*k = p++;

int r = p - k;

printf("%d", r);

}

a) 4

b) 8

c) 1

d) Run time error

Now, p and k contains the same memory address. Since, in line 6, p (the pointer variable to int) is post incremented. Now, r will be 1.  
Since, in line 7, p has the incremented value

**Hence, the answer is c) 1**

**14. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = 5, b = -7, c = 0, d;

d = ++a && ++b || ++c;

printf("\n%d%d%d%d", a, b, c, d);

}

a) 6 -6 0 0

b) 6 -5 0 1

c) -6 -6 0 1

d) 6 -6 0 1

Answer) 6 -6 0 1

**(c will not be incremented since ||’s left side is evaluated to 1)**

**15. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = -5;

int k = (a++, ++a);

printf("%d\n", k);

}

a) -4

b) -5

c) 4

d) -3

**Answer) d) -3**

Now, here, comma acts as an operator. It will choose the rightmost expression’s value and assign k with it. Now, since, ‘,’ acts as a sequence point, it’s left side expression will be evaluated and after it it’s right side expression will be evaluated(even though, the rightmost expression’s value will be assigned to k).

Now, the left side expression is a++. Now, a’s incremented value will be available when right side expression ++a is evaluated (here, it is pre-increment operator). Hence, k is -3

1. **What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = -3;

int k = i % 2;

printf("%d\n", k);

}

a) Compile time error

b) -1

c) 1

d) Implementation defined

**Answer: b**

Because, i/2 is -1.

Now, remainder=-3-(-2)=-1

**17. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 3;

int l = i / -2;

int k = i % -2;

printf("%d %d\n", l, k);

return 0;

}

a) Compile time error

b) -1 1

c) 1 -1

d) Implementation defined

**Answer: b**

**18. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 5;

i = i / 3;

printf("%d\n", i);

return 0;

}

a) Implementation defined

b) 1

c) 3

d) Compile time error

**Answer: b**

**19. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = -5;

i = i / 3;

printf("%d\n", i);

return 0;

}

a) Implementation defined

b) -1

c) -3

d) Compile time error

**Answer: b**

**20. What is the value of x in this C code?**

#include <stdio.h>

void main()

{

int x = 5 \* 9 / 3 + 9;

}

a) 3.75

b) Depends on compiler

c) 24

d) 3

Now, \* and / has same precedence. And, associativity as left to right. Hence, \* will operate first, then /. Now, precedence of \* and / is higher than +.

Hence, **Answer is c)24.**

**21. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 5.3 % 2;

printf("Value of x is %d", x);

}

a) Value of x is 2.3

b) Value of x is 1

c) Value of x is 0.3

d) Compile time error

**Answer: d**

Modulus operator wont work on floating point constants.

**22. What is the output of this C code?**

#include <stdio.h>

void main()

{

int y = 3;

int x = 5 % 2 \* 3 / 2;

printf("Value of x is %d", x);

}

a) Value of x is 1

b) Value of x is 2

c) Value of x is 3

d) Compile time error

**Answer: a**

Because, %, \*, / has same precedence. And, they have associativity from left to right.

**23. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = 3;

int b = ++a + a++ + --a;

printf("Value of b is %d", b);

}

a) Value of x is 12

b) Value of x is 13

c) Value of x is 10

d) Undefined behaviour

**Answer: d**

Now, a’s value is modified more than once between two consecutive sequence point.

**24. The precedence of arithmetic operators is (from highest to lowest)**

a) %, \*, /, +, –

b) %, +, /, \*, –

c) +, -, %, \*, /

d) %, +, -, \*, /

**Answer: a**

**25. Which of the following is not an arithmetic operation?**

a) a \*= 10;

b) a /= 10;

c) a != 10;

d) a %= 10;

**Answer: c**

It is a relational operator.

**26. Which of the following data type will throw an error on modulus operation(%)?**

a) char

b) short

c) int

d) float

**Answer: d**

**27. Which among the following are the fundamental arithmetic operators, i.e., performing the desired operation can be done using that operator only?**

a) +, –

b) +, -, %

c) +, -, \*, /

d) +, -, \*, /, %

**Answer: a**

**28. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 10;

double b = 5.6;

int c;

c = a + b;

printf("%d", c);

}

a) 15

b) 16

c) 15.6

d) 10

Answer: a

lot of implicit typecasting is involved

For instance, a+b ‘s value will be temporarily stored as double.

It is implicitly typecasted to double when a+b’s value is assigned to c.

**29. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 10, b = 5, c = 5;

int d;

d = a == (b + c);

printf("%d", d);

}

a) Syntax error

b) 1

c) 10

d) 5

**Answer: b**

Now, == relational operator has higher precedence than assignment operator.   
  
a==b+c will return 1.

1 is assigned to d.

**30. function tolower(c) defined in library works for**

a) ASCII character set

b) Unicode character set

c) ASCII and utf-8 but not EBSIDIC character set

d) Any character set

**Answer: d**

**31. What is the output of the below code considering size of short int is 2, char is 1 and int is 4 bytes?**

#include <stdio.h>

int main()

{

short int i = 20;

char c = 97;

printf("%d, %d, %d\n", sizeof(i), sizeof(c), sizeof(c + i));

return 0;

}

a) 2, 1, 2

b) 2, 1, 1

c) 2, 1, 4

d) 2, 2, 8

**Answer: c**

Now, c+i is an expression. Which will be evaluated to a value. Now, that value will be integral. Hence, sizeof(c+i) will be 4.

**32. Which type conversion is NOT accepted?**

a) From char to int

b) From float to char pointer

c) From negative int to char

d) From double to char

Answer: b

Explanation: Conversion of a float to pointer type is not allowed.

**33. What will be the data type of the result of the following operation?**

**(float)a \* (int)b / (long)c \* (double)d**

a) int

b) long

c) float

d) double

Answer) d) double.

Read the rules of the implicit type casting:

Implicit Type Conversion Also known as ‘automatic type conversion’.

Done by the compiler on its own, without any external trigger from the user.

Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.

All the data types of the variables are upgraded to the data type of the variable with largest data type.

**bool -> char -> short int -> int -> unsigned int -> long -> unsigned ->**

**long long -> float -> double -> long double**

**34. Which of the following type-casting have chances for wrap around?**

a) From int to float

b) From int to char

c) From char to short

d) From char to int

Answer: b

**bool -> char -> short int -> int -> unsigned int -> long -> unsigned ->**

**long long -> float -> double -> long double**

Typecasting in opposite direction will always have the chance of wrapping around.

**35. Which of the following typecasting is accepted by C?**

a) Widening conversions

b) Narrowing conversions

c) Widening & Narrowing conversions

d) None of the mentioned

**Answer: c**

**36. When do you need to use type-conversions?**

a) The value to be stored is beyond the max limit

b) The value to be stored is in a form not supported by that data type

c) To reduce the memory in use, relevant to the value

d) All of the mentioned

**Answer: d**

**37. What is the output of this C code?**

#include <stdio.h>

void main()

{

float x = 0.1;

if (x == 0.1)

printf("Sanfoundry");

else

printf("Advanced C Classes");

}

a) Advanced C Classes

b) Sanfoundry

c) Run time error

d) Compile time error

**Answer: a**

Now, 0.1 is double constant. Not a float constant.

To get the result as Sanfoundry, you have to change the program to the following:

#include <stdio.h>

void main()

{

float x = 0.1;

if (x == 0.1f)

printf("Sanfoundry");

else

printf("Advanced C Classes");

}

**38. Comment on the output of this C code?**

#include <stdio.h>

void main()

{

float x = 0.1;

printf("%d, ", x);

printf("%f", x);

}

a) 0.100000, junk value

b) Junk value, 0.100000

c) 0, 0.100000

d) 0, 0.999999

**Answer: b**

Because, of float’s internal storage technique.

A typical 32-bit layout looks something like the following:

3 32222222 22211111111110000000000

1 09876543 21098765432109876543210

+-+-----------+---------------------------------------+

| | | |

+-+-----------+---------------------------------------+

^ ^ ^

| | |

| | +-- significand

| |

| +------------------- exponent

|

+------------------------ sign bit

Like signed integer types, the high-order bit indicates sign; 0 indicates a positive value, 1 indicates negative.

The next 8 bits are used for the exponent. Exponents can be positive or negative, but instead of reserving another sign bit, they're encoded such that 10000000 represents 0, so 00000000 represents -128 and 11111111 represents 127.

The remaining bits are used for the significand. Each bit represents a negative power of 2 counting from the left, so:

01101 = 0 \* 2-1 + 1 \* 2-2 + 1 \* 2-3 + 0 \* 2-4 + 1 \* 2-5

= 0.25 + 0.125 + 0.03125

= 0.40625

Some platforms assume a "hidden" leading bit in the significand that's always set to 1, so values in the significand are always between [0.5, 1). This allows these platforms to store values with a slightly greater precision (more on that below). My example doesn't do this.

**39. What is the output of this C code?**

(7 and 8 are entered)

#include <stdio.h>

void main()

{

float x;

int y;

printf("enter two numbers \n", x);

scanf("%f %f", &x, &y);

printf("%f, %d", x, y);

}

a) 7.000000, 7

b) Run time error

c) 7.000000, junk

d) Varies

**Answer: c**

**40. What is the output of this C code?**

#include <stdio.h>

void main()

{

double x = 123828749.66;

int y = x;

printf("%d\n", y);

printf("%lf\n", y);

}

a) 0, 0.0

b) 123828749, 123828749.66

c) 12382874, 12382874.0

d) 123828749, 0.000000

Answer: d

Because, y is an integer variable and it is being read as double variable.

Now, double has 1 signed bit, 11 exponent bit and 52 mantissa or significand bit.

Now, again 0, because of internal storage technique of double variable

**41. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 97;

char y = x;

printf("%c\n", y);

}

a) a

b) b

c) 97

d) Run time error

Answer) a)a

**42. When double is converted to float, the value is?**

a) Truncated

b) Rounded

c) Depends on the compiler

d) Depends on the standard

**Answer: c**

**43. What is the output of this C code?**

#include <stdio.h>

int main()

{

unsigned int i = 23;

signed char c = -23;

if (i > c)

printf("Yes\n");

else if (i < c)

printf("No\n");

}

a) Yes

b) No

c) Depends on the compiler

d) Depends on the operating system

**Answer: b**

(signed and unsigned bit concept, Implicit typecasting concept)

Now, signed char: char has 8 bits

Now, ~a=-(a+1)  
  
Hence, ~a+1=-a

Now, -23’s bitwise representation is

~(00010111)+00000001

=11101000+00000001  
  
=11101001  
  
(=-128+64+32+8+1=-23)

Now, if it is implicitly typecasted to unsigned int

00000000 00000000 00000000 11101001

Which has value 128+64+32+8+1=128+105=233

Hence, **No**

**44. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 23;

char c = -23;

if (i < c)

printf("Yes\n");

else

printf("No\n");

}

a) Yes

b) No

c) Depends on the compiler

d) Depends on the standard

**Answer: b**

Again, char will be implicitly typecasted to int**. Try this on your own.**

**45. What is the difference between the following 2 codes?**

#include <stdio.h> //Program 1

int main()

{

int d, a = 1, b = 2;

d = a++ + ++b;

printf("%d %d %d", d, a, b);

}

#include <stdio.h> //Program 2

int main()

{

int d, a = 1, b = 2;

d = a++ +++b;

printf("%d %d %d", d, a, b);

}

a) No difference as space doesn’t make any difference, values of a, b, d are same in both the case

b) Space does make a difference, values of a, b, d are different

c) Program 1 has syntax error, program 2 is not

d) Program 2 has syntax error, program 1 is not

**Answer) d) Program 2 has syntax error, program 1 is not.**

**46. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 1, b = 1, c;

c = a++ + b;

printf("%d, %d", a, b);

}

a) a = 1, b = 1

b) a = 2, b = 1

c) a = 1, b = 2

d) a = 2, b = 2

**Answer: b**

**47. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 1, b = 1, d = 1;

printf("%d, %d, %d", ++a + ++a+a++, a++ + ++b, ++d + d++ + a++);

}

a) 15, 4, 5

b) 9, 6, 9

c) 9, 3, 5

d) Undefined (Compiler Dependent)

**Answer: d**

In line 5, commas act as separators. Now, a’s value is incremented more than once b/w two consecutive points. Hence, Undefined.

**48. For which of the following, “PI++;” code will fail?**

a) #define PI 3.14

b) char \*PI = “A”;

c) float PI = 3.14;

d) none of the Mentioned

**Answer: a**

Because, ++ operator wont find any lvalue. Since, PI is substituted to 3.14 before compilation. (done in preprocessing step)

**49. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 10, b = 10;

if (a = 5)

b--;

printf("%d, %d", a, b--);

}

a) a = 10, b = 9

b) a = 10, b = 8

c) a = 5, b = 9

d) a = 5, b = 8

**Answer: c**

**50. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 0;

int j = i++ + i;

printf("%d\n", j);

}

a) 0

b) 1

c) 2

d) Compile time error

**Answer: a**

Now, initially it will look like undefined. But, i++ +i (here, I’s value is post incremented) and i’s value is modified once and post increment operator has higher precedence than binary plus.

**51. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 2;

int j = ++i + i;

printf("%d\n", j);

}

a) 6

b) 5

c) 4

d) Compile time error

**Answer: a**

Now, associativity wont play any rule. But, precedence will.

Now, between 2 sequence points i’s value is not modified more than once. And pre increment operator has higher precedence over binary plus.

**52. Comment on the output of this C code?**

#include <stdio.h>

int main()

{

int i = 2;

int i = i++ + i;

printf("%d\n", i);

}

a) = operator is not a sequence point

b) ++ operator may return value with or without side effects

c) it can be evaluated as (i++)+i or i+(++i)

d) = operator is a sequence point

**Answer: a**

1. **What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 0;

int x = i++, y = ++i;

printf("%d % d\n", x, y);

return 0;

}

a) 0, 2

b) 0, 1

c) 1, 2

d) Undefined

**Answer: a**

It will not generate undefined behaviour. Consider the following sequence point rule:

In C and C++, sequence points occur in the following places. (In C++, overloaded operators act like functions, and thus operators that have been overloaded introduce sequence points in the same way as function calls.)

Between each declarator in each declarator sequence; for example, between the two evaluations of a++ in int x = a++, y = a++

**53. What is the output of this C code?**

#include <stdio.h>

int main()

{

int i = 10;

int \*p = &i;

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

}

a) 10

b) 11

c) Garbage value

d) Address of i

**Answer: a**

**54. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 97;

int y = sizeof(x++);

printf("X is %d", x);

}

a) X is 97

b) X is 98

c) X is 99

d) Run time error

**Answer) a**

Because, x’s value will be incremented in compile time and y=sizeof(98) will happen. Now, Since, 98 is an integral constant, y will be 4. However, x’s value will not be incremented in runtime.

**55. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 4, y, z;

y = --x;

z = x--;

printf("%d%d%d", x, y, z);

}

a) 3 2 3

b) 2 3 3

c) 3 2 2

d) 2 3 4

Answer: b

**55. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 4;

int \*p = &x;

int \*k = p++;

int r = p - k;

printf("%d", r);

}

a) 4

b) 8

c) 1

d) Run time error

**Answer: c**

**56. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = 5, b = -7, c = 0, d;

d = ++a && ++b || ++c;

printf("\n%d%d%d%d", a, b, c, d);

}

a) 6 -6 0 0

b) 6 -5 0 1

c) -6 -6 0 1

d) 6 -6 0 1

Answer: d

**57. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = -5;

int k = (a++, ++a);

printf("%d\n", k);

}

a) -4

b) -5

c) 4

d) -3

**Answer: d**

**58. What is the output of this C code?**

#include <stdio.h>

int main()

{

int c = 2 ^ 3;

printf("%d\n", c);

}

a) 1

b) 8

c) 9

d) 0

**Answer: a**

Now, ^ has higher precedence than assignment operator. Som, 2^3 will be evaluated. Now, ^ is bitwise xor. 10^11=01 =1

**59. What is the output of this C code?**

#include <stdio.h>

int main()

{

unsigned int a = 10;

a = ~a;

printf("%d\n", a);

}

a) -9

b) -10

c) -11

d) 10

**Answer: c**

Now, a=~a. Now, ~a=-(a+1)=hence, -11

**60. What is the output of this C code?**

#include <stdio.h>

int main()

{

if (7 & 8)

printf("Honesty");

if ((~7 & 0x000f) == 8)

printf("is the best policy\n");

}

a) Honesty is the best policy

b) Honesty

c) is the best policy

d) No output

View Answer

**Answer: c**

Now, according to the scope, the code is:

#include <stdio.h>

int main()

{

if (7 & 8)

{

printf("Honesty");

}

if ((~7 & 0x000f) == 8)

{

printf("is the best policy\n");

}

}

Now, obviously, honestly will not be printed. Since, 7&8 will be 0.

Now, ~7   
  
Let’s consider 7’s bitwise representation:

00000000 00000000 00000000 00000111

Now, ~7’s bitwise representation will be:

11111111 11111111 11111111 11111000  
  
Now, 0x000f is 15   
  
00000000 00000000 00000000 00001111  
  
Now, bitwise and of them will generate 8.

**61. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 2;

if (a >> 1)

printf("%d\n", a);

}

a) 0

b) 1

c) 2

d) No Output.

Now, a>>1 will be 1. But, this value 1 will not be assigned to a. Hence, a will be printed as 2.

**Answer: c**

**62. Comment on the output of this C code?**

#include <stdio.h>

int main()

{

int i, n, a = 4;

scanf("%d", &n);

for (i = 0; i < n; i++)

a = a \* 2;

}

a) Logical Shift left

b) No output

c) Arithmetic Shift right

d) bitwise exclusive OR

Answer: b

**Because, there is no printf statement.**

**63. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 97;

int y = sizeof(x++);

printf("x is %d", x);

}

a) x is 97

b) x is 98

c) x is 99

d) Run time error

**Answer: a**

**64. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 4, y, z;

y = --x;

z = x--;

printf("%d%d%d", x, y, z);

}

a) 3 2 3

b) 2 2 3

c) 3 2 2

d) 2 3 3

**Answer: d**

**65. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2;

x = x << 1;

printf("%d\n", x);

}

a) 4

b) 1

c) Depends on the compiler

d) Depends on the endianness of the machine

**Answer: a**

**Important Concept:**

**66. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = -2;

x = x >> 1;

printf("%d\n", x);

}

a) 1

b) -1

c) 2 31 – 1 considering int to be 4 bytes

d) Either -1 or 1

**Answer: b**

Now,

11111111 11111111 11111111 11111110 This is bitwise representation of -2.

Now, we are right shifting by 1 bit

01111111 11111111 11111111 11111111

20+21+……+230 =231-1

According to calculation.

But, I run this program. This is really generating -1

Bingo, you will learn a new concept.

The "Right Shift (Arithmetic)" is useful when used on twos-complement numbers. The sign bit (leftmost bit) is replicated at the high end of the number, instead of bringing in zeroes as with "Right Shift (Logical)". If the number is negative, replicating the sign bit with "Right Shift (Arithmetic)" keeps it negative.

Hence, right shifting

11111111 11111111 11111111 11111110

Will result

11111111 11111111 11111111 11111111

**(now, right shift arithmetic may or may not be used in c)**

**67. What is the output of this C code?**

#include <stdio.h>

int main()

{

if (~0 == 1)

printf("yes\n");

else

printf("no\n");

}

a) yes

b) no

c) compile time error

d) undefined

**Answer: b**

**68. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = -2;

if (!0 == 1)

printf("yes\n");

else

printf("no\n");

}

a) yes

b) no

c) run time error

d) undefined

View Answer

**Answer: a**Now, **!** this bitwise not has higher precedence than == relational operator.

**69. What is the output of this C code?**

#include <stdio.h>

int main()

{

int y = 0;

if (1 |(y = 1))

printf("y is %d\n", y);

else

printf("%d\n", y);

}

a) y is 1

b) 1

c) run time error

d) undefined

**Answer: a**

(now, parentheses around (y=1) makes the right side expression of | evaluated first. Otherwise, there is not order. Now, this assigns 1 to y. 1|y=1|1=1  
Hence, **y is 1 will be printed**

**70. What is the output of this C code?**

#include <stdio.h>

int main()

{

int y = 1;

if (y & (y = 2))

printf("true %d\n", y);

else

printf("false %d\n", y);

}

a) true 2

b) false 2

c) either option a or option b

d) true 1

**Answer: a)**

**In case of option c**

That means the output is compiler dependent. Now, (y=2) will be evaluated first, (right?) due to the presence of parentheses. If parentheses was not present, it would be compiler dependent. As, there is no particular order which tells which side of the & will be evaluated first.

1. **What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 0;

if (x = 0)

printf("Its zero\n");

else

printf("Its not zero\n");

}

a) Its not zero

b) Its zero

c) Run time error

d) None

**Answer: a**

**72. What is the output of this C code?**

#include <stdio.h>

void main()

{

int k = 8;

int x = 0 == 1 && k++;

printf("%d%d\n", x, k);

}

a) 0 9

b) 0 8

c) 1 8

d) 1 9

**Answer: b**

Now, relational operator == has the highest precedence. 0==1 will be evaluated first. Which will be evaluated to 0. Now, 0 && k++ will be evaluated. Since, left side of the && is 0, **k++**, this expression wont be evaluated. And x will have the value 0. Now, k will remain as 8.

**73. What is the output of this C code?**

#include <stdio.h>

void main()

{

char a = 'a';

int x = (a % 10)++;

printf("%d\n", x);

}

a) 6

b) Junk value

c) Compile time error

d) 7

**Answer: c**

As no lvalue for post increment operator.

**74. What is the output of this C code?**

#include <stdio.h>

void main()

{

1 < 2 ? return 1: return 2;

}

a) returns 1

b) returns 2

c) Varies

d) Compile time error

**Answer: d.  
  
Return statement cannot be a part of conditional operator. (ternary operator)**

**75. What is the output of this C code?**

#include <stdio.h>

void main()

{

unsigned int x = -5;

printf("%d", x);

}

a) Run time error

b) Aries

c) -5

d) 5

Answer: c

Because, the unsigned int memory location is read as int.

Now, what will it do? The same 4 byte memory:

11111111 11111111 11111111 11111011  
  
(since, -5 is ~4)

Now, the the memory is read as unsigned int it will print:

232-1-4=232-5

**76. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 1;

x \*= x + y;

printf("%d\n", x);

return 0;

}

a) 5

b) 6

c) Undefined behaviour

d) Compile time error

**Answer: b**

Now, + has higher precedence than \*=  
  
Hence,

x \*= x + y

Will act like: x\*=(x+y)

**77. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 2;

x /= x / y;

printf("%d\n", x);

return 0;

}

a) 2

b) 1

c) 0.5

d) Undefined behaviour

**Answer: a**

/= has lower precedence than /

Hence, it will act as:

x/=(x/y)  
  
Hence, Now, x/=1 it will result as 2.

**78. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 1, y = 0;

x &&= y;

printf("%d\n", x);

}

a) Compile time error

b) 1

c) 0

d) Undefined behaviour

**You must be thinking:**

Answer. x=x&&y  
x will be 0.

But, now notice, &&= is an invalid operator.

**Hence. A) Compile time error.**

**79. What is the type of the below assignment expression if x is of type float, y is of type int?**

y = x + y;

a) int

b) float

c) there is no type for an assignment expression

d) double

**Answer: a**

**(implicit typecasting chart:**

**bool -> char -> short int -> int ->unsigned int -> long -> unsigned -> long long -> float -> double -> long double**

**)**

**80. What is the value of the below assignment expression (x’s value)**

(x = foo())!= 1 considering foo() returns 2

a) 2

b) True

c) 1

d) 0

Now, != is relational operator. 2!=1 will return 1. However, x will remain 2.

**Answer: a**

**81. Operation “a = a \* b + a” can also be written as:**

a) a \*= b + 1;

b) (c = a \* b)!=(a = c + a);

c) a = (b + 1)\* a;

d) All of the mentioned

**Answer: d**

**82. for c = 2, value of c after c <<= 1;**

a) c = 1;

b) c = 2;

c) c = 3;

d) c = 4;

**Answer: d**

**83. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 1, b = 2;

a += b -= a;

printf("%d %d", a, b);

}

a) 1 1

b) 1 2

c) 2 1

d) 2 2

**Answer: c**

Now, += and -= has same precedence. Now, associativity is right to left.

Hence b-=a will be evaluated.  
  
**b will be 1.**

a+=1

**a will be 2.**

Now, it wont invoke any undefined behaviour. Because, a and b are modified once.  
  
**84. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = 4, n, i, result = 0;

scanf("%d", n);

for (i = 0;i < n; i++)

result += a;

}

a) Addition of a and n

b) Subtraction of a and n

c) Multiplication of a and n

d) Division of a and n

**Answer: c**

**85. Which of the following is an invalid assignment operator?**

a) a %= 10;

b) a /= 10;

c) a |= 10;

d) None of the mentioned.

**Answer: d**

**86. What is the output of this C code?**

**#include <stdio.h>**

**int main()**

**{**

**reverse(1);**

**}**

**void reverse(int i)**

**{**

**if (i > 5)**

**exit(0);**

**printf("%d\n", i);**

**return reverse(i++);**

**}**

a) 1 2 3 4 5

b) 1 2 3 4

c) Compile time error

d) Stack overflow

**Answer: d**

Because, i is post incremented. Now, function call takes place with the same value of i which is 1.

Now, if you have doubt regarding the sequence point rule. Read the rule again:

**Before a function is entered in a function call. The order in which the arguments are evaluated is not specified, but this sequence point means that all of their side effects are complete before the function is entered. In the expression f(i++) + g(j++) + h(k++), f is called with a parameter of the original value of i, but i is incremented before entering the body of f. Similarly, j and k are updated before entering g and h respectively. However, it is not specified in which order f(), g(), h() are executed, nor in which order i, j, k are incremented. If the body of f accesses the variables j and k, it might find both, neither, or just one of them to have been incremented. (The function call f(a,b,c) is not a use of the comma operator; the order of evaluation for a, b, and c is unspecified.)**

**87. What is the output of this C code?**

#include <stdio.h>

void reverse(int i);

int main()

{

reverse(1);

}

void reverse(int i)

{

if (i > 5)

return ;

printf("%d ", i);

return reverse((i++, i));

}

a) 1 2 3 4 5

b) Segmentation fault

c) Compilation error

d) Undefined behaviour

**Answer: a**

Comma acts as an operator here. So, comma is also a sequence point. Though, comma operator will return the rightmost expression’s value, it will make all expression evaluated. Hence, I++ will be evaluated. Now, since, ‘,’ acts as a sequence point, before the right part of the comma operator is the executed, I’s value is change to 2 during the first recursive call.

**88. In expression i = g() + f(), first function called depends on**

a) Compiler

b) Associativity of () operator

c) Precedence of () and + operator

d) Left to write of the expression

**Answer: a**

**89. What is the value of i and j in the below code?**

#include <stdio.h>

int x = 0;

int main()

{

int i = (f() + g()) || g();

int j = g() || (f() + g());

}

int f()

{

if (x == 0)

return x + 1;

else

return x - 1;

}

int g()

{

return x++;

}

a) i value is 1 and j value is 1

b) i value is 0 and j value is 0

c) i value is 1 and j value is undefined

d) i and j value are undefined

**Answer: d**

**90. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 0;

int z = y && (y |= 10);

printf("%d\n", z);

return 0;

}

a) 1

b) 0

c) Undefined behaviour due to order of evaluation

d) 2

**Answer: b**

Now, && operator has higher precedence than =.

Because, && acts as a sequence point. Hence, left side expression will be evaluated first.

**91. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 0;

int z = (y++) ? 2 : y == 1 && x;

printf("%d\n", z);

return 0;

}

a) 0

b) 1

c) 2

d) Undefined behaviour

**Answer: b**

Now, ? mark in ternary operator acts as a sequence point. Hence, when y==1 is being evaluated, y’s value is 1.

**92. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 0;

int z;

z = (y++, y);

printf("%d\n", z);

return 0;

}

a) 0

b) 1

c) Undefined behaviour

d) Compilation error

**Answer: b**

Now, comma will act as an operator. Now, it will also act as sequence point. Though, comma operator chooses the rightmost expression’s value as return value, all the expressions’ value will be evaluated.  
  
**93. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 0, l;

int z;

z = y = 1, l = x && y;

printf("%d\n", l);

return 0;

}

a) 0

b) 1

c) Undefined behaviour due to order of evaluation can be different

d) Compilation error

**Answer: b**

**94. What is the output of this C code?**

#include <stdio.h>

int main()

{

int y = 2;

int z = y +(y = 10);

printf("%d\n", z);

}

a) 12

b) 20

c) 4

d) Either 12 or 20

**Answer: b**

Because, parentheses around y=10 makes it executed first. Now, this assignment makes y as 10 and y’s value is reflected in memory. Hence, 20

**95.What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 2;

float f = y + x /= x / y;

printf("%d %f\n", x, f);

return 0;

}

a) 2 4.000000

b) Compile time error

c) 2 3.500000

d) Undefined behaviour

**Answer: b**

Now, the operator with highest precedence is /. Then +. Then =. Then /=.

f=y+x/=x/y;

f=y+x/=(x/y);  
f=y+x/=1;

f=4/=1;  
(f=4)/=1;

Now, f becomes 4 due to assignment. Hence, /= operator wont find any lvalue.

**Now, this will also cause the compilation error. But, this is wrong way of interpretation.**

**The right way of interpretation is the following:**

f = y + x /= x / y;

f=y + x /= 1;

f=4/=1;

Now, /= operator and = has same precedence. Hence, associativity will be applied. Now, /= will operate first. But, both side of it becomes integral constant. Hence, it cannot operate. Hence, compilation error.

So, indeed = and /= and %= has same operator precedence. And, I was interpreting the table wrongly.

**96. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 1, y = 2;

if (x && y == 1)

printf("true\n");

else

printf("false\n");

}

a) true

b) false

c) compile time error

d) undefined behaviour

**Answer: b**

Now, relation operator has higher precedence. y==1 will return 0. Now, x && 0 =0. (Hence, the fact that && acts as sequence point will not be used. Because, &&’s right side expression will be evaluated in both cases since,x=1 and ==’s higher precedence.

**98. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 1, y = 2;

int z = x & y == 2;

printf("%d\n", z);

}

a) 0

b) 1

c) Compile time error

d) Undefined behaviour

**Answer: b**

(relational == has higher precedence than &)

z=x&(y==2)  
z=x&1  
z=1&1  
z=1

**99. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 3, y = 2;

int z = x /= y %= 2;

printf("%d\n", z);

}

a) 1

b) Compile time error

c) Floating point exception

d) Segmentation fault

**Answer: c**

**Wrong way of interpretation**

Now, /= %= has same precedence.  
  
Direct assignment has greater precedence than them.  
  
Now, Let’s try to find.

z=x/=y%=2;

(z=x)/=y%=2; due to higher precedence.  
  
3/=y%=2; Now, /= and %= has same precedence. But, the associativity is right to left. So,

3/=(y%=2)  
  
3/=0  
  
Now, that means division by 0.  
  
Now, since, /=’s right hand operand is 0, it will give floating point exception.

But, that is giving me some conceptual problem. That means, if I change y to 3, the output which should come is: compilation error. Compiler should say, lvalue required as left operand of assignment.

But, it is not saying so.

**#include <stdio.h>**

**int main()**

**{**

**int x = 3, y = 3;**

**int z = x /= y %= 2;**

**printf("%d\n", z);**

**}**

Now, that will generate some result. It will generate 3.

That is only possible when =,/=,%= all have same precedence and only associativity concept plays the role.

Now, applying the same concept here:  
  
 #include <stdio.h>

int main()

{

int x = 2, y = 2;

float f = y + x /= x / y;

printf("%d %f\n", x, f);

return 0;

}

f = y + x /= x / y;  
f=y + x /= 1;

f=4/=1;

Now, /= operator and = has same precedence. Hence, associativity will be applied. Now, /= will operate first. But, both side of it becomes integral constant. Hence, it cannot operate. Hence, compilation error.

So, indeed = and /= and %= has same operator precedence. And, I was interpreting the table wrongly.

**Right way of interpreting the same problem:**

Now,=, /=, %= has same precedence.  
  
So, it will be based on associativity.

**100. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 3, y = 2;

int z = x << 1 > 5;

printf("%d\n", z);

}

a) 1

b) 0

c) 3

d) Compile time error

**Answer: a**

Bitwise left shift operator << has higher precedence than > (relational operator).

And, both have higher precedence than = operator.

z=(x<<1)>5;  
z=6>5;

Z=1;

**101. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 3; //, y = 2;

const int \*p = &x;

\*p++;

printf("%d\n", \*p);

}

a) Increment of read-only location compile error

b) 4

c) Some garbage value

d) Undefined behaviour

Now, const int \*p=&x  
  
Hence, p is a pointer which points to memory location of x.

Now, how will const play role, here?

The value of x cannot be changed.

\*p++  
  
Now, ++ has higher precedence than \* (dereference operator)

But, post incrementation is done.

Now, in next line p’s incremented value will be available. Hence, \*p will print garbage value.

**Answer: c**

**102. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 2;

int z = x ^ y & 1;

printf("%d\n", z);

}

a) 1

b) 2

c) 0

d) 1 or 2

**Answer: b**

int z = x ^ y & 1;  
  
Now, bitwise and & has higher precedence than bitwise xor ^.

And, both bitwise operator has higher precedence than =.

So,

z = x ^ y & 1;

z = x ^ (y & 1);

z=x^0;

z=x;

z=2

**103. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 2, y = 0;

int z = x && y = 1;

printf("%d\n", z);

}

a) 0

b) 1

c) Compile time error

d) 2

**Answer: c**

z=0=1 will be in some stage.

Now, for the second assignment operator both side are integral constants. Hence, = wont find any lvalue for performing assignment.

**104. What is the output of the code given below**

#include <stdio.h>

int main()

{

int x = 0, y = 2;

if (!x && y)

printf("true\n");

else

printf("false\n");

}

a) true

b) false

c) compile time error

d) undefined behaviour

**Answer) ! relational not has higher precedence than logical and.**

**a) true.**

**105. What is the output of this C code?**

#include <stdio.h>

int main()

{

int x = 0, y = 2;

int z = ~x & y;

printf("%d\n", z);

}

a) -1

b) 2

c) 0

d) Compile time error

**Answer: b**

Because, bitwise negation has higher precedence than &.

z=~x&y

z=(11111111 11111111 11111111 11111111)& (00000000 00000000 00000000 00000010)

z=00000000 00000000 00000000 00000010

**106. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = 5 \* 3 + 2 - 4;

printf("%d", a);

}

a) 13

b) 14

c) 12

d) 1 6

**Answer: a**

a = 5 \* 3 + 2 - 4  
  
Now, \* has higher precedence.

a = (5 \* 3) + 2 - 4

a=15+2-4

Now, + and - have same precedence. However, associativity will play a role. Both + and - have associativity from left to right.  
  
Now,   
  
a=17-4  
a=13

**107. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = 2 + 4 + 3 \* 5 / 3 - 5;

printf("%d", a);

}

a) 7

b) 6

c) 10

d) 9

Answer: b

\* and / has higher precedence than +,-.

\* and / both operator’s precedence is same. So, associativity will be applicable.   
  
+ and - have same precedence. So, again associativity will play role.  
  
a = 2 + 4 + 3 \* 5 / 3 - 5  
  
a = 2 + 4 + (3 \* 5) / 3 - 5

a = 2 + 4 + 15 / 3 - 5  
  
a = 2 + 4 + 5 - 5

a=(2+4)+ 5 - 5

a=6+5-5

a=11-5

a=6

**108. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = 5 \* 3 % 6 - 8 + 3;

printf("%d", a);

}

a) 10

b) 2

c) -2

d) -3

**Answer: c**

You can do it yourself.

**109. What is the output of this C code?**

#include <stdio.h>

void main()

{

int b = 6;

int c = 7;

int a = ++b + c--;

printf("%d", a);

}

a) Run time error

b) 15

c) 13

d) 14

**Answer: d**

Now, ++(pre increment operator) has higher precedence.

So, b will be incremented first. Now, since, it is pre increment, b’s incremented value will immediately be reflected in memory/cache.  
  
Now, post decrement operator will work. But, since, it is post decrement, the decremented value will be applicable after next sequence point.

**110. What is the output of this C code?**

#include <stdio.h>

void main(

{

double b = 8;

b++;

printf("%lf", b);

}

a) 9.000000

b) 9

c) 9.0

d) Run time error

**Answer: a**

**111. What is the output of this C code?**

#include <stdio.h>

void main()

{

double b = 3 % 0 \* 1 - 4 / 2;

printf("%lf", b);

}

a) -2

b) Floating point Exception

c) 1

d) None of the mentioned

**Answer: b**

Now, % and \* and / has highest precedence. And, their associativity is from left to right.

**112. What is the output of this C code?**

#include <stdio.h>

void main()

{

double b = 5 % 3 & 4 + 5 \* 6;

printf("%lf", b);

}

a) 2

b) 30

c) 2.000000

d) Run time error

**Answer: c**

All values are integral constants. (note that binary plus, %, \*, - etc operator could work even if both side operands are integral constants.)

b=5 % 3 & 4 + 5 \* 6

Now, % and \* has highest precedence. Associativity is left to right.

b=(5 % 3) & 4 + 5 \* 6

b=1 & 4 + 5 \* 6

b=1 & 4 + (5 \* 6)  
  
b=1 & 4 + 30

Now, + has higher precedence than &.

b=1 & (4 + 30);

b=1&34;

b=1;

**113. What is the output of this C code?**

#include <stdio.h>

void main()

{

double b = 3 && 5 & 4 % 3;

printf("%lf", b);

}

a) 3.000000

b) 4.000000

c) 5.000000

d) 1.000000

**Answer: d**

**Operator precedence order:**

%> &> &&

**114. What is the output of this C code?**

#include <stdio.h>

void main()

{

double b = 5 & 3 && 4 || 5 | 6;

printf("%lf", b);

}

a) 1.000000

b) 0.000000

c) 7.000000

d) 2.000000

**Answer: a**

**Operator precedence order:**  
&> |> &&> ||

**115. What is the output of this C code?**

#include <stdio.h>

void main()

{

int k = 0;

double b = k++ + ++k + k--;

printf("%d", k);

}

a) 6

b) 1

c) 5

d) undefined

**Answer: d**

K’s value is modified more than once b/w two consecutive sequence points.

**116. What is the output of this C code?**

#include <stdio.h>

void main()

{

int b = 5 - 4 + 2 \* 5;

printf("%d", b);

}

a) 25

b) -5

c) 11

d) None of the mentioned

**Answer: c**

\* has highest precedence than - and +.

- and + has equal precedence and their associativity is left to right.

**117. What is the output of this C code?**

#include <stdio.h>

void main()

{

int b = 5 & 4 & 6;

printf("%d", b);

}

a) 5

b) 6

c) 3

d) 4

**Answer: d**

Now, bitwise and’s associativity is left to right.

**118. What is the output of this C code?**

#include <stdio.h>

void main()

{

int b = 5 & 4 | 6;

printf("%d", b);

}

a) 6

b) 4

c) 1

d) 0

**Answer: a**

Now, & has higher precedence than |.

**119. What is the output of this C code?**

#include <stdio.h>

void main()

{

int b = 5 + 7 \* 4 - 9 \* (3, 2);

printf("%d", b);

}

a) 6

b) 15

c) 13

d) 21

**Answer: b**

Since,  
(3,2) has a parentheses around it. It will be evaluated first. Now, ‘,’ will act as an operator. It will choose the rightmost value.

b = 5 + 7 \* 4 - 9 \* 2;

Now, \* has highest precedence and has associativity from left to right.

b = 5 + (7 \* 4) - 9 \* 2;

b = 5 +28 - 9 \* 2;

b = 5 +28 - 18;  
  
b = 5 +10;

b=15

**120. What is the output of this C code?**

#include <stdio.h>

void main()

{

int h = 8;

int b = (h++, h++);

printf("%d%d\n", b, h);

}

a) 10 10

b) 10 9

c) 9 10

d) 8 10

Answer: c

Now, (h++,h++) ‘;’ will act as an operator as well as sequence point.

So, left expression will be evaluated first. Now, h++ is done. Now, since, post increment operator is applied, the incremented value of h will be applicable after the next sequence point, which is the comma here.

Now, when right side expression h++ is evaluated, h’s value is 9. Now, again, post increment operator is applied. Hence, incremented value will be applicable after next sequence point.

h is 10  
b is 9

**121. What is the output of this C code?**

#include <stdio.h>

void main()

{

int h = 8;

int b = h++ + h++ + h++;

printf("%d\n", h);

}

a) 9

b) 10

c) 12

d) 11

**Answer: d**

Now, h is modified more than once between two consecutive sequence point. Hence, in low level compiler it can provoke undefined behaviour.

Otherwise, it is clearly is 11. (actually undefined behaviour would happen, if we mix pre increment operator and post increment operator )

**122. What is the output of this C code?**

#include <stdio.h>

void main()

{

int h = 8;

int b = 4 \* 6 + 3 \* 4 < 3 ? 4 : 3;

printf("%d\n", b);

}

a) 3

b) 33

c) 34

d) Run time error

Answer: a

b = 4 \* 6 + 3 \* 4 < 3 ? 4 : 3

Now, ternary operator has lowest precedence.

Now, operators are playing role only in left side of ?.

Now, 4 \* 6 + 3 \* 4 <3  
  
Now, \* operator has higher precedence and has associativity from left to right.

**Precedence order:**

‘\*’ > ‘+’ > ‘<’

Now,

It will be evaluated as 36< 3.

Now, since, ?’s left side expression is evaluated to 0 values.  
  
**b will be 3.**

**123.What is the output of this C code?**

#include <stdio.h>

void main()

{

int h = 8;

int b = 4 \* 6 + 3 \* 4 ? 4 : 3;

printf("%d\n", b);

}  
a) 4  
b) 3  
c) Compilation error

d) undefined behaviour

**Answer) a) 4**  
  
Now, ?’s left side will be evaluated to non zero value. Hence, b is 4

**124.What is the output of this C code?**

#include <stdio.h>

void main()

{

int h = 8;

int b = 4 \* 6 + 3 \* 4 ? 4+3 : 3;

printf("%d\n", b);

}  
a)7  
b)3  
c) Compilation error

d) undefined behaviour

Now, operator precedence order:

\* > + > ?:

Now, \* has associativity from left to right. So has +.

b = 4 \* 6 + 3 \* 4 ? 4+3 : 3;

b = (4 \* 6) + 3 \* 4 ? 4+3 : 3;

b = 24 + 3 \* 4 ? 4+3 : 3;

b = 24 + (3 \* 4) ? 4+3 : 3;

b =24+12 ? 4+3 : 3;

b=(24+12)? 4+3: 3; (Now, + has associativity from left to right. Also, ? is ?: acts as a sequence point. That means the left side expression will be evaluated first and all side effects of previous expressions or statements would be executed/ completed before this point and all the side effects of the subsequent expressions would not start their evaluation.)

b=36? 4+3: 3;

b=36? (4+3): 3;

b=36? 7: 3;

b=7

**125. What is the output of this C code?**

#include <stdio.h>

void main()

{

int a = 2 + 3 - 4 + 8 - 5 % 4;

printf("%d\n", a);

}

a) 0

b) 8

c) 11

d) 9

**Answer: b**

**126. What is the output of this C code?**

#include <stdio.h>

void main()

{

char a = '0';

char b = 'm';

int c = a && b || '1';

printf("%d\n", c);

}

a) 0

b) a

c) 1

d) m

**Answer: c**

**127. What is the output of this C code?**

#include <stdio.h>

void main()

{

char a = 'A';

char b = 'B';

int c = a + b % 3 - 3 \* 2;

printf("%d\n", c);

}

a) 65

b) 58

c) 64

d) 59

**Answer: d**

**128. Which of the following operators has an associativity from Right to Left?**

a) <=

b) <<

c) ==

d) +=

**Answer: d**

**129. Which operators of the following have same precedence?**

P. "!=", Q. "+=", R. "<<="

a) P and Q

b) Q and R

c) P and R

d) P, Q and R

**Answer: b**

**130. Comment on the following statement?**

n = 1;

printf("%d, %dn", 3\*n, n++);

a) Output will be 3, 2

b) Output will be 3, 1

c) Output will be 6, 1

d) Output is compiler dependent

**Answer: d**

**131. Which of the following option is the correct representation of the following code?**

e = a \* b + c / d \* f;

a) e = (a \* (b +(c /(d \* f))));

b) e = ((a \* b) + (c / (d \* f)));

c) e = ((a \* b) + ((c / d)\* f));

d) Both e = ((a \* b) + (c / (d \* f))); and e = ((a \* b) + ((c / d)\* f));

**Answer: d**

Explanation: Verified by e = 1 \* 2 + 3 / 4 \* 5; and then using respective braces according to the option.

**132. What care must be taken during swapping 2 numbers?**

b = (b / a);

a = a \* b;

b = a / b;

a) Data type should be either of short, int and long

b) Data type should be either of float and double

c) All data types are accepted except for (char \*)

d) This code doesn’t swap 2 numbers

**Answer: b**

**133. What should be the output of the following program:**

#include<stdio.h>

int main()

{

int a = 1, b = 2, c = 3, d = 4, e;

e = c + d = b \* a;

printf("%d, %d\n", e, d);

}

a) 7, 4

b) 7, 2

c) 5, 2

d) Syntax error

**Answer: d**

**134. Which of the following is the correct order of evaluation for the given expression?**

a = w % x / y \* z;

a) % / \* =

b) / \* % =

c) = % \* /

d) \* % / =

**Answer: a**

**135. Which function in the following expression will be called first?**

a = func3(6) - func2(4, 5) / func1(1, 2, 3);

a) func1();

b) func2();

c) func3();

d) Cannot be predicted

**Answer: d**

Though, operator precedence will play a rule. Like / will operate first.

We cannot control the order of function calling b/w func2 and func3.

**136. Which of the following operator has the highest precedence in the following?**

a) ()

b) sizeof

c) \*

d) +

**Answer: a**

**137. Which of the following is a ternary operator?**

a) &&

b) >>=

c) ?:

d) ->

**Answer: c**

**138.Which of the following are unary operators?**

a) sizeof

b) –

c) ++

d) all of the mentioned

**Answer) d)**

**139. Where in C the order of precedence of operators do not exist?**

a) Within conditional statements, if, else

b) Within while, do-while

c) Within macro definition

d) None of the mentioned

**Answer: d**

**140. Associativity of an operator are:**

a) Right to Left

b) Left to Right

c) Random fashion

d) Both Right to Left and Left to Right

Answer) it could be either right to left or left to right. **Hence, d**

**141. Which of the following method are accepted for assignment?**

a) 5 = a = b = c = d;

b) a = b = c = d = 5;

c) a = b = 5 = c = d;

d) None of the mentioned

**Answer) b) In case of a and c, both will generate compilation error.**

**142. Which of the following is NOT possible with any 2 operators in C?**

a) Different precedence, same associativity

b) Different precedence, different associativity

c) Same precedence, different associativity.

d) All of the mentioned

**Answer) c) Same precedence , different associativity.**

**143. Which of the following is possible with any 2 operators in C?**

a) Same associativity, different precedence

b) Same associativity, same precedence

c) Different associativity, different precedence

d) All of the mentioned

**Answer) d)**

**144. Which of the following operators has the lowest precedence?**

a) !=

b) &&

c) ?:

d) ,

Answer) d) ,

**145. Comment on the output of this C code?**

#include <stdio.h>

int main()

{

int x = 3, i = 0;

do {

x = x++;

i++;

} while (i != 3);

printf("%d\n", x);

}

a) Undefined behaviour

b) Output will be 3

c) Output will be 6

d) Output will be 5

**Answer)c**

**146. What is the output of this C code?**

#include <stdio.h>

int main()

{

int a = -1, b = 4, c = 1, d;

d = ++a && ++b || ++c;

printf("%d, %d, %d, %d\n", a, b, c, d);

return 0;

}

a) 0, 4, 2, 1

b) 0, 5, 2, 1

c) -1, 4, 1, 1

d) 0, 5, 1, 0

**Answer) a)**

**148. What is the output of this C code?**

#include <stdio.h>

int main()

{

int p = 10, q = 20, r;

if (r = p = 5 || q > 20)

printf("%d", r);

else

printf("No Output\n");

}

a) 1

b) 10

c) 20

d) No Output

**Answer) a)**