Basic Refresher

Number System

- In the mathematics there are mainly 4 type of number system
- Decimal (Base 10, 0 to 9)
- Binary (Base 2, 0 and 1)
- Octal (Base 8, 0 to 7)
- Hexadecimal (Base 16, 0 to 16)

What is the purpose of learning here? When we type any letter or word, the computer translates them into numbers since machines can only understand the binary numbers. To understand this conversion better we will explore the number system here.

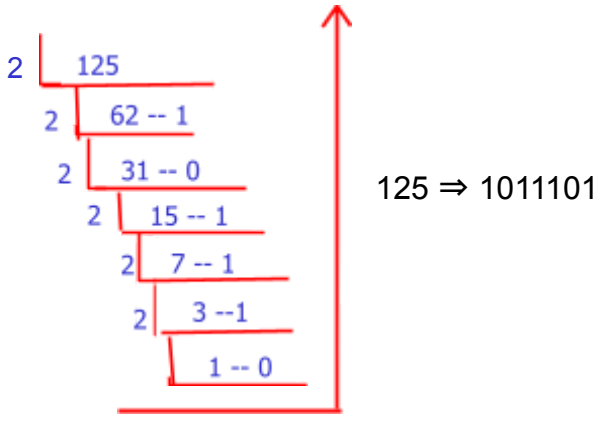
Decimal to Binary Conversion

- Whenever a decimal number has to be converted to binary we will always use the LCM method by dividing the number by 2.

E.g., number = 125, then binary will be

Decimal to binary

125 to binary



125 ⇒ 1011101

Binary to decimal

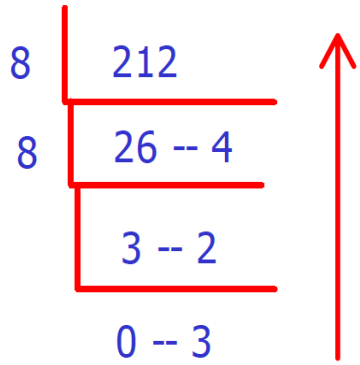
==> 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 + 1 * 2^4 + 1*2^5 + 1*2^6 + 0 *2^7

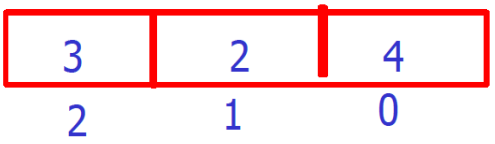
==> 1 + 0 + 4 + 8 + 16 + 32 + 64 + 0

==> 125

Decimal to octal

212 to octal ===> 0324

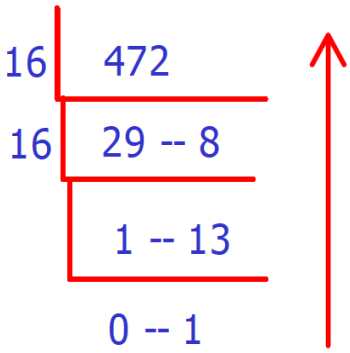




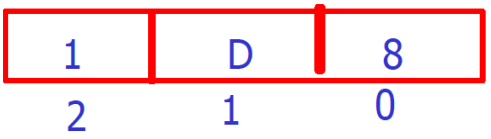
==> 3*8^2 + 2*8^1 + 4*8^0
==> 192 + 16 + 4
==> 212

Decimal to hexadecimal

472 to hexadecimal ==> 1D8



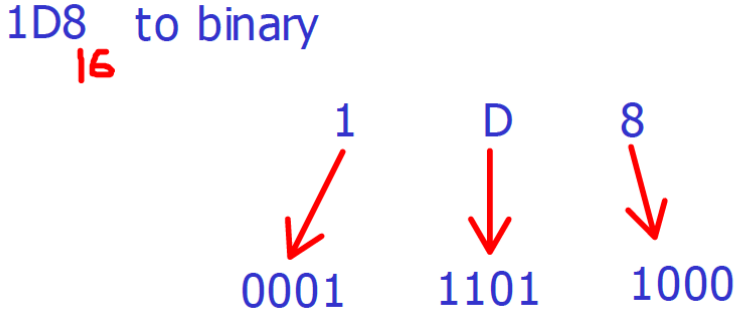
==>



1 * 16^2 + D*16^1 + 8 * 16^0
256 + 208 + 8
472

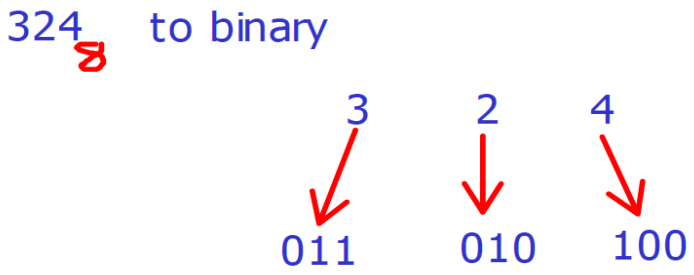
Hexadecimal to binary :

- The maximum value of hexa is F which is 15, so to represent 15 4 bits are required.
- Therefore whenever hexa is to be converted to binary then each digit should be converted to 4bit binary value like below:



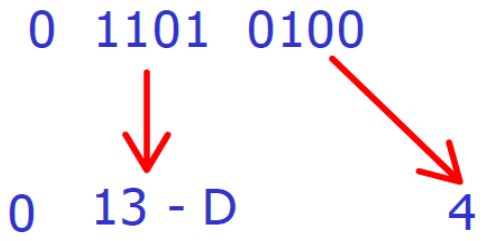
Octal to binary :

- The maximum value of octal is 7, so to represent 7, 3 bits are required.
- Therefore whenever octal is to be converted to binary then each digit should be converted to 3bit binary value like below:



Octal to hexadecimal:

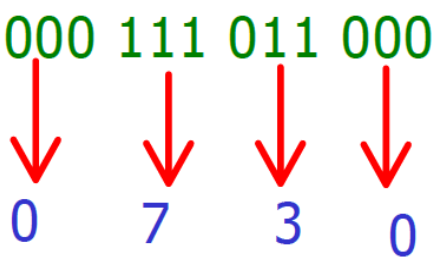
- To convert octal number to hexadecimal
  - Convert octal number to binary
  - group binary number into 4 bits each
  - take the equivalent 4-bit value of hexa
  - E.g., 324 octal to hexa
  - 324 in binary is 011 010 100
  - Group above binary to 4 bits then it will be:



- S0, 324 octal == D4 in hexa

Hexadecimal to octal:

e.g, 1D8 in octal



Data Representation

- In a computer system different data is represented in a specific way. Here, we will try to explore different ways of representing data in computer
- First, is Bit representation, it's the only language which is understood by the machine.
    - Bit is derived from the word binary digit.
    - It has 2 possible states: true or false or 0 or 1.
- Byte is a collection of 8-bit.
    - its a smallest unit of information in the computer memory.
- Character is a 1byte integer which is represented differently in different languages with the help of character code like, ASCII, EBCD, UTF etc.
    - In C, character is represented with the ASCII code.
    - Characters are represented with ‘ ‘ and a character will be 1byte information
    - For example if you use any character internally these are represented as follows:

      '0' → 48(ASCII code) → 0011 0000

      'A' → 65 → 0100 0001
- Next, word is the capacity of a processor, which defines how much information it can fetch or process at a time.
    - For example, 8-bit MC → MC can fetch 8-bit of information at one time/one cycle 32 --> fetching 32-bits of information at a time.
- Integer numbers are represented as a direct binary conversion
    - for example, positive 13 will be in 32 bits

      0000 0000 0000 0000 0000 0000 0000 1101
    - Negative number -13 will be stored as 2s complement

      1's complement + 1

      0000 0000 0000 0000 0000 0000 0000 1101

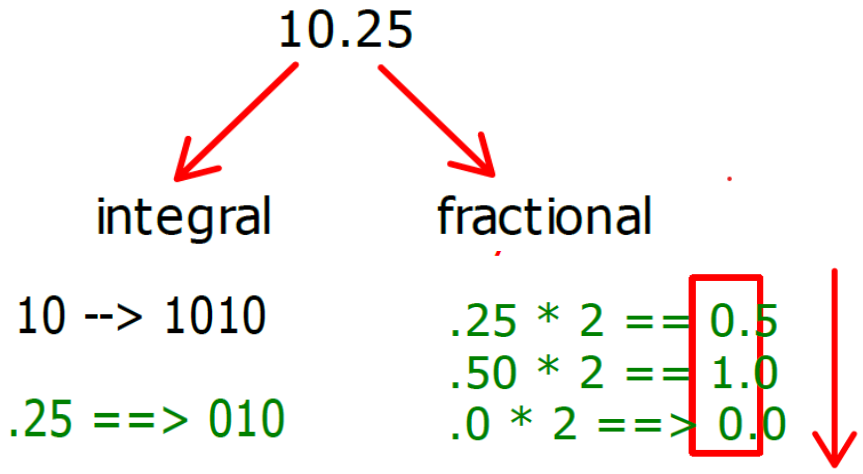
      1111 1111 1111 1111 1111 1111 1111 0010
      +0000 0000000 0000 0000 00000000 0001
      1111 1111 1111 1111 1111 1111 1111 0011
    - In other word -13 in 2’s complement can be in 8-bit system

      ==> 256 -13 ==> 255

Float Representation

- Real values in C are represented as a IEEE 754 standard format which follows following steps

step 1: Convert fractional and integral value to binary



SO, 10.25 ⇒ 1010.010

Step 2 : Convert the result of step1 to standard exponent formula

$1.Xe^{+/-y}$

1010.010
101.0010
10.10010
1.010010

y --> number of shift you will do to get the format

y=3

if shift is towards left then value is +ve

or if towards right side value is -ve

X --> mantissa --> 01001000000000000000000 (23 bits in float)

sign bit - 0

step 3: Find the exponent part

--> add or subtract y with the standard bias number
--> for floating point standard bias number is --> 127
--> if y is positive add else if negative subtract

exponent = 127 + 3 == 130 ⇒

0	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- If negative only is given then only sign bit will be 1 other rest of the steps are same as previous
- If the value is double then step 1 and 2 remain the same. In step 3 the standard bias number will be 1023 for double.

2. 1.7 --> never ending number

==> 1

1.10110011001100110.....

.7 * 2 = 1.4
.4 * 2 = 0.8
.8 * 2 = 1.6
.6 * 2 = 1.2
.2 * 2 = 0.4
.4 * 2 = 0.8
A.8 * 2 = 1.6
.6 * 2 = 1.2
.2 * 2 = 0.4

step 2: no need to convert so
--> y=0

X==> 1 0110 0110 0110 0110 0110 01
sign = 0

step 3: exponent

127 + 0 == 127==> 0111 1111

0	0	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1

For double

--> All the steps are same only change is representation of exponent and mantissa and standard bias number is 1023

--> exponent --> 11
--> mantissa --> 52 1. 2.5

10.10

2. y = 1

1.010e

X - 01000000000000000000000000000000000000000000000000

sign = 0

3. exponent -->
1023 ==> 1023 + 1 ==> 1024 ⇒ 1000000000

0 01000000000 01000000000000000000000000000000000000

Data types

- Data Type specifies which type of value a variable has and how much memory can be allocated to each.

- In C data types are divided into primitive/basic and non primitive(user defined) data types.

- In basic data types following are the division:
    - Integral
        - int : integer type and memory allocation is compiler implementation dependent. If turboc then sizeof(int) is 2 bytes, If gcc then the sizeof(int) is 4 bytes.
        - char: Character type is a 1byte integer data. All the characters in C are converted to ASCII equivalent code.
    - Real
        - Float: 4bytes real value which is a single precision data type
        - Double: 4byte real value which is a double precision datatype.

- Size of datatype can be obtained with the help of sizeof operator.
    - sizeof(variable_name);
    - sizeof(data type);
    - sizeof(constant);
    - sizeof datatype;
    - sizeof returns the value in unsigned int(%u) if 32 bit system else unsigned long int(%lu) in 64 bit system.
    - The format specifier is %zu which is a portable one.

Modifiers and Qualifiers

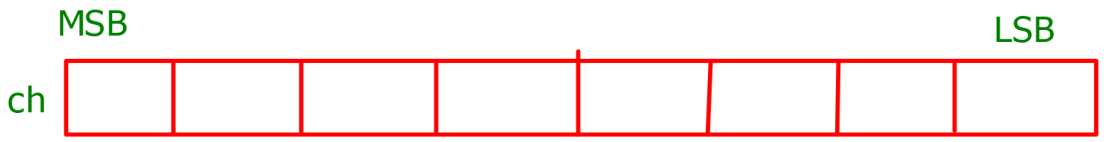
- Modifiers and qualifiers are the special keywords which will modify the width, accessibility and memory scope of the variables.
- Modifiers are divided as follows:
    - size modifier: short, long, long long

short	long	long long

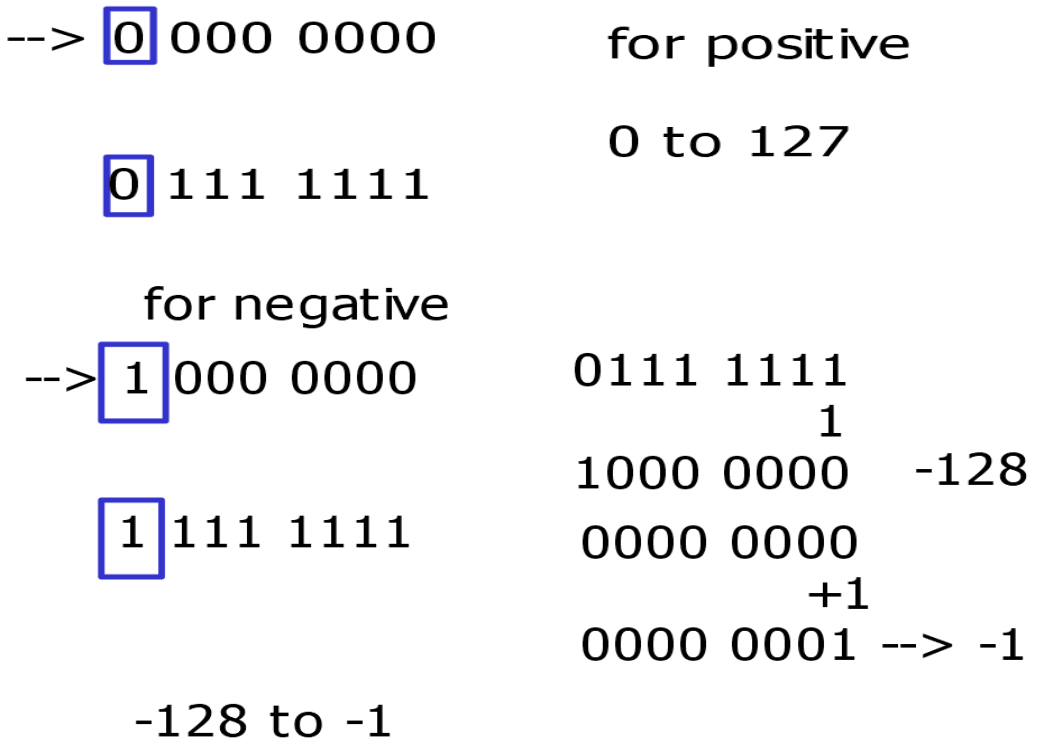
● short modifiers can be used with only int datatype
● It will modify the width of int to 2bytes in gcc.
● short int num;
short num;
● format specifiers:
int - %hd
hexa - %hx/%hX
octal - %ho
	● long modifiers can be used either with int or double data type
	● It will modify the width of int and double which is compiler implementation dependant -
	for int 4/8, for double - 12/16.
	● long int num;
	long num;//int long double num;
	● format specifiers:
	int - %ld
	hexa - %lx/%lX
	octal - %lo
	double - %Lf
		● long long modifiers can be used with only int datatype
		● It will modify the width of int depending on the compiler. Sizeof long long will be 8byte/16 bytes
		● long long int num;
		long long num;
		● format specifiers:
		int - %lld
		hexa - %llx/%llX
		octal - %llo

    - sign modifiers: signed and unsigned
        - signed is a default modifier for the integral data type. Signed variables are capable of holding both positive and negative values.
        - In signed, MSB is dedicated to the sign bit.
        - So, the range of the signed variable can be calculated as,
            - $-2^{(n-1)}$ to $+2^{(n-1)}-1$
            - if character then the range is: $-2^{(8-1)}$ to $+ 2^{(8-1)}-1 = -128$ to $+127$
            - The values outside the specified range then it will result in overflow and underflow(discussed later)

example: signed char ch;



- Here, MSB(Most Significant Bit is dedicated for sign bit), if 0 its a positive number, if 1 then negative number,
- So,



- The range will be, -128 to +127\
- Example,
  - signed char ch = 123; //binary is: 0111 1011
  - char ch = 133;
    - Here, binary of 133 = 1000 0101, MSB bit is 1
    - So, compiler will take the 2s complement of the number

      0111 1010

      0000 0001

      0111 1011 → -123

Unsigned Variables

- Only positive values are stored in unsigned types.
- No special dedication for MSB bit.
- So, range = 0 to 2^n - 1, so for unsigned char range = 0 to 255
- Example,

  unsigned char ch = -13;

  - Here, first negative value will be converted to 2s complement
  - 0000 1101

    1111 0010

    00000001

    1111 0011
- When u try to print the value then the output will be positive equivalent of 1111 0011 which is, 243

- Now, variables can be declared with both the signedness and size modifiers like,
  - unsigned long int;
  - long long int; // by default its signed
  - short unsigned int;

Statements

Conditional Constructs

- In C, conditional constructs are used to construct the statements depending on the condition whether it's true or false. Its divided into 2 groups:
    - Single iterative : Here, statements will be executed only else depending on the condition.
    - Multiple iterative: Here, statements are executed repeatedly until the certain condition is true. Once the condition becomes false it will stop the loop.

Single Iteration:

- If and its family: If is a single iterative conditional construct which will execute the statements only once.

  syntax:

  ```
  if(expression)
  {
     statements;
  }
  ```

- Examples:

  ```
  int num = 10;
  if(num == 10)
  {
          printf(“num is equal to 10\n”);\
  }
  ```

```
if (num < 5);
  printf("If: num < 5\n");
```

```
if (num < 5)
{
    ;//dummy code
}
 printf("If: num < 5\n")
```

```
if (num < 5);
   {
      printf("num < 5\n");
   }
   else
   {
       printf("num == 5");
   }
```

```
if(num<5)
{
  ;
}
{
     printf("num < 5\n");
}
else
   {
      printf("num == 5");
   }
```

switch conditional construct

- Switch is also a single iterative condition construct which will execute the statements once which is matching with the case label.

  syntax:

  ```
  switch(expression)
  {
     case label:
          statements;
          break;
    case label:
          statements;
          break;
   default:
          statements
  }
  ```

- Where, case label has to be integral constant or enum.
- Real constants are not allowed --> 2.5,4.5,7.8.....
- Case label should be unique
- Multiple values are not allowed in case label
- A break statement is used to break the switch cases.
- If break is not used then switch will result in fall through condition like
  - switch(expression)

    ```
    {
       case 10:
            printf("10 is selected\n");
       case 20:
            printf("20 is selected\n");
       default:
            printf("None is matching\n");
    }
    ```

    - In the above example if the expression is having 10, then all the 3 printf is executed.

switch vs if else

- consider the below scenario where x = 10, then execution time for if is

```
if (x == 100)
{
}
else if(x == 90)
{
}
else if(x == 80)
{
}
else
{
}
```

1 cycle

2 cycle

3 cycle 4th cycle

200nsec * 4 == 800nsec

For switch JUMP default:

```
case 100:
.
.
case 90:
.
.
case 80:
.
```

.
default:
1cycle == 200nsec

Multi Iteration Conditional construct:

- While loop is known as entry controlled loop where statements are executed repeatedly until the condition is true.
- syntax:

```
while(expression)
{
  statement;
}
```

- Example:

```
int iter = 0;
while(iter < 5)
{
   printf("Looped %d times\n",iter);
   iter++;
}
```

- In the above loop, iter is 0 so the condition is true it will enter the loop.
  - prints, looped 0 times then increments the iter = 1
  - continues till iter = 5
  - Once the condition is false comes out of the loop.

1. whileÀàint main()
{
int iter;

iter = 0;
while (iter < 5)
printf("Looped %d times\n", iter);
iter++;

return 0;
}

Interpretation:

```
int main()
 {
 int iter;

 iter = 0;
 while (iter < 5)
 {
 printf("Looped %d times\n", iter);
 }
 iter++;

 return 0;
 }
```

Output: looped 0 times

2.

```
int main()
{
        int iter;
        iter = 0;
        while (iter < 5);
        {
                printf("Looped %d times\n", iter);
                ter++;
        }
        printf("iter: %d\n", iter);

        return 0;
}
```

```
int main()
{
        int iter;
        iter = 0;
        while (iter < 5)
        {
            ;
         }
        {
                printf("Looped %d times\n", iter)
                ter++;
        }
        printf("iter: %d\n", iter);

        return 0;
}
```

```
int main()
{
int iter; iter = 6;
while (iter < 5);
{
printf("Looped %d times\n", iter); iter++;
}
printf("iter: %d\n", iter);

return 0;
}
```

compiler view

```
int main()
{
int iter; iter = 6;
while (iter < 5)
{
;
}
{
printf("Looped %d times\n", iter); iter++;

}
printf("iter: %d\n", iter); return 0;
```

do…while loop

```
iter = 0;
do
{
printf("Looped %d times\n", iter);
iter++;
} while (iter < 5);
```

1. Looped 0 times
   iter = 1
   1 < 5
2. looped 1 times
   iter = 2
   2 < 5
3. looped 2 times
   iter = 3
   3 < 5
4. looped 3 times
   iter = 4
   4 < 5
5. looped 4 times
   iter = 5
   5 < 5 --> exit the loop

While vs do..while

While	do..while
--> entry controlled loop if the condition is true then enters loop	--> exit controlled loop enter the loop then check for the condition
--> if condition is true then only statements will be executed	--> atleast for once the statements gets executed
where and when? number of times/iteration is unknown	--> Menu-driven application ATM --> do you want to continue - y/n Gaming -->

For loop:

- For loop is also like while loop, which will execute the statements repeatedly until the condition is true.
- For loop syntax:

```
for(initialisation; condition ; post evaluation)
{
    statements;
}
```

- For loop will be executed as stated below:
  1. Initialization
  2. Condition - if true goto step 3 else goto step 6
  3. Statements
  4. Post evaluation expression
  5. Goto step 2
  6. Exit

1.

```
int main()
{

int iter = 0;

for (iter = 0 ; iter < 10; iter++ );
{
printf("Looped %d times\n", iter);
}
return 0;
}
```

```
int main()
{
 int iter = 0;
for (iter = 0 ; iter < 10; iter++ )
{
;
}
{
printf("Looped %d times\n", iter);
}
return 0;
}
```

2.

```
while(iter++ < 5)
{
printf("Looped %d times\n", iter);
}
```

Post increment -->
assign the value/compare the value
next time use the incremented value

1. 0++ < 5 ==> 0 < 5
   (iter = iter+1)
   Looped 1 times

2. 1++ < 5 ==> 1 < 5
   (iter = iter+1)
   Looped 2 times

3. 2++ < 5 ==> 2 < 5
   iter = iter+1
   looped 3 times

4. 3++ < 5 => 3 < 5
   iter = iter+1
   Looped 4 times

5. 4++ < 5 => 4< 5
   iter=iter+1
   Looped 5 times

6. 5++ < 5 => 5 < 5 --> false
   iter = iter+1 = 6

3.

```
while(++iter < 5)
{
printf("Looped %d times\n", iter);
}
```

Pre increment==>
first increment the value and then assign/ compare

1. ++0 < 5 ==> 1 < 5
   Looped 1 times
2. ++1 < 5 ==> 2 < 5
   looped 2 times
3. ++2 < 5 ==> 3 < 5
   looped 3 times
4. ++3 < 5 ==> 4 < 5
   looped 4 times
5. ++4 < 5 ==> 5 < 5
   exit the loop

Nested for loop:

```
Nested for loop

for(i = 0; i < 3; i++)
{
   for(j = 0 ; j < 2 ; j++)
   {
        printf("Hello\n");
    }
}
```

	col 0	col 1
row 0	hello	hello
row 1	hello	hello
row 2	hello	hello

1. i = 0
   0 < 3
   1.1 j = 0
   0 < 2
   Hello
   1.2 j = 1
   1 < 2
   Hello
   1.3 j=2
   2 < 2 --> false
   --> exit inner loop

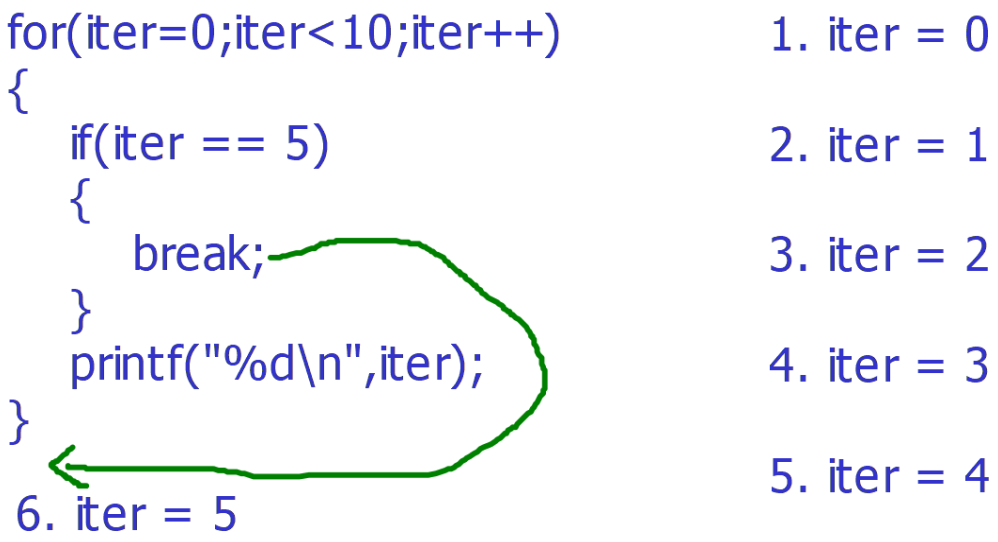
2. i = 1
   1 < 3
   2.1 j = 0
   0 < 2
   Hello
   2.2 j = 1
   1 < 2
   Hello
   2.3 j=2
   2 < 2 --> false
   --> exit inner loop

3. i = 2
   2 < 3
   2.1 j = 0
   0 < 2
   Hello
   2.2 j = 1
   1 < 2
   Hello
   2.3 j=2
   2 < 2 --> false
   --> exit inner loop

4. i = 3
   3 < 3 --> false --> exit the loop

Break:

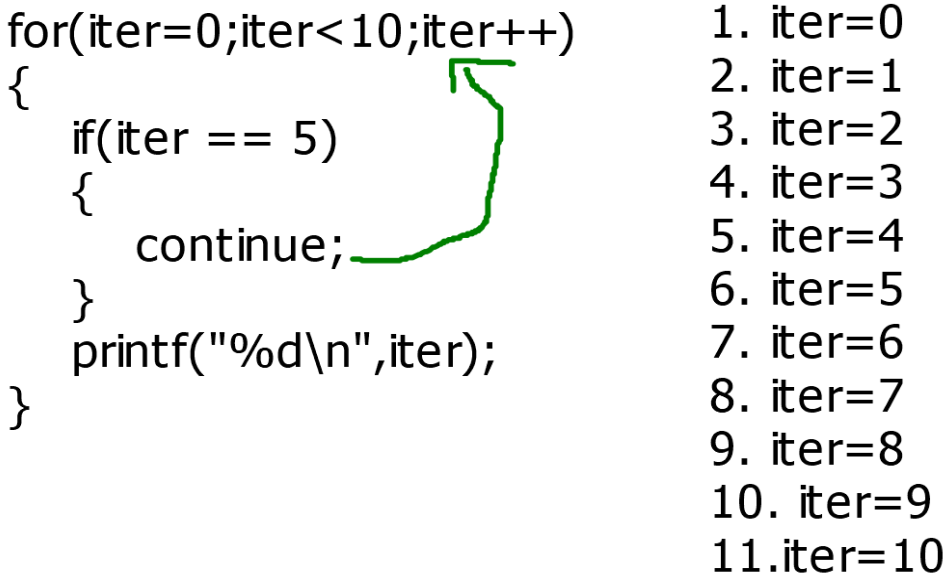
- used to exit from the loop or switch case depending on the condition
- Break should be within the loop



continue statement

--> only used in multi iterative conditional construct --> do..while,while,for

--> used to skip that specific iteration of the loop



Goto Statement--> unconditional jump

```
int iter = 5;

goto lab1;
printf("Hello\n");

lab1:
printf("World\n");
```

--> generally it is avoided due to difficulty in tracing the flow

Which one is Efficient?

```
for(i = 0 ; i < 10; i ++)
  for(j = 0 ; j < 100 ;j++)
```

1.
outer loop:
initialization:1
post evaluation expression: 10
condition: 11
total 22 machine cycle

2. inner loop: for single
initialisation: 1
condition: 101
post eval expre:100

total 202 * 10 == 2020

22+2020 ==> 2042 Machine cycle

```
for(i = 0 ; i < 100; i ++)
  for(j = 0 ; j < 10 ;j++)
```

1.
Outer Loop
1 + 101 + 100 == 202

inner loop

1+11+10 == 22 * 100 == 2200

total ==> 202 + 2200
==> 2402 machine cycle

1st one is effecient

Operators

- Operator is a special symbol which will perform a specific task on the operand and returns a value.
- Operators are divided on operands as below:
  - Unary : One operand - ++, - -, +, - etc
  - Binary: arithmetic, logical, bitwise etc.
  - Ternary: ?:

Unary Operators

--> used with one operand

--> highest precedence in the table

```
int x, y=10;
x=y;

x = +y;
```

```
x = -y;

x = -(y);
x = -10
```

```
int x, y=-10;

x = -y; // x = -(-10)
x = 10
```

```
int x, y=-10;

x = + - y; //+ - (-10)

x = +10
```

Increment and Decrement Operator

1. Pre increment

   1 --> increment first
   2 --> assign the value

```
y = ++x;
```

```
int y, x = 10
```

1. x = x+1
   x = 10+1
   x=11

2. Incremented value will be assigned to y
   y = 11

2. Post increment

1 --> assign first/use the original value first
2 --> increment the value of variable

y=x++;

1. x value will be assigned to y, i.e, 10 will be stored in y
   y = 10

2. x = x+1
   x = 10+1
   x = 11

3. Pre decrement

--> decrease value by 1
x-- ==> 9
--x ==> 8

x=5

y = --x; 4 4

4. Post decrement

y = x--; y = 5 x = 4

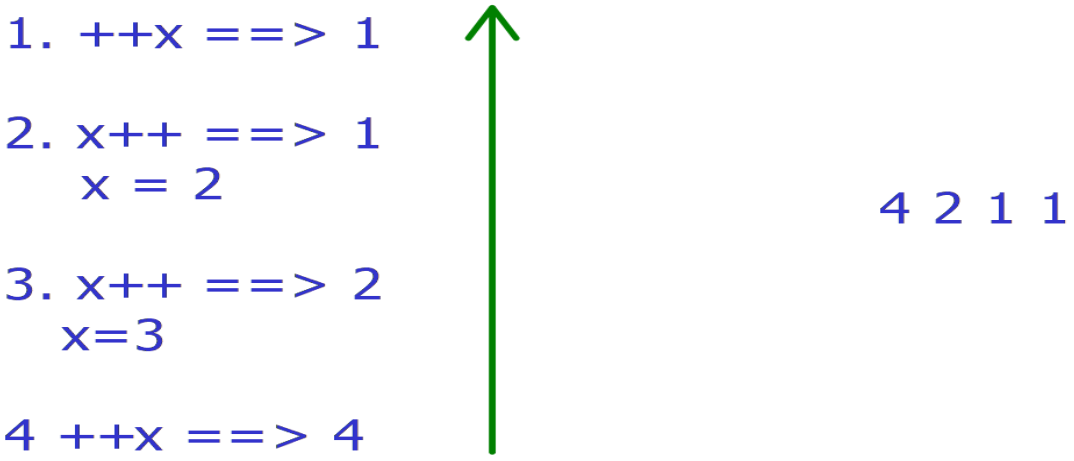
5. y = y++;

- These expression will have undefined behaviors because variable y is incrementing assigning at the same time
- So, output cannot be predicted.

6. x =0;

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

- This also results in undefined behavior



- Because of post increment and pre increment in the same line
- I am supposed to get 4 2 11 but output will be different

sizeof() --> function or operator?

--> for sizeof u can pass any type of data

--> can also pass data type - char int float

--> sizeof(), sizeof var

Type Conversion

- **Type conversion is a process of converting one type to another whenever required.**

Type conversion

1. int x=10, y=4, z;

z = x / y;

z = 2

==>

10/4

int = int / int ==> integer result ==> 2

2. int x = 10, y = 4;

float z;

z = x / y;

==>

x / y ==> int / int = 10 / 4 = 2

z = 2.000000

3. int x= 10;

float y = 4, z;

z = x / y;

==>

x / y ==> int / float

==> float / float

10 / 4 ==> 10.000000 / 4.000000

==> implicit type conversion

==> int is the lower rank type, so it will be converted to higher rank type float

```
int x = 10, y=4;
float z;

z = (float) x / y;       // explicit type conversion
```

float / int ==> float / float ==> 10.000000 / 4.000000 ==> 2.500000

Unary Conversion

--> if char and short both will be converted to integer

```
char x = 12, y=20, z;

z = x + y;
```

==> x + y ==> char + char ==> int + int

```
char x = 12;
short int y = 20, z;

z = x + y;
```

==> short + char ==> int + int

Logical Operator – AND, OR, NOT

1. AND operator --> &&

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

0 --> false
1 --> True

==> For AND if any 1 input is false then output will be false
if both input is true then only output is true

2. OR Operator --> ||

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

if any one input is true then output will be true

3. NOT Operator --> !

A	Output
0	1
1	0

Examples:

1. int a = 10,b=20;
   a && b ==> 10 && 20 ==> true && true ==> true ==> 1 1

   All the values are true except 0, NULL

Circuit Logic:

- Circuit logic is the way of evaluation used by compiler to optimize code
- For example if OR is used, num1 = 1, num2 = 0, then

  ```
  if (++num1 || num2++)
  {
      //statement
  }
  ```

  Here, ++num1 ⇒ ++1 is true value so second is not evaluated
- Similarly if(num2++ && num1++)
  Here, the first expression is false so the second is not evaluated.

Examples:

1.

```
a = 50
b = -50
c = 0
d = 10
```

```
1
```

```
a || b && c && d

a || (b && c) && d

a || ((b &&c) && d)

50 || ((-50 && 0) && 10)
```

2.

```
a && b && c || d

==> (a && b) && c || d
==> ((a && b) && c) || d
==> ((50 && -50) && 0) || 10
==> ((1) && 0) || 10
==> 0 || 1
==> 1
```

```
a && b && c && d

0
```

Compound Statements:

num1 += num2 += num3 += num4;

```
1. num3 += num4;
   num3 = num3 + num4;
        = 1.7 + 1.5
        = 3.2

2. num2 += num3

   num2 = num2 + num3;
        = 1 + 3.2
        = 4.2
   num2 = 4.2
   num2 = 4

3. num1 += num2
   num1 = num1 + num2
        = 1 + 4
        = 5
```

Bitwise Operators

- Bitwise operators work on bits of the given values.
- Bitwise operators are used whenever a particular bit/s has to be set, clear or toggle.

1. Bitwise AND - &

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

2. Bitwose OR -- |

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

3. Bitwise XOR -- ^

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Examples:

1. char x = 0x61 y = 0x31

   x & y

   x = 0110 0001

   y = 0011 0001

   0010 0001 ==> 0x21 ==> 33

2. x = 0xAA y = 0x57

   x & y

   1010 1010

   0101 0111

   0000 0010 ==> 0x02

3. int x = 10 y = 15

   0000 0000 0000 0000 0000 0000 0000 1010

   0000 0000 0000 0000 0000 0000 0000 1111

   0000 0000 0000 0000 0000 0000 0000 1010

4. OR --> char x = 0x61 y = 0x13

x | y

0110 0001

0001 0011

0111 0011 0x73 ==> 115

5. x = 0xAA y = 0x53

x | y

1010 1010
0101 0011

1111 1011 --> 0xfb --> 251

XOR --> x= 0xBC y = 0x35

x ^ y ==> 1011 1100
0011 0101
1000 1001 ==> 0x89 --> 137

Bitwise Complement --> (~)

---> 1's compliment of a given number

A	Output
0	1
1	0

1. x = 0xBC

~x ==> 1011 1100

0100 0011 ==> 0x43

Shift Operators

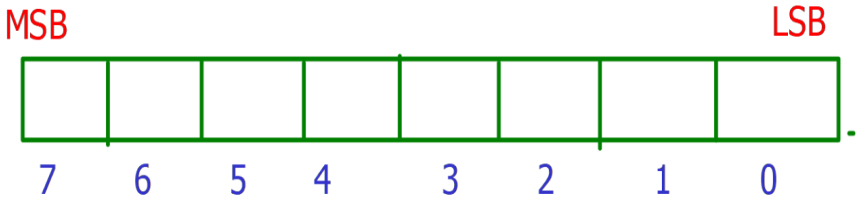
1. **Left Shift --> bits will be shifted to left side depending on the number of shift**
2. **right shift --> bits will be shifted to right side**

1. Left shift: <<

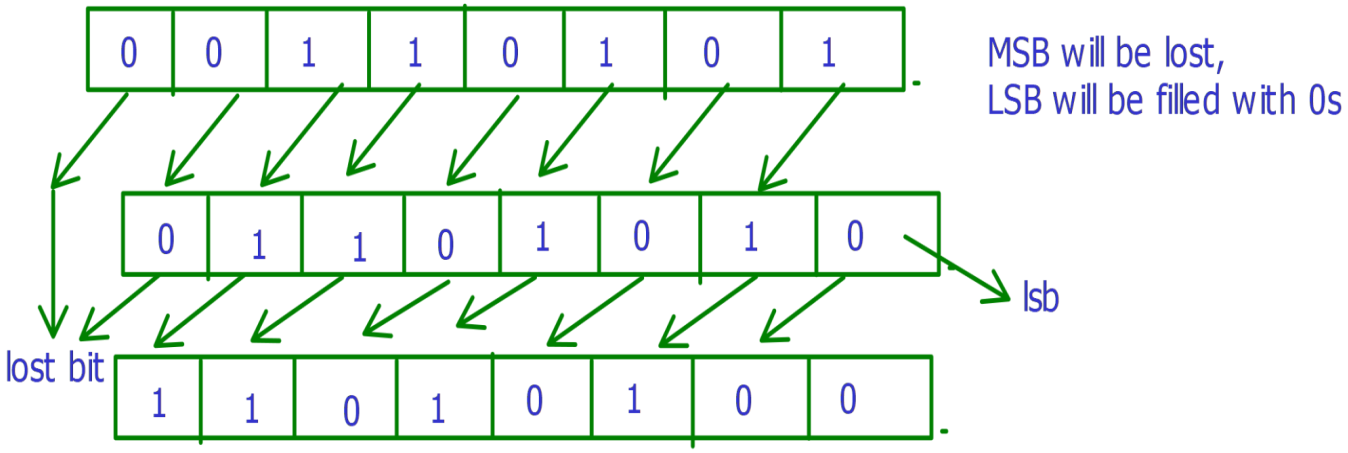
value << no_of_bits (no_of_shift)

char a = 0x35

a << 2



0001 1000
0000



==> 1101 0100 ==> 0xD4

0x35 --> 53 ==> 53 * 2^0
1. first shift --> 0110 1010 --> 0x6A --> 106 ==> 53 * 2^1
2. second shift --> 0xD4 --> 212 ==> 53 * 2^2

--> effecient way of multiplying 2 to the power values

ouptput of shift

resultant = value << no_of_bits ==> resultant * 2^no_of_bits

2.

2. 0x16 << 3 ==>

1st shift --> 22 --> 0001 0110 --> 0010 1100 --> 44 --> 2C

2nd shift --> 0101 1000 == 88 == 58

3rd shift --> 1011 0000 == 176 == B0

22 * 2^3 ==> 22 * 8 ==> 176 ==> B0

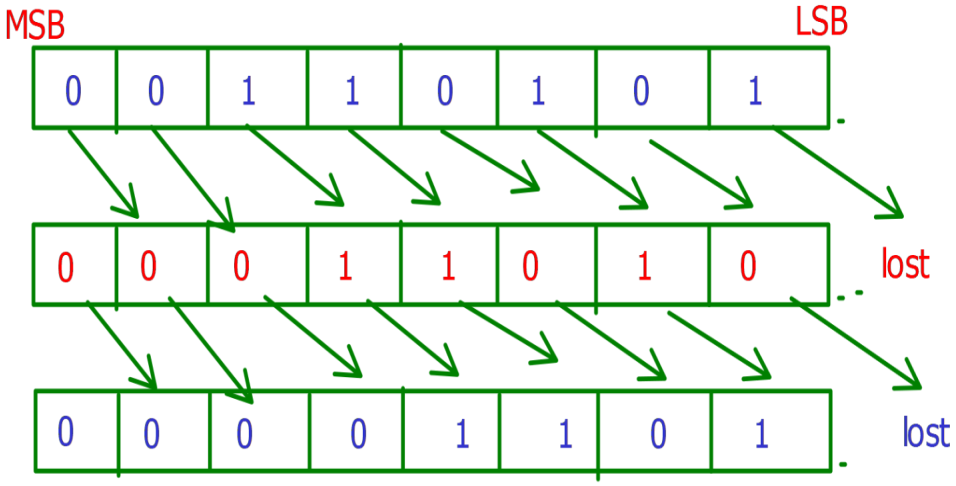
Right Shift: Unsigned right shift

unsigned char ch = 0x35;

ch >> 2

MSB will be filled with 0

0000 1101 --> 0x0D



--> unsigned right shift effecient way of dividing number by 2 power values

0x35 --> 53 ==> 53 / 2^0 ==> 53

0x1A --> 26 ==> 53 / 2^1 ==> 26

0x0D --> 13 ==> 53 / 2^2 ==> 13

value >> no_of_bits

resultant = value / 2^no_of_bits

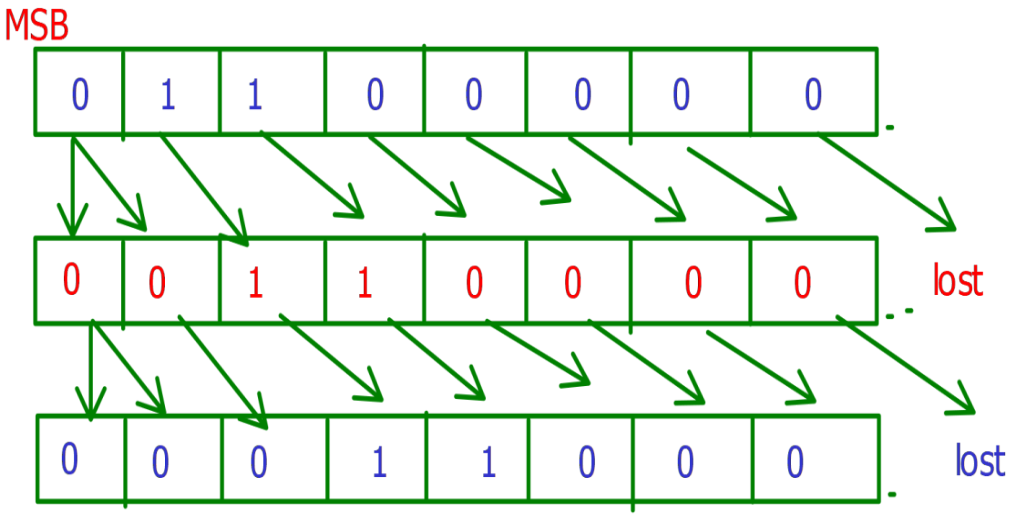
Signed right shift:

signed right shift

signed char ch = 96;

ch >> 2

previous MSB will be filled
with current shift of MSB



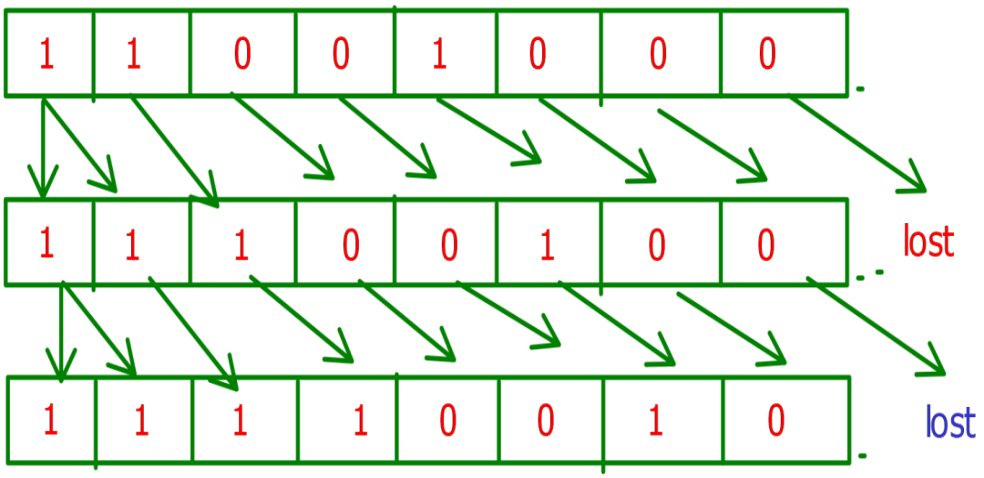
96 ==> 96 / 2^0 ==> 96
96 ==> 96 / 2^1 ==> 48
96 / 2^2 ==> 24

signed right shift - negative number

signed char ch = -56

ch >> 2

00111000
11000111
+1
11001000



1st shift --> 1110 0100 -56 / 2^1 = -28
0001 1011
+1
0001 1100 ==> -28

2nd shift --> 1111 0010 ==> -14 -56 / 4 ==> -14

Points to remember:

- ❖ Number of shift should be positive value, negative shift value will result in undefined behavior (output cannot be predicted)
- ❖ right operand (number of bits/shift) has to be within the range of left operand

  for example,

  unsigned char ch = 90;

  - so valid right operand values are 0 to 7
  - in case of integer --> 0 to 31
  - if right operand value is more than the range of bits of left operand then output will be undefined behavior

- ❖ After shifting, if the result is not within the range of datatype then undefined behavior

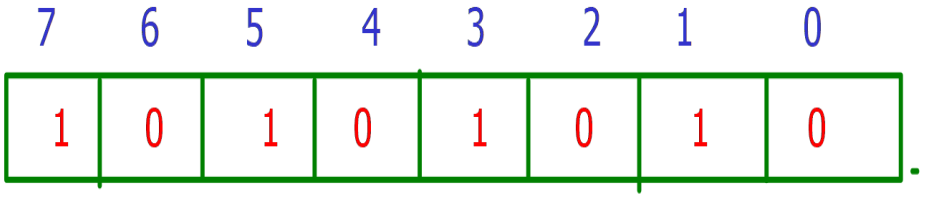
  e.g, signed char ch = 96;

  ch << 4 ==> 1536 //the value is out of range

Applications of Bitwise operators:

Set bit --> Whatever the bit might be, make it as 1 (either it can be 1 or 0)

value = 0xAA



set the bit present at the position 4 --> 1011 1010

--> chose any bitwise operator, peform the operation with mask

```
  1010 1010
| 0001 0000      0x10 ==> mask
  1011 1010
```

set the bit present in 6th position

```
1010 1010
0100 0000      0x40 ==> mask
1110 1010
```

==> whenever it is a set bit, you have to use bitwise OR operator

--> 0xAA | mask == result

--> 1 << 4

1010 1010
0001 0000

1011 1010

1011 1010

0000 0001
0000 0010
0000 0100
0000 1000
0001 0000

1 << 6

1010 1010
0100 0000
1110 1010

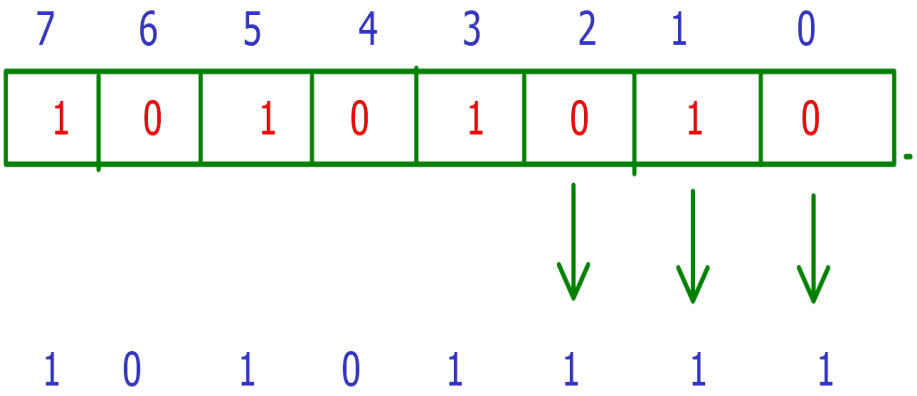
0100 0000

mask = 1 << position

result = value | mask

set 3-bits from lsb of the given value

value = 0xAA



--> 1010 1010
0000 0101
1010 1111

1010 1010
0000 0111
1010 1111

Generic mask --> only 3-bits has to unproteced

mask = (1 << no_of_bits) - 1

0000 1000 - 1 ==> 7

Clear bit --> whatever the bit may be, you have to make it as 0

value = 0xAA

1010 1010 ==> 1010 0010

clear the 3rd bit of given value

bitwise AND - &

```
  1010 1010
& 1111 0111   F7
  1010 0010
```

Generic mask

1010 1010

1111 0111

1010 0010

~(1 << pos)
~(1 << 3) ==> 0000 1000 => 1111 0111

==> whenever clear bit is asked, you should use bitwise AND operator with desired mask

clear 4 bits from LSB of the given value

```
1010 1010
1111 0000

1010 0000
```

generic mask -->

mask = ~ ((1 << no_of_bits) - 1)

~((1 << 4) - 1)

~((16) - 1)
~15

0000 1111
1111 0000

Toggle bit --> 0 to 1 and 1 to 0

0xAA ==> 10101010 1010 0010

11110111
10100010

1010 1010 ==> 1011 1010
0001 1000

==> whenever toggle bit is asked, use XOR (^)

Generic Mask

mask = 1 << pos

res = value ^ mask;

mask = 1 << 3

1010 1010

^ 0000 1000
1010 0010

Ternary Operator

- works with 3 operands → ?:
- syntax: condition ? expression_true : expression_false
- For some scenarios or requirement ternary is the optimization for if..else
- Example:

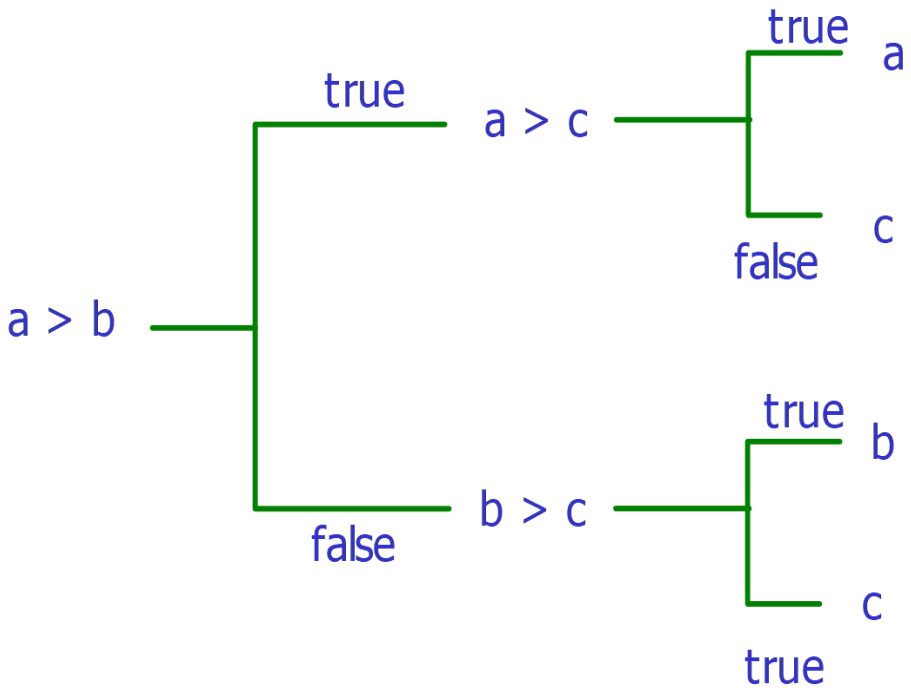
  max = num1 > num2 ? num1 : num2;

  num1 > num2 > printf("Num1 is max\n") : printf("Num2 is max\n");

  - Can also have nested ternary like,

```
if(a > b)
{
  if(a > c)
   {
     pf(a);
   }
  else
  {
     pf(c);
  }
}
else
{
 if(b > c)
 {
    pf(b)
  }
 else
  {
    pf(c);
}
```

1. `res = a > b ? a > c ? a : c : b > c ? b : c;`



Comma operator:

- Comma operator have certain usage with for loop and in some expression, or function calling
- It will evaluate the expression but consider or return the right most value to the function or expression.

```
int num1,num2,num3;

num1 = (1, 2, 3)

num1 = (x = 1 + 2, y = 2 + 2, z = 3 + 3);

x = 5, 10, 5;

x = (y=100, z = y + 10)
```

Here, num1=3

num1 = 6, x = 3, y = 4, z = 6

Examples:

j = i++ ? i++, ++j : ++j, i++;

= 0++ ? i++, ++j : ++j, i++; ==> i = 1

?: ,

(i++ ? (i++, ++j) : ++j), i++;

0++? not eval : ++j, i++; ==> i = 1, j = 1 i = 2

```
for(i = 0, j = 0;i < 5, j < 10;i++, j++)
{

}
```

1. i = 0 , j=0

   0 < 5, 0 < 10

2. i = 1, j = 1

   1 < 5, 1 < 10

3. i = 2, j = 2

4. i = 3, j = 3

6. i = 5, j = 5

   5 < 5, 5 < 10

7. i = 6, j = 6

   6 < 5, 6 < 10

8. i = 7, j = 7

9. i = 8 , j = 8

10. i = 9, j = 9

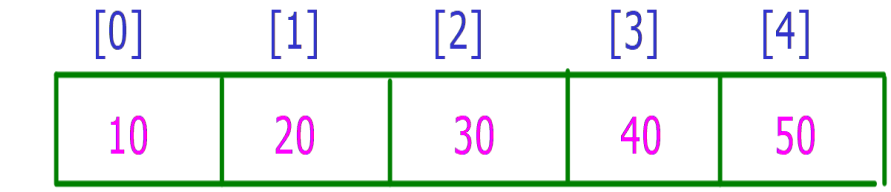
11. i = 10, j = 10

10 < 10

Arrays

- Array is a collection of the same type of data.
- huge data type which can store more than one value
- used to organize the data

syntax:
datatype array_name[SIZE]; e.g,
int arr[5]; //integer array of size 5
int arr[5] = {10,20,30,40,50};



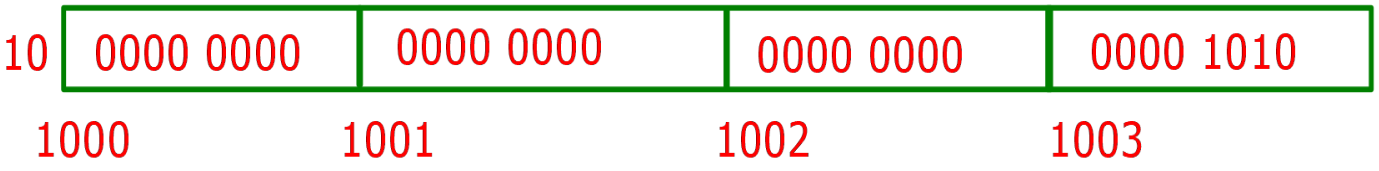
1000 1004 1008 1012 1016 1020 (assumption)

Total Memory allocated = sizeof_datatype * sizeof_array

= 4 * 5
= 20

arr[0] ==> 10
arr[1] ==> 20
arr[2] ==> 30
arr[3] ==> 40
arr[4] ==> 50

how each element is represented?



How to print an array?

- Arrays are printed with the help of loops
- Example:

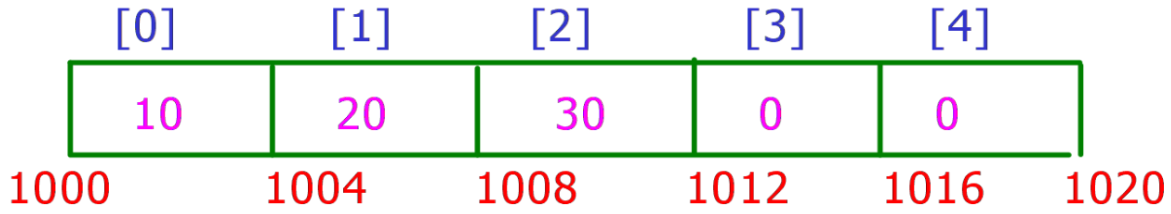
```
for(index = 0; index < 5 ; index++)
    printf("%d\n",arr[index]);
```

Different ways of declaring an array

1. int arr[5] = {10, 20, 30, 40, 50};

2. Partial initialisation

int arr[5] = {10, 20, 30};



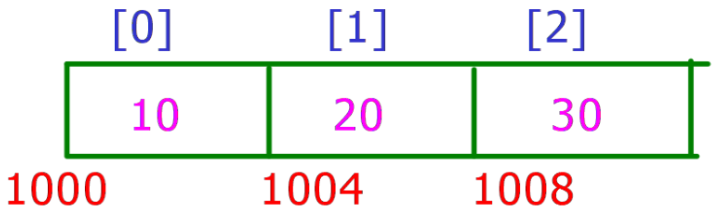
-- rest of the elements will be initialised 0

3. int arr[] = {10, 20, 30};

memory allocated ==> sizeof_datatype * number_0f_elements

4 * 3

12bytes



4. int arr[];

// error, invalid

5. int arr[] = {10, 20, ,30, 40};

// invalid, error

6. int arr[5];

--> by default array elements will have garbage values

Reading array from user:

- ```
  for(index=0;index < 5;index++)
  {
      scanf("%d",&arr[index]);
  }

  for(index=0;index < 5;index++)
  {
     printf("%d ",arr[index]);
  }
  ```
  ```