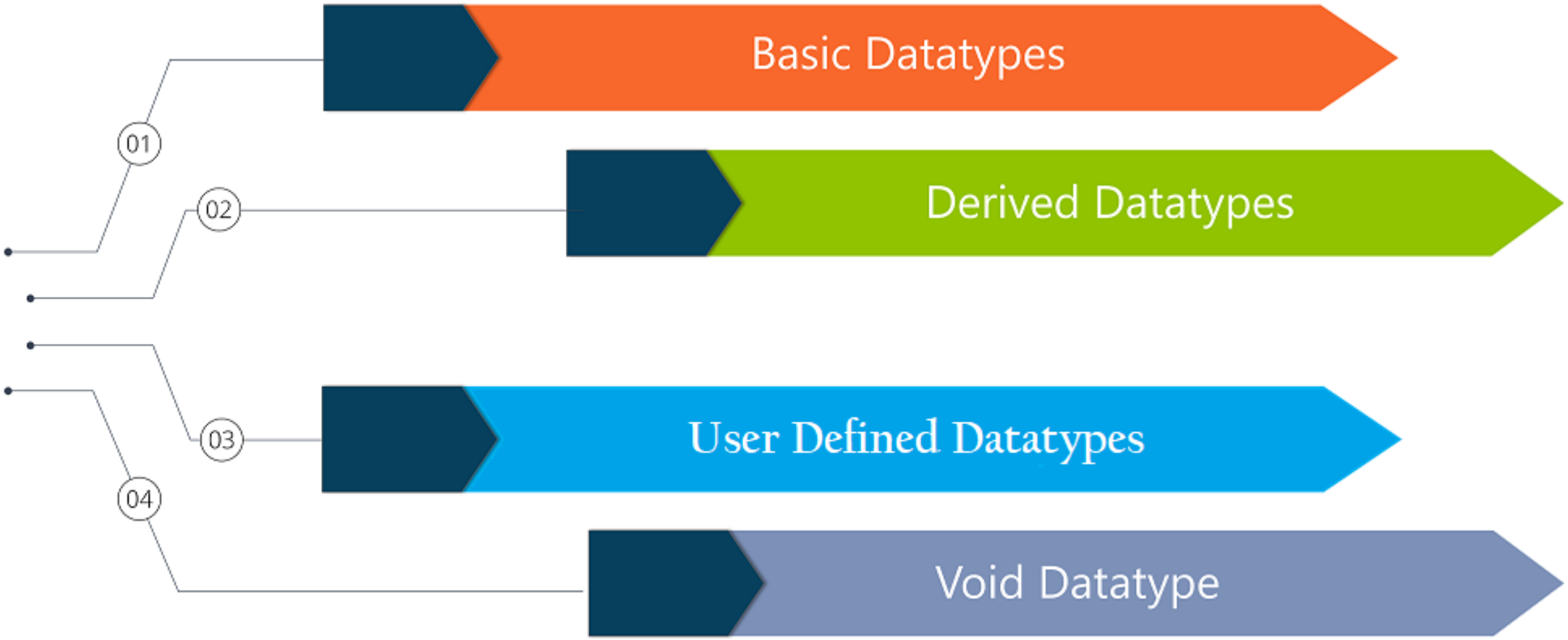
Datatypes and Operators

Data Types


01
Basic Datatypes
02
Derived Datatypes
03
User Defined Datatypes
04
Void Datatype

Data Types –

- Data Type is used to define the type of value to be used in a Program. Based on the type of value specified in the program, the amount of Bytes(memory) will be allocated to the variables used in the program.
- Data types are broadly classified into four main types. They are:
  - Basic (or) Primary data type (Fundamental Data Types)
  - Derived data type
  - User defined data type.
  - Void data type
- **Primary Data Type**
  - Integers are represented as int, character are represented as char, floating point value are represented as float, double precision floating point are represented as double and finally void are primary data types.
  - Primary data type offers extended data types. Long int, long double are extended data types.

Integer Data Type

- Integer data type can store only the whole numbers.

Name	C Representation	Size(bytes)	Range	Format Delimiter
Integer	Int	2	-32768 to 32767	**%d**
Short Integer	short int / short	2	-32768 to 32767	**%d**
Long Integer	long int / long	4	-2147483648 to 2147483647	**%ld**
Signed Integer	signed int	2	-32768 to 32767	**%d**
Unsigned Integer	unsigned int	2	0 to 65535	**%u**
Signed Short Integer	signed short int / short	2	-32768 to 32767	**%d**
Unsigned Short Integer	unsigned short int / short	2	0 to 65535	**%u**
Signed Long Integer	signed long int / long	4	-2147483648 to 2147483647	**%ld**
Unsigned Long Integer	unsigned long int / long	4	0 to 4294967295	**%lu**

Floating Point Data Type

- Floating Point data types are also known as Real Numbers. It can store only the real numbers.

Name	C Representation	Size (bytes)	Range	Format Delimiter
Float	float	4	3.4 e-38 to 3.4 e+38	**%f**
Double	double	8	1.7e-308 to 1.7e+308	**%f**
Long Double	long double	10	3.4 e-4932 to 3.4 e+4932	**%lf**

Character Data Type

- Normally a single character is defined as char data type. It is specified by the keyword char. Char data type uses 8 bits for storage. Char may be signed char or unsigned char.

Name	C Representation	Size(bytes)	Range	Format Delimiter
Character	char	1	-128 to 127	**%c**
Signed Character	signed char	1	-128 to 127	**%c**
Unsigned Character	unsigned char	1	0 to 255	**%c**

Void Datatypes

- ✓ The void data type is an empty data type that is used as a return type for the functions that return no value in C.

Example:

- void function(int n)
- int function(void)

Limits of Data Types –

```
#include <stdio.h>
#include <limits.h>
void main()
{
   printf("CHAR_BIT    :   %d\n", CHAR_BIT);
   printf("CHAR_MAX    :   %d\n", CHAR_MAX);
   printf("CHAR_MIN    :   %d\n", CHAR_MIN);
   printf("INT_MAX     :   %d\n", INT_MAX);
   printf("INT_MIN     :   %d\n", INT_MIN);
   printf("LONG_MAX    :   %ld\n", (long) LONG_MAX);
   printf("LONG_MIN    :   %ld\n", (long) LONG_MIN);
   printf("SCHAR_MAX   :   %d\n", SCHAR_MAX);
   printf("SCHAR_MIN   :   %d\n", SCHAR_MIN);
```

Limits of Data Types –

```
printf("SHRT_MAX   :   %d\n", SHRT_MAX);
printf("SHRT_MIN   :   %d\n", SHRT_MIN);
printf("UCHAR_MAX  :   %d\n", UCHAR_MAX);
printf("UINT_MAX   :   %u\n", (unsigned int) UINT_MAX);
printf("ULONG_MAX  :   %lu\n", (unsigned long) ULONG_MAX);
printf("USHRT_MAX  :   %d\n", (unsigned short) USHRT_MAX);
}
```

Output:

```
CHAR_BIT   :   8
CHAR_MAX   :   127
CHAR_MIN   :   -128
INT_MAX    :   2147483647
INT_MIN    :   -2147483648
LONG_MAX   :   9223372036854775807
LONG_MIN   :   -9223372036854775808
SCHAR_MAX  :   127
SCHAR_MIN  :   -128
SHRT_MAX   :   32767
SHRT_MIN   :   -32768
UCHAR_MAX  :   255
UINT_MAX   :   4294967295
ULONG_MAX  :   18446744073709551615
USHRT_MAX  :   65535
```

Cont..

- **Derived Data Types –**
  - These data types derived from the basic data types. Derived data types available in C are:
    - Array Type. Ex: char[], int[]….
    - Pointer Type. Ex: char*, int*, etc
    - Function Type. Ex: int(int,int), float(int), etc
- **User Defined Data Types –**
  - The C language provides flexibility to the user to create new data types. These newly created data types are called User Defined Data Types. In C these can be created as:
    - Structures.
    - Unions
    - Enumeration.

Storage Classes –

- In C language, each variable has a storage class which decides scope, visibility and lifetime of that variable. The following storage classes are most frequently used in C programming.

1. Automatic variables
2. External variables
3. Static variables
4. Register variables

Storage Classes –

Automatic variables A variable declared inside a function without any storage class specification, is by default an automatic variable.

- They are created when a function is called and are destroyed automatically when the function exits.
- Automatic variables can also be called local variables because they are local to a function. By default they are assigned garbage value by the compiler.

```
main()
{
    int detail;
    //or
    auto int detail; //Both are same
}
```

Storage Classes –

- **External variables:** Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.
- Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.
- The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file.

Storage Classes –

First File: main.c

```
#include <stdio.h>
extern i;
main()
{
printf("value of the external integer is = %d\n", i);
}
```

Second File: original.c

```
#include <stdio.h>
i=48;
```

Result:

```
value of the external integer is = 48
```

Storage Classes –

Static variables Instead of creating and destroying a variable every time when it comes into and goes out of scope, static is initialized only once and remains into existence till the end of program.

- A static variable can either be internal or external depending upon the place of declaration.
- Scope of internal static variable remains inside the function in which it is defined.
- External static variables remain restricted to scope of file in which they are declared.
- They are assigned 0 (zero) as default value by the compiler.

Storage Classes –

```
#include<stdio.h>
void test(); //Function declaration
int main()
{
    test();
    test();
    test();
    return 0;
}
void test()
{
    static int a = 0; //Static variable
    a = a+1;
    printf("%d\t",a);
}
```

Output:

1 2 3

Storage Classes –

Register variables Register variable inform the compiler to store the variable in CPU registers instead of memory.

- Register variable has faster access than normal variable.
- Frequently used variables are kept in register.
- Only few variables can be placed inside register.
- NOTE : We can never get the address of such variables.
- Syntax :
  - register int number;

Storage Classes –

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block
extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Memory	Zero	Global	program runtime

C Scopes

- Scope means: The visibility of the variable in the block/program.
- Visibility means accessibility (or) area of the program we can access the variable.
- There are four types:
  - File scope or Global Scope
  - Block scope or Local scope
  - Function prototype scope
  - Function scope

File Scope/ Global Scope – Identifier is said to have global scope / file scope if it is defined outside the function and whose visibility is entire program

Block Scope/ Local Scope - If an identifier has scope only within the area where it is declared then the scope of the Identifier is called Block Scope

C Scopes

- **Global Scope**: Can be accessed any where in a program.

```
int a;//global variable
main()
{
      a = 2;
      fun1();
}
void fun1()
{
      printf("%d",a);
}
```

C Scopes

- **Block Scope**: A Block is a set of statements enclosed within left and right braces. Blocks may be nested in C (a block may contain other blocks inside it).
- A variable declared in a block is accessible in that block and all inner blocks of that block, but not accessible outside the block.
- What if the inner block itself has one variable with the same name?
  - If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the pint of declaration by inner block.

C Scopes

```
void main()
{
    int x = 10, y = 20;
    {
        // The outer block contains declaration of x and y, so following statement is valid
        and prints 10 and 20
        printf("x = %d, y = %d\n", x, y);
        {
            // y is declared again, so outer block y is not accessible in this block
            int y = 40;
            x++; // Changes the outer block variable x to11
            y++; // Changes this block's variable y to 41
            printf("x = %d, y = %d\n", x, y);
        }
        // This statement accesses only outer block's variables
        printf("x = %d, y = %d\n", x, y);
    }
}
```

Output:

```
x = 10, y = 20
x = 11, y = 41
x = 11, y = 20
```

C Scopes

Function Prototype Scope - An identifier that appears within a function prototype's list of parameter declarations has *function prototype scope*. The scope of such an identifier begins at the identifier's declaration and terminates at the end of the function prototype declaration list.

For example:

```
int students ( int david, int susan, int mary, int john );
```

- In this example, the identifiers (david, susan, mary , and john) have scope beginning at their declarations and ending at the closing parenthesis. The type of the function students is "for the actual parameter names to be used function returning int with four int parameters." In effect, these identifiers are merely placeholders after the function is defined.

C Scopes

- **Function Scope –** A *label* is the only kind of identifier that always has **Function Scope.** It can be used any where within the function in which an identifier - labeled statement specifying the label name appears.

For example:

```
int func1(int x, int y, int z)
{
        label: x += (y + z); /* label has function scope */
        if (x > 1)
                goto label;
}
int func2(int a, int b, int c)
{
        if (a > 1)
                goto label; /* illegal jump to undefined label */
}
```

Formatted Input and Output

Formatted Output: printf

- printf() provides the formatted output conversion.
- The syntax is as follows:
  - int ptintf(char *format,...)
- It returns number of characters it printed.
- ... indicated the variable number of arguments.

  example:

  printf("control string %s",str);
- It takes two types of arguments

  1. Ordinary characters
  2. Conversion Specifications %

Formatted Input and Output

CHARACTER	ARGUMENT TYPE	PRINTED AS
d	int	decimal number
O	int	unsigned octal number
x,X	int	unsigned hexa decimal number
u	int	unsigned decimal number
c	char	single character
s	char*	print characters from the string
f	float	Fractional number

Formatted Input and Output

Formatted Input: scanf

- scanf() provides the formatted input conversion.
- The syntax is as follows:
  - int scanf(char *format,...)
- It returns number of inputs it has read.
- ... indicated the variable number of arguments.

example:

```
scanf("%s",&str);
```

Reading and Writing Characters

- In C language many programs processing character data as text input and text output like a stream of characters.

- In C language the standard input and output library provides functions for reading and writing characters at a time.
  - getchar(): Reading next character from keyboard
  - putchar(): Writing a character each time it is called.
  - getchar() and putchar() are similar to scanf() and printf() respectively.

Formatted Input and Output

Example write a c Program that copies input to its output one character at a time;

```
#include<stdio.h>
int main()
{
    int c;
    c=getchar();
    while(c!=EOF)
    {
        putchar(c);
        c=getchar();
    }
}
```

Operators

Classification of Operators

Classification Based on Number of Operands

Based upon the number of operands on which an operator operates, the operators are classified as:

- **Unary Operators**: A unary operator operates on only one operand. For example, in the expression -3, - is a unary minus as it operates on only one operand.
- **Binary Operators:** A binary operator operates on two operands. It requires an operand towards left and right. Example 3-4.
- **Ternary Operator:** It operates on three operands. Conditional operator is the only ternary operator in C.

Operators based on classification

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Conditional Operators
6. Special Operators (Miscellaneous Operators)
7. Bitwise operators
8. Increment and decrement operators
9. Unary operators
10. Equality operators

Miscellaneous Operators

1. Function call operator (())
2. Array subscript operator ([])
3. Member Selection operators
    1. Direct Member access operator (Dot Operator(.))
    2. Indirect member access operator (Arrow Operator (->))
4. Indirection operator (*)
5. Conditional operator
6. Comma operator
7. sizeof operator.
8. Address operator (&)

Arithmetic Operators

Arithmetic Operators

Operator	Meaning	Details
+	Addition	Performs addition on integer numbers, floating point numbers. The variable name which is used is the operand and the symbol is operator.
-	Subtraction	Subtracts one number from another.
*	Multiplication	Used to perform multiplication
/	Division	It produces the Quotient value as output.
%	Modulo	It returns the remainder value as output.

/*Program to demonstrate arithmetic operators in C */

```
#include <stdio.h>
main()
{
        int a=9,b=4,c;
        c=a+b;
        printf("a+b=%d\n",c);
        c=a-b;
        printf("a-b=%d\n",c);
        c=a*b;
        printf("a*b=%d\n",c);
        c=a/b;
        printf("a/b=%d\n",c);
        c=a%b;
        printf("Remainder when a divided by b=%d\n",c);
}
```

Relational Operators

Relational Operators

- Relational Operators are used to compare two same quantities.
- The general form is
  - **(exp1 relational operator exp2)**
- There are six relational operators. They are mentioned as follows.

Operator	Meaning	Example
<	is less than	5<3 returns false (0)
<=	is less than or equal to	5<=3 return false (0)
>	is greater than	5>3 returns true (1)
>=	is greater than or equal to	5>=3 returns true (1)
= =	is equal to	5==3 returns false (0)
!=	is not equal to	5!=3 returns true(1)

Relational Operators

```
#include <stdio.h>
main()
{
        int a = 21, b = 10, c ;
        if( a == b )
                printf(“ a is equal to b\n" );
        else
        {
                printf("a is not equal to b\n" );
        }
        if ( a < b )
                printf("a is less than b\n" );
        else
                printf("a is not less than b\n" );
        if ( a > b )
                printf("a is greater than b\n" );
        else
                printf("a is not greater than b\n" );
        /* Lets change value of a and b */
        a = 5;
        b = 20;
        if ( a <= b )
                printf("a is either less than or equal to  b\n" );
        if ( b >= a )
              printf("b is either greater than  or equal to b\n" );
}
```

Relational Operators

Output:

a is not equal to b

a is not less than b

a is greater than b

a is either less than or equal to b

b is either greater than or equal to b

Equality Operators

- **E**quality operator (=) is used together with condition. The value of the expression is one or zero. If the expression is true the result is one, if false result is zero.

Operator	Meaning
= =	Equal to
!=	Not equal to

Logical Operators

Logical Operators

- Logical Operators are used when we need to check more than one condition.
- Logical Operators are used in decision making.
- Logical expression yields value 0 or 1.

Logical Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical Operators

Logical AND (&&)

The result of the Logical AND operator will be TRUE If both value is TRUE otherwise the result will be always False.

Eg: a > b && x = = 10

The expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

Logical OR (||)

If any of the expression is true the result is true else false otherwise. The result is similar to basic Binary addition.

Eg: a < m || a < n

It evaluates to true if a is less than either m or n and when a is less than both m and n.

Logical NOT (!)

It acts upon single value. If the value is true result will be false and if the condition is false the result will be true.

Eg (!a)

Example

```
#include <stdio.h>
 int main()
{
      int age=65;   char status = ‘L’;
      if (age >= 60 && status == ‘L’)//&& Operator : Both conditions are true
          printf(“Issue pension \n");
    else
          printf(“Don’t Issue pension \n");
}
```

Increment and Decrement Operators

What are increment and decrement operators????

- Increment operator is used to increment the value of a variable by one.
- Similarly Decrement operator is used to decrement the value of a variable by one.

For example

Increment

```
int a=5;
a++;
```

a = 6

Decrement

```
int a=5;
a--;
```

a = 4

a++; is same as a = a+1;

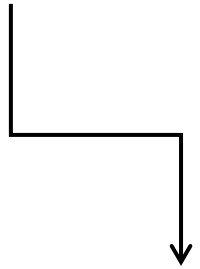
a - -; is same as a = a-1;

Increment and Decrement Operators

- Both are Unary operators.
  - Because they are applied on single operand.
    - a++;
    - a ++ b (Not correct)

Increment and Decrement Operators

```
a ++;
a = a + 1;
(a+b)++;
(a+b) = (a+b)+1;  (Not possible)
```



Compiler is expecting a variable as an increment operand but we are providing an expression (a+b) which does not have the capability to store the data

Types of Increment and Decrement operators

Pre – Increment operator

++a

Post – Increment operator

a++

Pre – Decrement operator

- - a

Post – Decrement operator

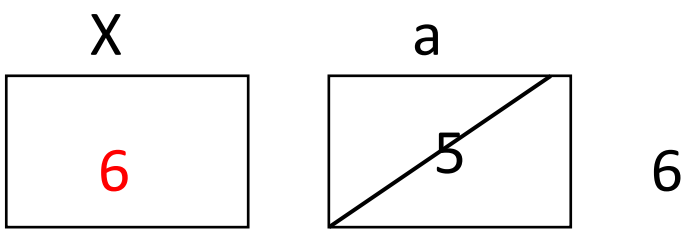
a - -

Question: what is the difference between pre-increment and post-increment operator OR pre-decrement and post-decrement operator????

- Pre – means first increment/decrement then assign it into another variable.
- Post – means first assign it into another variable then increment/decrement.

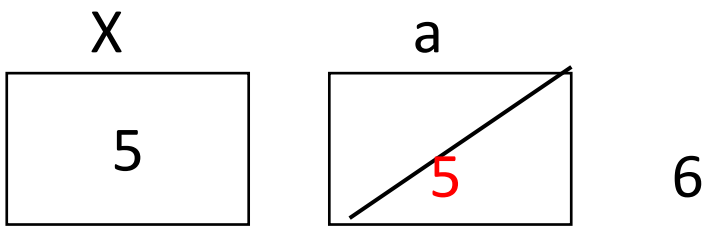
Example: Let us consider a = 5

X = ++ a;



X = 6, a=6

X = a ++;



X = 5, a=6

C program to perform increment operation

```
#include<stdio.h>
void main( )
{
   int a,b,x=10,y=10;
   a = x++;
   b = ++y;
   printf(“The value of a is: %d”,a);
   printf(“The value of b is: %d”,b);
}
```

Output

The value of a : 10

The value of b : 11

Decrement Operators

- It is used to decrease the value of the Operand by 1. There are two types of Decrement Operators in C Language. They are pre decrement operator and post decrement operator.

Prefix decrement Operator

- It is used to decrease the value of the variable by 1. Here the value of 1 is subtracted to the variable first along with the given variable value.
- **Eg:** --g ------> prefix decrement (Decrement g, then evaluate g)
- int x = 5;
- int y = --x; // x is now equal to 4, and 4 is assigned to y

Postfix Decrement Operator

- It is used to decrease the value of the variable by 1. Here the value of 1 is subtracted to the variable first along with the given variable value.
- **Eg:** g-- ----> postfix decrement (Evaluate g, then decrement g)
- int x = 5;
- int y = x--; // x is now equal to 4, and 5 is assigned to y

C program to perform decrement operation

```
#include<stdio.h>
void main( )
{
   int a,b,x=10,y=10;
   a = x--;
   b = --y;
   printf(“The value of a is: %d”,a);
   printf(“The value of b is: %d”,b);
}
```

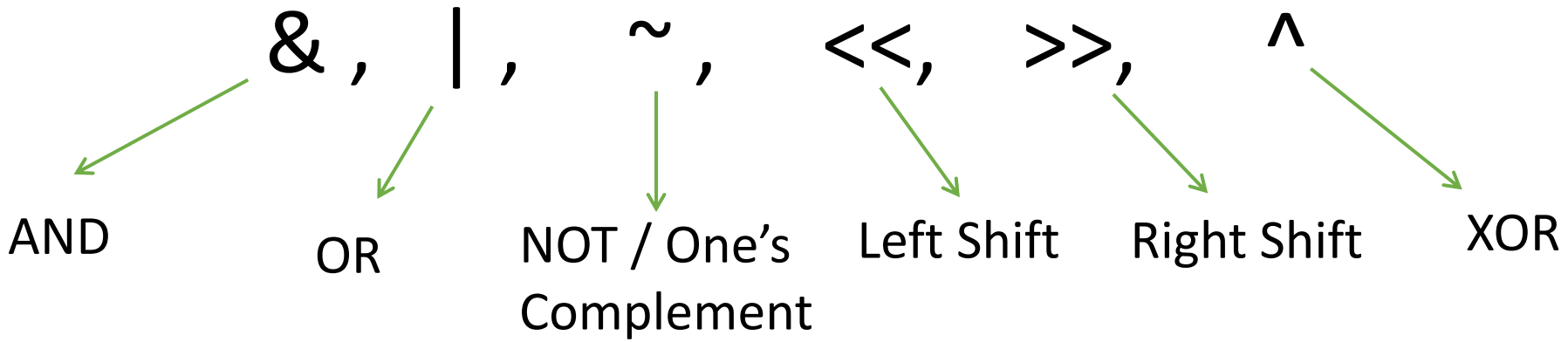
Output

The value of a : 10

The value of b : 9

Bitwise Operators

Bitwise Operators



As the name suggests - it will perform its operations on the individual bits in the given value rather than the whole value

Bitwise AND (&) operator

- Bitwise AND(&) is a binary operator.
- It takes two bits at a time and perform AND operation .
- Bitwise AND returns 1 when both the bits are 1.

For Example: 7 & 4

7 → 0 1 1 1

4 → 0 1 0 0

0 1 0 0

7 & 4 = 4

Bitwise OR (|) operator

- It takes two bits at a time and perform OR operation .
- Bitwise OR returns 0 when both the bits are 0. Otherwise it returns 1.

For Example: 7 | 4

7 → 0 1 1 1

4 → 0 1 0 0

0 1 1 1

7 | 4 = 7

Bitwise XOR (^) operator

- It takes two bits at a time and perform XOR operation.
- Bitwise XOR returns 1 when two bits are different. Otherwise it returns 0

For Example: 7 ^ 4

7 → 0 1 1 1

4 → 0 1 0 0

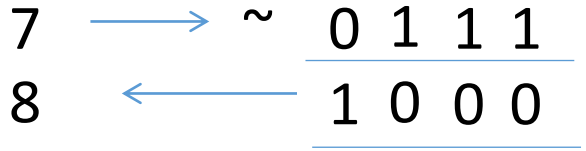
0 0 1 1

7 ^ 4 = 3

Bitwise NOT (~) operator

- Bitwise NOT (~) is a Unary operator.
- Its job is to complement each bit one by one.
- Result of NOT is 0 when the bit is 1, and the result is 1 when the bit is 0.

For Example: ~ 7



~ 7 = 8

Difference between bitwise and logical operators

```
#include<stdio.h>
int main()
{
        char x=1, y=2; //x=1(0000 0001),
                        //y=2 (0000 0010)
        if(x&y)             //1&2 = 0 (0000 0000)
                printf("Result of x & y is 1");
        if(x&&y)    //1&&2 = TRUE  && TRUE = TRUE =1
                printf("Result of x && y is 1");
}
```

Bitwise Left Shift (<<) operator

Left Shift << operator is a binary operator.

First operand << Second operand

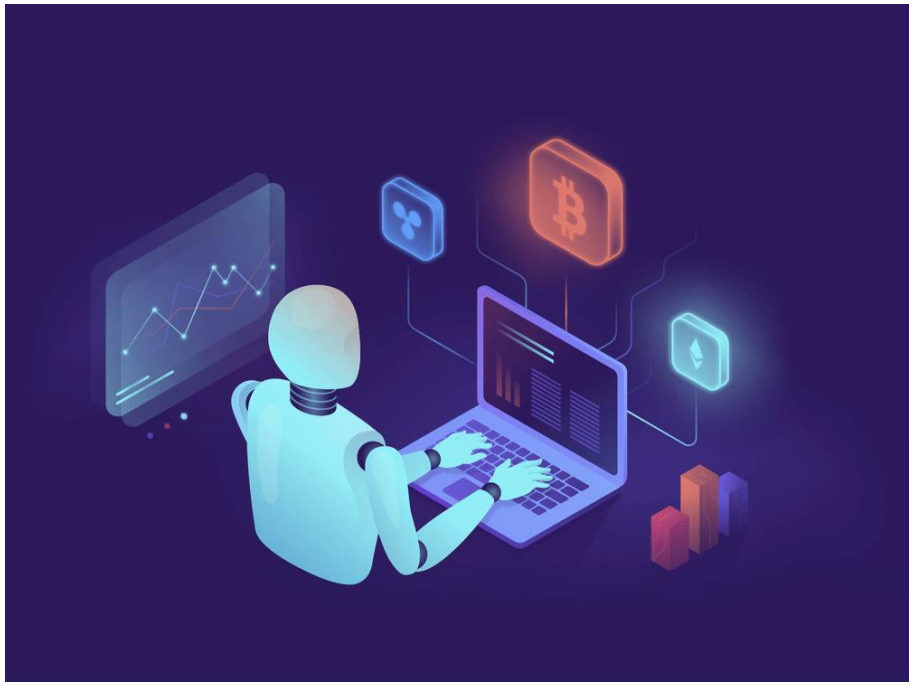
Whose bits get left shifted

Decides the number of places to shift the bits

When bits are shifted left then trailing positions are filled with zeros.

How left shift works???

```
#include<stdio.h>
int main()
{
        int v= 3;    //  3 in binary = 0000 0011
        printf("%d",v<<1);
        return 0;
}
```

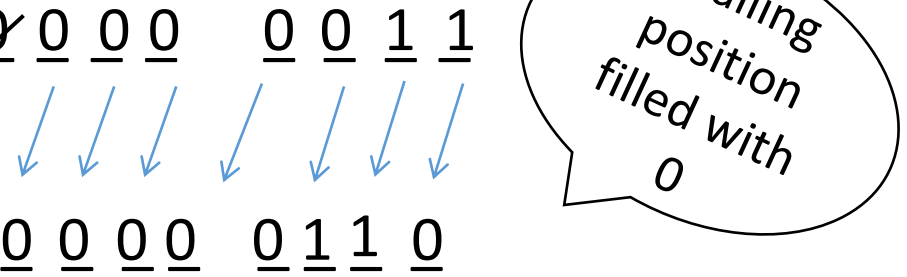


How left shift works???

V= 3

V << 1

Left shift by one position



Result of v << 1 = 6

How left shift works???

- Left shifting is equivalent to left operand is multiplied by 2 power(right operand)

Example:

V= 3

V << 1

1 is our right operand

Result of V << 1 = [3 X 2^{1}] = 6

3 is our left operand

Result of V << 2 = [3 X 2^{2}] = 12

Bitwise right shift (>>) operator

Right Shift >> operator is a binary operator.

First operand >> Second operand

Whose bits get right shifted

Decides the number of places to shift the bits

When bits are shifted right then leading positions are filled with zeros.

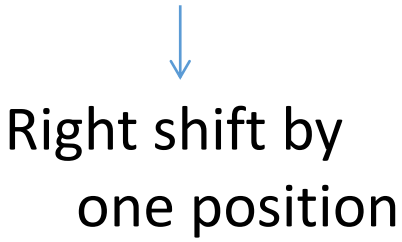
How right shift works???

```
#include<stdio.h>
int main()
{
        int v= 3;    //  3 in binary = 0000 0011
        printf("%d",v >> 1);
        return 0;
}
```

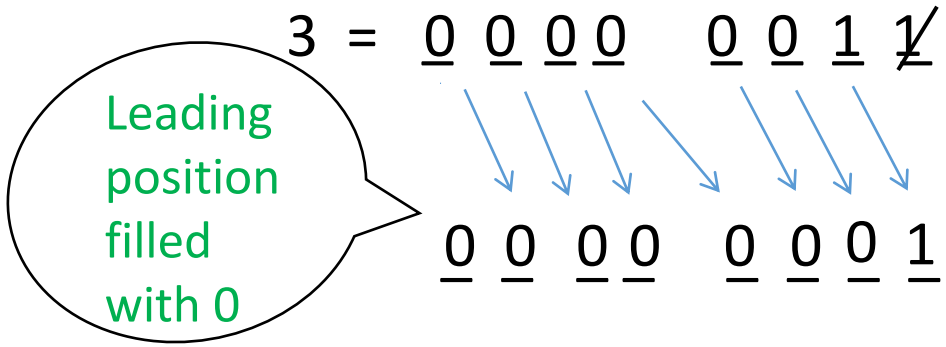
How right shift works ???

V= 3

V >> 1



Right shift by one position



3 = 0 0 0 0 0 0 1 1

0 0 0 0 0 0 0 1

Result of v >> 1 = 1

Contd..

- Right shifting is equivalent to division by 2 power(right operand)

Example:

V= 3

V >> 1

1 is our right operand

Result of V >> 1 = [3 / 2^{1}] = 1

3 is our left operand

V= 32

Result of V >> 2 = [32 / 2^{2}] = 8

Assignment Operators

Assignment Operators

The Assignment Operator evaluates an expression, from the right of the expression and substitutes it to the variable on the left of the expression. The general form is

identifier = expression;

Eg: x = a + b;

(i) Simple assignment operator

In Simple assignment operator only '=' equal to operator is used. The general form is

identifier = expression;

Eg: y=35; x=ax+b-c;

(ii) Shorthand assignment operator

Shorthand assignment operator must be an arithmetic operator or bitwise operator. The general form is

identifier <operator> = expression;

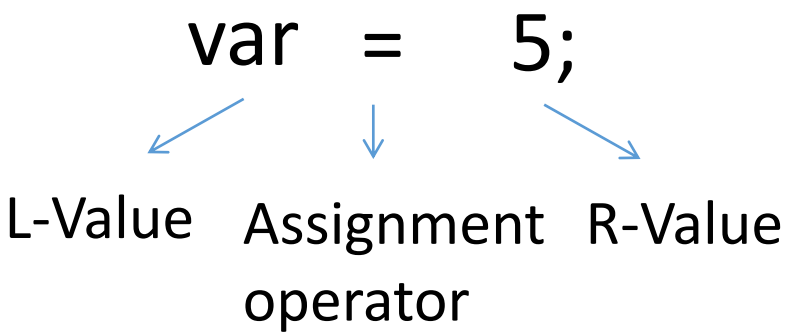
Assignment operator (=)

Values to a variable can be assigned using Assignment operator

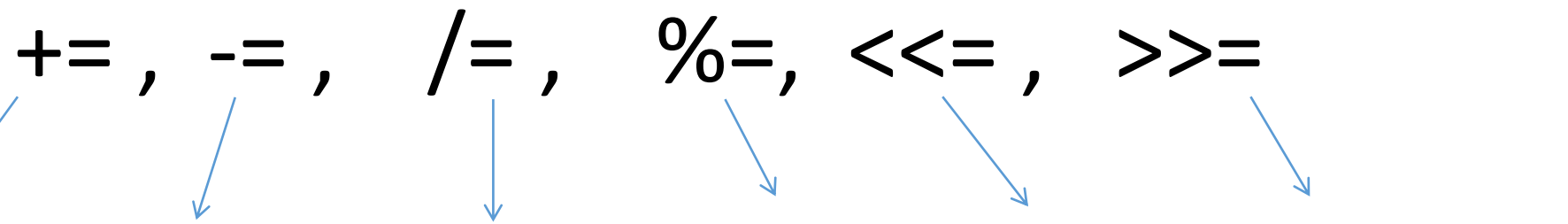
Assignment operator requires two operands (L-value and R- Value)

This operator copies R-values into L-value.

Example:



Shorthand Assignment Operators



a += 1 ➔ to a = a+1

a -= 1 ➔ to a = a - 1

a /= 2 ➔ to a = a / 2

a %= 2 ➔ to a = a % 2

a <<= 1 ➔ to a = a << 1

a >>= 2 ➔ to a = a >> 2

Shorthand assignment operator

Operator	Meaning	Simple. Assign	Shorthand
+=	Assign sum	x = x + 1	x+ =1
-=	Assign difference	y = y - 1	y-=1
*=	Assign product	z=z*(x+y)	z*=(x+y)
/=	Assign quotient	y=y/(x+y)	y/=(x+y)
%=	Assign remainder	x=x%z	x%=z
~=	Assign one's complement		
<<=	Assign left shift	x = x << z	x << = z
>>=	Assign right shift		
&=	Assign bitwise AND	y = y&x	y&=x
|=	Assign bitwise OR		
^=	Assign bitwise X - OR	z = z^y	z^=y

Conditional Operators

Conditional Operator

? :

Conditional operator is also called as a Ternary operator. Therefore it requires three operands.

Syntax:

(expression1) ? (expression 2) : (expression3);

Why do we need this conditional operator ??

```
char result;
int marks;
if(marks>18)
   result='P';
else
  result = 'F';
Printf(" %c", result);
```

```
char result;
int marks;
result = (marks>18) ? 'P' : 'F';
Printf(" %c", result);
```

Cont..

False

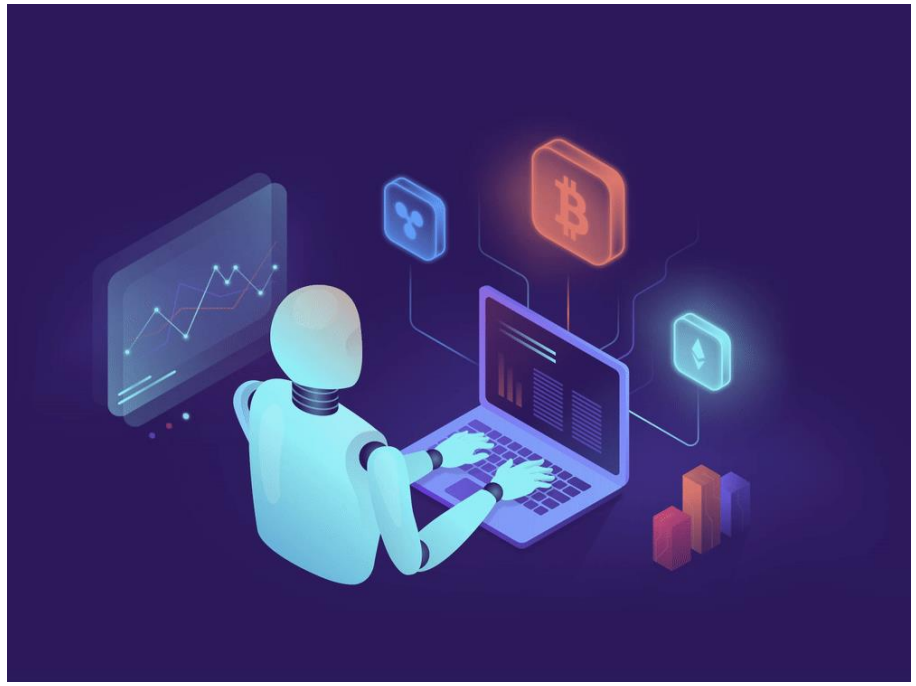
result = (marks>18) ? ‘P’ : ‘F’;

True

result = P

result = F

This expression is called as conditional expression



Other Special Operators

Special Operators

- Special operators are known as separators. They are
  - Ampersand (&)
  - Braces ({ })
  - Colon (:)
  - Ellipsis (…)
  - Asterisk (*)
  - Brackets ([])
  - Comma (,)
  - Hash (#)
  - Parenthesis (())
  - Semicolon (;)

Special Operators

Ampersand (&)

- It is also known as address operator. It is used before the variable name. It indicates memory location of the variable.
- It is denoted by '&'.
- Using '&' prefix to the variable name it gives the address of that variable.

Example:

```
#include<stdio.h>
void main()
{
    int n=10;
    printf("\n value of n is : %d",n);
    printf("\n value of &n is : %u", &n);
}
```

Output

value of n is : 10

Value of &n is:1002

Special Operators

Asterisk (*)

Asterisk (*) is also known as indirection operator. It is used before identifier name. It indicates creation of pointer variable.

Braces ({ })

The opening brace ({) and closing brace (}) specify the start and end of compound statement in a function.

Brackets

Brackets [] also referred as array subscript operator. It is used to indicate single and multi dimensional arrays.

Eg: int x[10]; float l[10][20];

Colon (:)

Colon (:) is used in labels. It is used in unconditional control statement i.e., in goto statement.

Special Operators

Comma Operator (,)

It is used to link expressions together. It is used together with variables to separate one variable from another. It is used in for loop. It has the lowest precedence among operators

Eg:

```
for(n=1,m=10;n<=m; n++, m++)
int a,b,c;
sum= (x=5,y=3,x+y);
```

Ellipsis (…)

Ellipsis (…) are three continuous dots with no white spaces between them. It is used in function prototype. It indicates that the function can have any number of arguments.

Eg:

```
void fun(char s,int n, float f, …);
```

Hash (#)

Hash **(#)** is also known as pound sign. It is used to indicate preprocessor directives.

Eg:

```
#include<stdio.h>
```

Special Operators

Parenthesis (())

Parenthesis **(())** is also known as function call operator. It is used to indicate the open and end of function prototypes, function call, function parameters. Parentheses are used to group expressions.

Semicolon (;)

Semicolon **(;)** is a statement delimiter. It is used to end a C statement.

Eg: g=d+h;

Unary Operators

- Unary operators act on single operand to produce new value. It precedes with operands.

Operator	Meaning
-	Unary minus
+	Unary Plus
++	Increment by 1
--	Decrement by 1
Sizeof	Return the size of operand

Eg:

- -786 -0.64 -(a+b)

- When operator is used before variable then it is prefix notation. When operator is used after variable then it is postfix notation.

Unary Operators

- **Eg: x=3**

Expression	Result
++x	4
x++	3
--x	2
x--	3

- **Size-of Operator**
  - This operator is used to return the size of string or character. It cannot be used in together with Integers. The syntax is
  - sizeof(variable-name);

```
#include<stdio.h>
void main()
{
    int x;
    printf("The value of x is %d",sizeof(x));
}
```

Output:

The value of x is 2

Precedence of Operators

- Outermost Parenthesis is evaluated first.
- Then innermost parenthesis.
- If there is two or more parenthesis, then the order of execution is from left to right.
- Next Multiplication and Division are performed.
- Finally Addition and Subtraction.

Expressions

- An **Expression** in C is made up of one or more operands. The simplest form of an expression consists of a single operand.
- For example 3 is an expression that consists of a single operand.
- In general a meaningful expression consists of one or more operands.
- For example a=2+3; which involves three operands, namely a, 2, 3 and two operators (=, +).
- **Operands:** An operand specifies an entity on which operation is to be performed.
  - Example a=printf("Hello")+2; is a valid expression involving three operands, namely a variable (a), a function call (printf()), and a constant (2).

Cont..

- **Operators:** An **operator** specifies the operation to be applied on its operands.
- The above example involves three operators, namely function call operator (), addition + and assignment operator =.
- Based on the number of operators present in the expression, expression are classified into two types:
  - Simple Expression and
  - Compound Expression.
- **Simple Expression:** An expression that has only one operator is called Simple Expression.
  - **Example:** a=a+2;

Cont..

- **Compound Expression:** An expression that involves more than one operator is called Compound expression.
  - **Example:** b=2+3*5;
- While evaluating compound expression we must determine the order in which operators will operate.
- For example, to determine the result of evaluation of the expression b=2+3*5, we must determine the order in which =, + and * or *, + and = will operate in these cases the result may be different.
- To avoid this situation the operators will operate depends upon the **Precedence** and the **Associativity** of the operators.

Precedence and Associativity of operators

Precedence of operators:

- Each operator in C has precedence associated with it. In compound expression, if the operators involved are of different precedence, the highest precedence operator will operate first.
- For example, b=2+3*5

Associativity of Operators:

- In a compound expression, when several operators of the same precedence appear together, the operators are evaluated according to their Associativity. An operator can be either left – to – right associative or right – left – associative.
- The operators with the same precedence always have the same associativity. Let us look at various operators, their classification, precedence and associativity.

Example

```
#include <stdio.h>
void main()
{
    int a=2,b=3,c=2;
    printf(" a * b / c = %d\n“,a*b/c);
    printf(" a + b - c = %d\n",a+b-c);
    printf(" a + b * c = %d\n",a+b*c);
}
```

Expression:

a + b * d – c/a

= a + (b * d) – (c/a)

= a + (3 * 5) – (c/a)

= a + 15 – (4/2)

= 2 + 15 – 2

= 17 – 2

= 15

C Operator Precedence and Associativity Table

Operator	Description	Category	Precedence	Associativity
( ) [ ] . ->	Parentheses (function call) Brackets (array subscript) Member selection via object name Member selection via pointer		Level -I (Highest)	left-to-right
++ -- + - ! ~ (type) * & sizeof	Increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of type) Deference Address (of operand) Determine size in bytes on this implementation	Unary Operators	Level – II	right-to-left

Cont..

Operator	Description	Category	Precedence	Associativity
* / %	Multiplication/division/modulus	Multiplicative operators	Level - III	left-to-right
+ -	Addition/subtraction	Additive operators	Level – IV	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	Shift Operators	Level – V	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	Relational operators	Level – VI	left-to-right
== !=	Relational is equal to/is not equal to	Equality operators	Level – VII	left-to-right
&	Bitwise AND	Bitwise Operator	Level – VIII	left-to-right
^	Bitwise exclusive OR	Bitwise Operator	Level – IX	left-to-right
|	Bitwise inclusive OR	Bitwise Operator	Level – X	left-to-right
&&	Logical AND	Logical Operator	Level – XI	left-to-right
||	Logical OR	Logical Operator	Level – XII	left-to-right

Cont..

Operator	Description	Category	Precedence	Associativity
? :	Ternary conditional	Conditional Operator	Level – XIII	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	Assignment and Shorthand assignment operators	Level – XIV	right-to-left
,	Comma (separate expressions)	Comma operator	Level – XV	left-to-right

Note 1: Parentheses are also used to group sub-expressions to force a different precedence;

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.


Thank you!