

Fundamentals of Programming

CCS1063/CSE1062

Lecture 3

Professor Noel Fernando

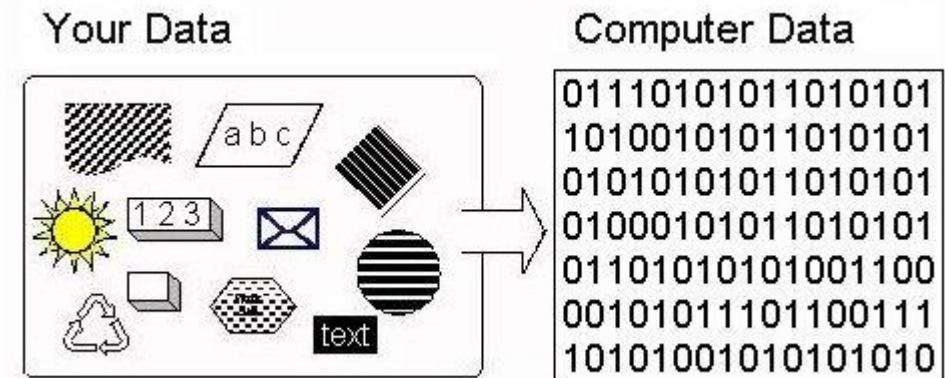
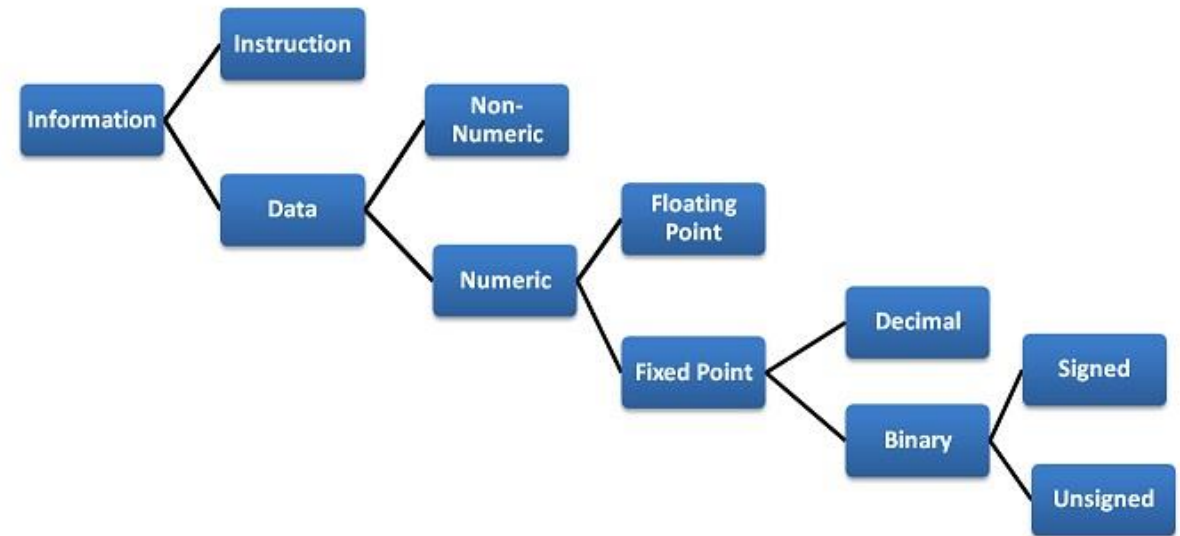


Contents

- Bit Patterns
- Binary Numbers
- Data Type formats
- Character representation
- Integer Representation
- Floating point number representation
- C language data types
- Operators in C

Data representation

- Data representation refers to the manner in which data is stored in the computer.
- There are several different formats for the data storage.
- It is important for computer problem solvers to understand the basic formats.



Why is it Important?

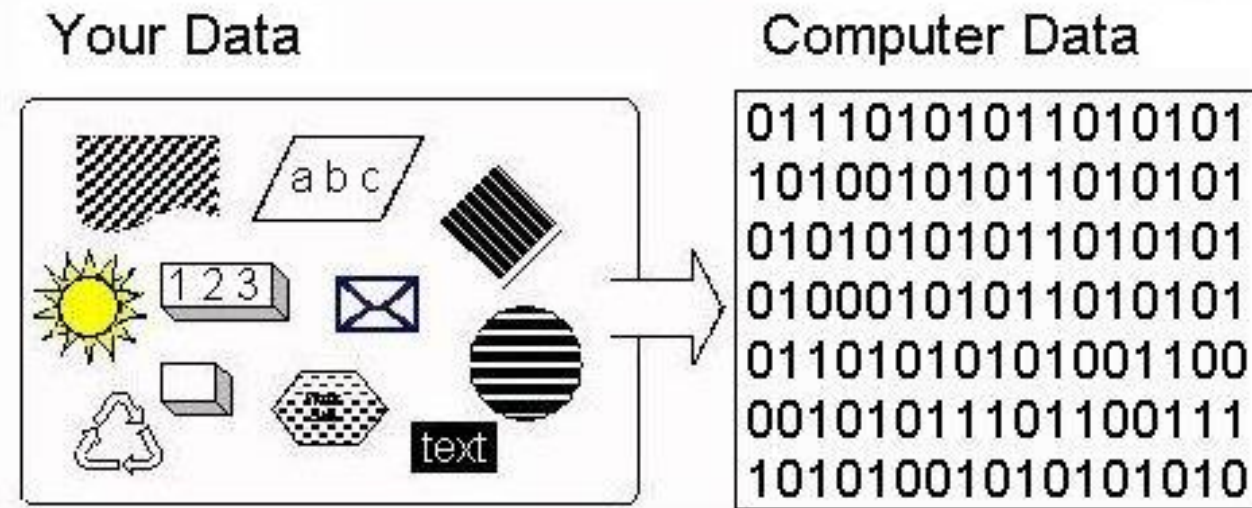
- As an example:
- We will learn that since we have finite storage, it is possible to overflow a storage location by trying to store too large a number.
- Most programming languages provide multiple data types each providing different length storage for variables
- It is up to the programmer to choose the data type with a length that won't overflow.
- Knowing how numbers are represented in storage helps one to understand this.

Entire Data types in c:

Data type	Size(bytes)	Range	Format string
Char	1	-128 to 127	%c
Unsigned char	1	0 to 255	%c
Short or int	2	-32,768 to 32,767	%i or %d
Unsigned int	2	0 to 65535	%u
Long	4	-2147483648 to 2147483647	%ld
Unsigned long	4	0 to 4294967295	%lu
Float	4	3.4 e-38 to 3.4 e+38	%f or %g
Double	8	1.7 e-308 to 1.7 e+308	%lf
Long Double	10	3.4 e-4932 to 1.1 e+4932	%lf

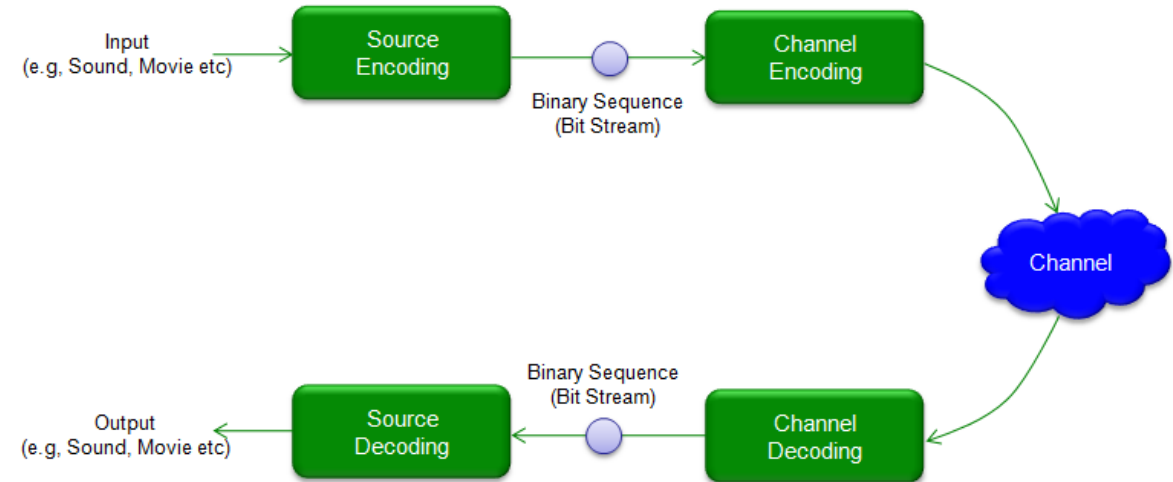
Bit pattern

- As you recall from an earlier slides , data may like various forms: characters, numbers, graphical, etc.
- All data is stored in the computer as a sequence of **Bits** (binary digits)
- This is a Universal storage format for all data types, and it is called a **bit pattern**.



Bits and Bytes

- A bit is the smallest unit of data stored in a computer and it has a value of 0 or 1.
- It is like a switch on(1) or Off(0)
- In computers, bits are stored electronically in RAM and auxiliary storage devices by two state digital circuits.
- The storage device itself doesn't know what the bit pattern represents, but software (application S/W, O/S, and I/O device firmware) stores and interprets the pattern
- That data is coded then stored and when retrieved it is decoded.
- A byte is a string of 8 bits and is called a character when data is text.



Binary Numbers

- Each Bit pattern is a binary number, that is a number represented by 0's and 1's rather than 0,1,2,3.....9 as decimal numbers are.
- For example, bit pattern like 1010 and 11011 are also binary numbers.
- Binary numbers are based on powers of 2 rather than powers of 10 as decimal numbers are

2	256	0
2	128	0
2	64	0
2	32	0
2	16	0
2	8	0
2	4	0
2	2	0
	1		

$\therefore 256_{10} = 100000000_2$

- For example, if one tries to store 256 in 1-byte there is overflow because the largest value storable in 8 bits is 255 as one can see from the following table:

Binary number:	1	1	1	1	1	1	1	1
Powers of 2:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal value:	128	64	32	16	8	4	2	1

- Note that $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$

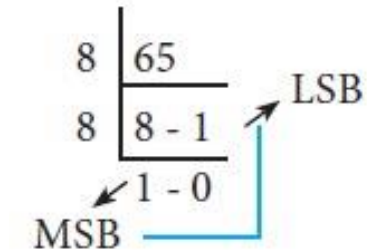
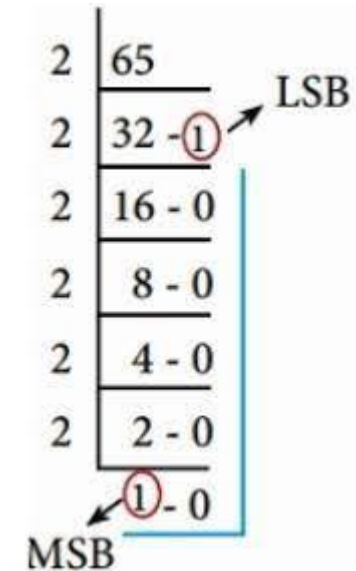
Data Types format

- As we have learned, fundamentally all data is stored as a bit pattern.
- But the different data types have different bit pattern formats.

- We want to learn the formats for:
 - **Characters** (for example, left, Lane, a, ?, \)
 - **Integers** (for example, 1, 453, -10, 0)
 - **Floating point numbers** (for example, 3.14159, 1.2, -567.235, 0.009)

Character representation

- The American standard code for information interchange (ASCII) is the scheme used to assign a bit pattern to each of the characters
- ASCII charts come in different flavors:
- Some have 7 bits string, some 8 or more
- Some show the binary code for the various characters as well as the code represented in other number systems e.g. decimal, hexadecimal and octal
- For example, the letter A has the ASCII code:
- 1000001 in n Binary, 65 in decimal, 41 in hexadecimal and 101 in Octal



$$(65)_{10} = (101)_8$$

Decimal Number 65 It Is Hexadecimal: 41

Subset of ASCII Chart

Char	Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char	Dec	Bin
Space	32	010 0000	0	48	011 0000	@	64	100 0000	P	80	101 0000	`	96	110 0000	p	112	111 0000
!	33	010 0001	1	49	011 0001	A	65	100 0001	Q	81	101 0001	a	97	110 0001	q	113	111 0001
"	34	010 0010	2	50	011 0010	B	66	100 0010	R	82	101 0010	b	98	110 0010	r	114	111 0010
#	35	010 0011	3	51	011 0011	C	67	100 0011	S	83	101 0011	c	99	110 0011	s	115	111 0011
\$	36	010 0100	4	52	011 0100	D	68	100 0100	T	84	101 0100	d	100	110 0100	t	116	111 0100
%	37	010 0101	5	53	011 0101	E	69	100 0101	U	85	101 0101	e	101	110 0101	u	117	111 0101
&	38	010 0110	6	54	011 0110	F	70	100 0110	V	86	101 0110	f	102	110 0110	v	118	111 0110
'	39	010 0111	7	55	011 0111	G	71	100 0111	W	87	101 0111	g	103	110 0111	w	119	111 0111
(40	010 1000	8	56	011 1000	H	72	100 1000	X	88	101 1000	h	104	110 1000	x	120	111 1000
)	41	010 1001	9	57	011 1001	I	73	100 1001	Y	89	101 1001	i	105	110 1001	y	121	111 1001
*	42	010 1010	:	58	011 1010	J	74	100 1010	Z	90	101 1010	j	106	110 1010	z	122	111 1010
+	43	010 1011	;	59	011 1011	K	75	100 1011	[91	101 1011	k	107	110 1011	{	123	111 1011
,	44	010 1100	<	60	011 1100	L	76	100 1100	\	92	101 1100	l	108	110 1100		124	111 1100
-	45	010 1101	=	61	011 1101	M	77	100 1101]	93	101 1101	m	109	110 1101	}	125	111 1101
.	46	010 1110	>	62	011 1110	N	78	100 1110	^	94	101 1110	n	110	110 1110	~	126	111 1110
/	47	010 1111	?	63	011 1111	O	79	100 1111	_	95	101 1111	o	111	110 1111			

Numeric Representation

- ASCII codes are an inefficient method for representing numbers.
- For example, the number 1024 using 8 bit ASCII would require four bytes or 32 bits of storage.
- Arithmetic operations on numbers represented in ASCII are very complicated.
- Representing the precision of a number, that is, the number of digits stored, may require large amount of space when stored in ASCII
- There are more efficient schemes for numbers.

Numeric Representation

- Integer Representation
 - Sign and modulus
 - One's complement
 - Two's complement
- Representation of Fractions
- Floating point or real.
- IEEE

Integer representation

- An integer is a whole number, that is a number without a decimal portion.
- Integers may positive, negative or zero
- A plus-sign (not required) or minus sign in front of the number is used to represent positive and negative numbers and zero
- There are two categories of integer representation : unsigned and signed



an Integer
is a whole number from the set of negative, non-negative, positive and 0 numbers.

- **Negative** : -1, -2, -3, -4, -5 ...
- **Non-negative** : 0, 6, 7, 8, 9 ...
- **Positive** : 1, 2, 3, 4, 5 ...
- **Zero** : 0 all by itself

How can we represent a binary number?

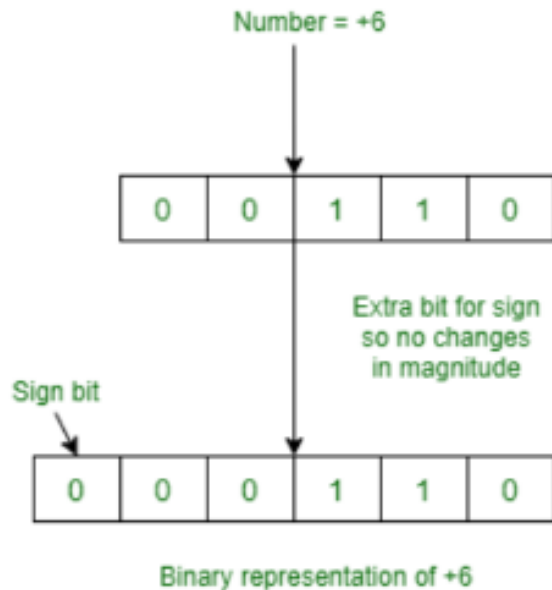
- Binary number can only be represented by a 0 or 1, so we cannot use a (-) to denote a negative number.
- Therefore, we cannot use a specialized symbol so it must be either a 0 or 1.
- There are THREE methods used, they are:
- These are:
 1. Sign-Magnitude method,
 2. 1's Complement method, and
 3. 2's complement method.

1. Signed Magnitude Method

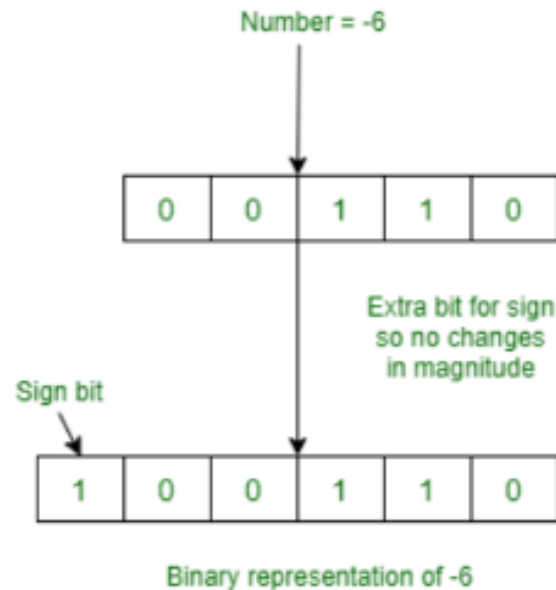
- Add an extra sign bit to recognize negative and positive numbers.
- Sign bit has 1 for negative number and 0 for positive

- E.g. with a 8 bit value

$$\bar{28} = \underbrace{0}_{\text{SIGN}} \underbrace{0011100}_{\text{Modulus}}$$



6 bit value



1 Signed Magnitude (SM)

- How one can represent the decimal number using sign bit representation:
- (i) Represent the decimal number as binary
- (ii) Left bit (MSB) used as the sign bit
- Only have 7 bits to express the number
- $12_{10} = 00001100$
- $-12_{10} = 10001100$
- How many representations are there for zero?

- For n bits register, MSB will be **sign bit** and **(n-1) bits will be magnitude**.
- Then, Negative lowest number that can be stored is $-(2^{(k-1)}-1)$ and positive largest number that can be stored is $(2^{(k-1)}-1)$.
- If $n=8$,
- MSB for sign digit, remaking 7 digits for magnitude

Range is $-2^{8-1}-1$ to $2^{8-1}-1$

$$=-2^7-1 \text{ to } 2^7-1$$

$$=-(128-1) \text{ to } (128-1)$$

$$=-127 \text{ to } 127$$

An 8-bit sign-magnitude representation, then, can represent any integer from $-127_{(10)}$ to $127_{(10)}$.

Example 1

- Add the numbers (+5) and (+3) using a computer.
- The numbers are assumed to be represented using 4-bit SM notation.

```
111  <- carry generated during addition
0101 <- (+5) First Number
+ 0011 <- (+3) Second Number
1000 <- (+8) Sum
```

Example 2

- Add the numbers (-4) and (+2) using a computer.
- The numbers are assumed to be represented using 4-bit SM notation.

```
000 <- carry generated during addition
1100 <- (-4) First number
+ 0010 <- (+2) Second Number
1110 <- (-2) Sum
```

Here, the computer has given the wrong answer of -6 = 1110, instead of giving the correct answer of -2 = 1010.

Disadvantages

- There are two notations for 0 (0000 and 1000), which is very inconvenient when the computer wants to test for a 0 result.
- It is not convenient for the computer to perform arithmetic.
- Due to the above mentioned ambiguities, SM notation is generally not used to represent signed numbers inside a computer.

2. One's Complement

- In digital electronics, the binary system is one of the most common number representation techniques.
- As its name suggest binary number system deals with only two number 0 and 1, this can be used by any device which is operating in two states.
- one's complement is changing or exchanging all the 0's into 1 and all the 1's into 0 of any number.
- **Method: Invert the ones and zeros**
- $11_{10} = 00001011$
- $-11_{10} = 11110100$
- **0 in MSB implies positive**
- **1 in MSB implies negative**

E.g 1. Write the one's complement of -34 in 8-bit 1's complement form

+ 34 = **0** **0** **1** **0** **0** **0** **1** **0**

 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

- 34 = **1** **1** **0** **1** **1** **1** **0** **1** (1's complement of + 34)

E.g. 2 - Represent -60 in 8-bit 1's complement form

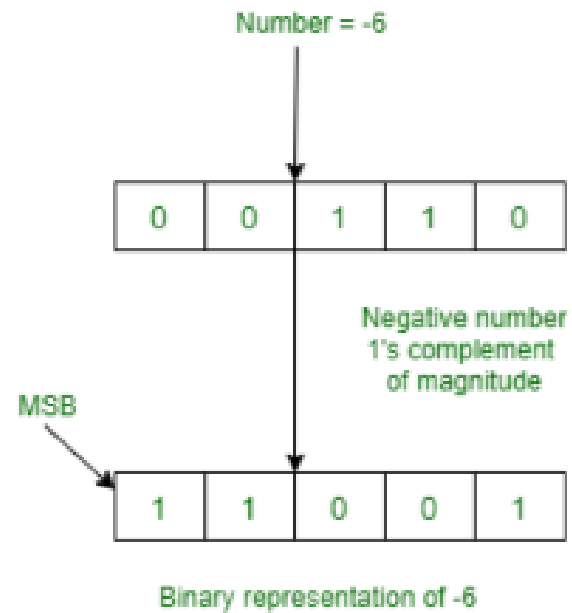
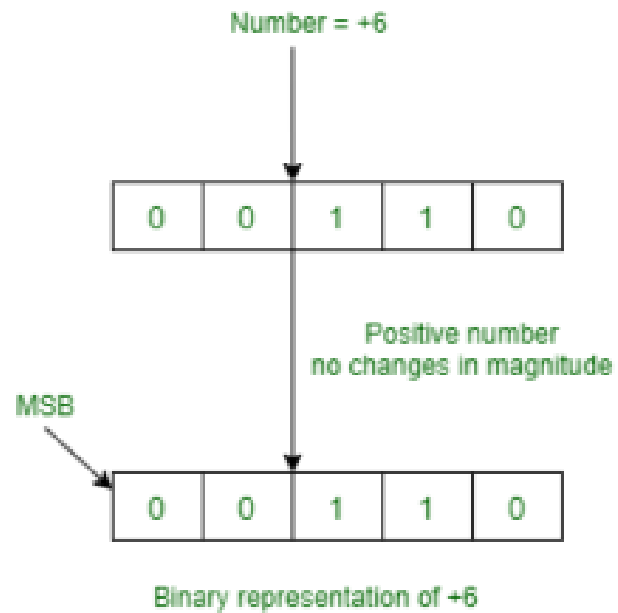
To represent **-60** in 1's complement form

+ 60 = **0 0 1 1 1 1 0 0**
 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

- 60 = **1 1 0 0 0 0 1 1** (1's complement of + 60)

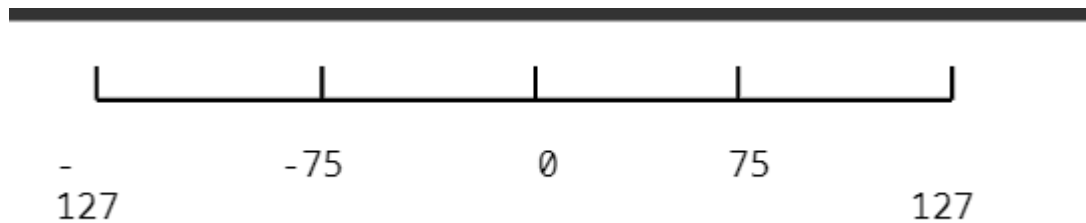
Example

- Write the one's complement of 6



One's Complement

- What is a complement ?
- It is the opposite of something.
- Because computers do not like to subtract, this method finds the complement of a positive number and then addition can take place.



One's Complement

- Example Find the one's complement of +100
- 1. Convert +100 to binary
- 2. Swap all the bits.
- 3. Check answer by adding 127 to your result.

Example

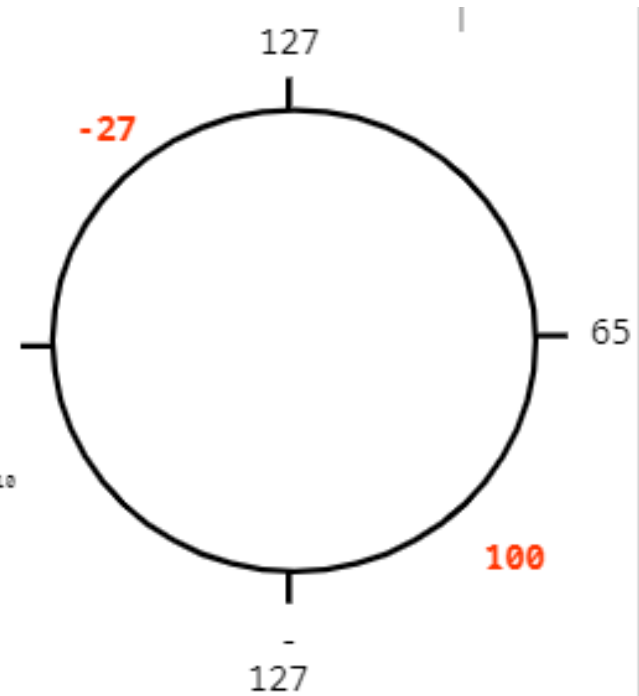
1. $+100 = 01100100_2$

2. 01100100_2

$$\underline{10011011_2}$$

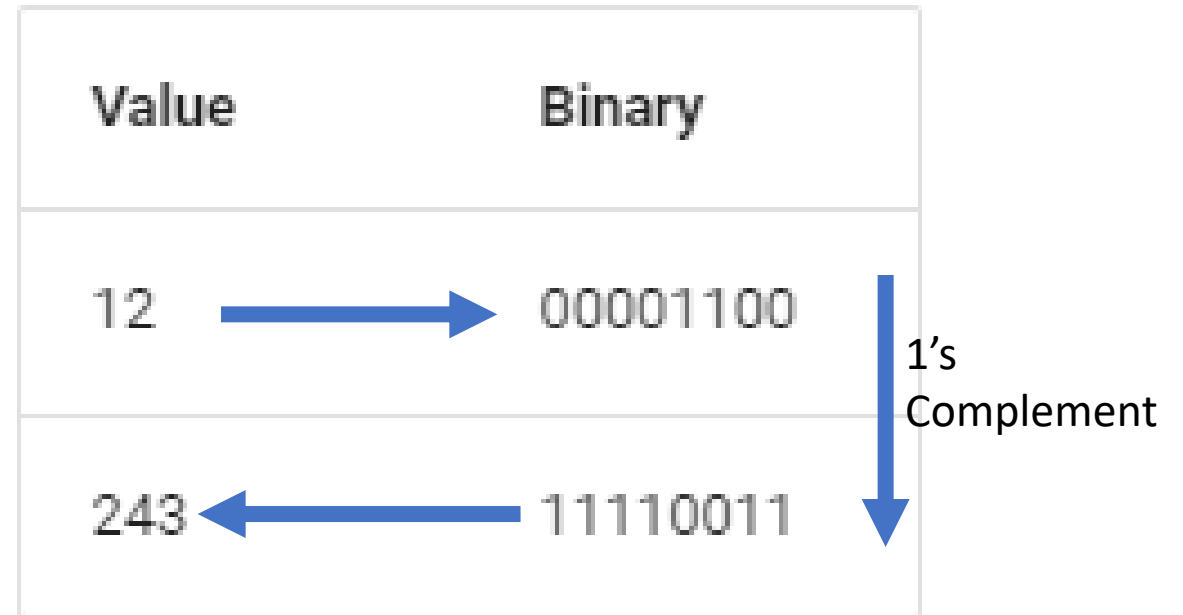
3. $10011011_2 = -27_{10}$

4. $127_{10} + -27_{10} = 100_{10}$



Negative -1s Complement representation

- We can represent 1s complement using the following formula:
 - $-x = 2^n - x - 1$
- Consider a number 12.
- If we are using 8 bits 1s complement number, we can represent -12 as -the binary representation of $-12 = 2^8 - 12 - 1 = 243$.



(3) Two's Complement

- How can negative numbers be represented using only binary 0's and 1's so that a computer can “read” them accurately?
- Consider the binary numbers from 0000 to 1111 (i.e., 0 to 15 in base ten)
- and,
- 1001 to 1111 will represent the negative numbers -1 → -7 respectfully.
- In a computer, numbers are stored in registers where there is reserved a designated number of bits for the storage of numbers in binary form.
- It is easy to change a negative integer in base ten into binary form using the method of two's complement.

Two's Complement

Step 1: Write the absolute value of the given number in binary form. Prefix this number with 0 indicate that it is positive.


Step 2: Take the complement of each bit by changing zeroes to ones and ones to zero.

Step 3: Add 1 to your result.


Output is the two's complement representation of the negative integer.

- **EXAMPLE:**


Find 2's complement of -20

Step 1: $20_{10} = 00010100_2$  **Step 1**

0 0 0 1 0 1 0 0 → Binary number

1 1 1 0 1 0 1 1 → One's complement  **Step 2**

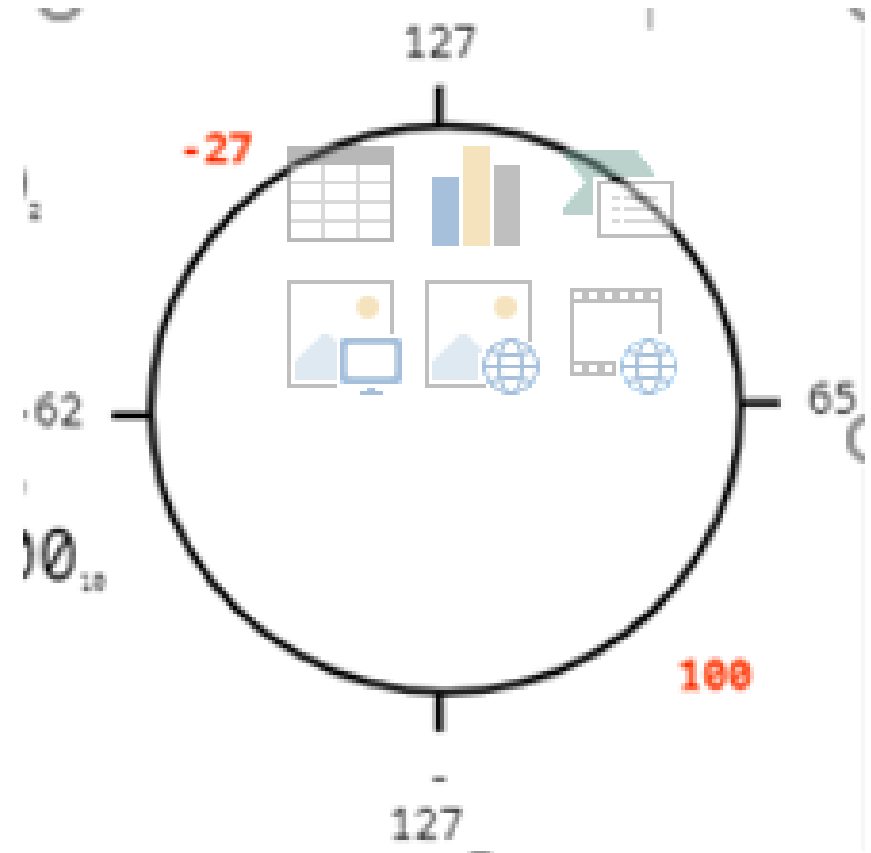
1 1 1 0 1 0 1 1
+ 1
1 1 1 0 1 1 0 0

 **Step 3**

→ 2s complement

Two's Complement

- 8 bits two's complement
- Find 2's complement of -63
- Step 1: Absolute value of Binary with prefixes -63 = 00111111
- Step 2: take one's complement, 11000000
- Step 3: Add 1
- Answer 11000001
-



Floating point numbers

- A floating point number, is a positive or negative whole number with a decimal point.
- For example, 5.5, 0.25, and -103.342 are all floating point numbers
- There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases.

Floating point numbers

- Example 1
- 85.125
- $85 = 101001$
- $0.125 = 001$
- $85.125 = 1010101.001$
 $= 1.0101001 \times 2^6$

A simple program in C

- Consider the following simple C program

- `#include <stdio.h>`

```
int main() {  
    printf("Hello World!");  
    return 0;  
}
```

- `h` permits the programmer to carry out input/output operations .
- Standard Input Output Header, or "`stdio.h`," contains features for handling file and console I/O, including the "`printf()`" function for formatted output
- and the "`scanf()`" function for formatted input.

The enhancements of the simple program

- #include <stdio.h>

```
int main() {  
    printf("Welcome to JSP!\n");  
    printf("I am studying C.");  
    return 0;  
}
```

What is **\n** exactly?

The newline character(\n)

Welcome to JSP
I am studying C

Few valid escape sequences

Escape Sequence	Description	Example	Output
\t	Creates a horizontal tab	<pre>#include <stdio.h> int main() { printf("Welome to SJP University!\t"); printf("I am Studying C."); return 0; }</pre>	Welcome to SJP University I am studying C
\\	Inserts a backslash character (\)	<pre>#include <stdio.h> int main() { printf("Welcome to SJP University!\\"); printf("I am studying C."); return 0; }</pre>	Welcome to SJP University!\\I am studying C.
\"	Inserts a double quote character	<pre>#include <stdio.h> int main() { printf("They call as \"Sri Jayewardene pura University\"."); return 0; }</pre>	They call as "Sri Jayewardene pura University".

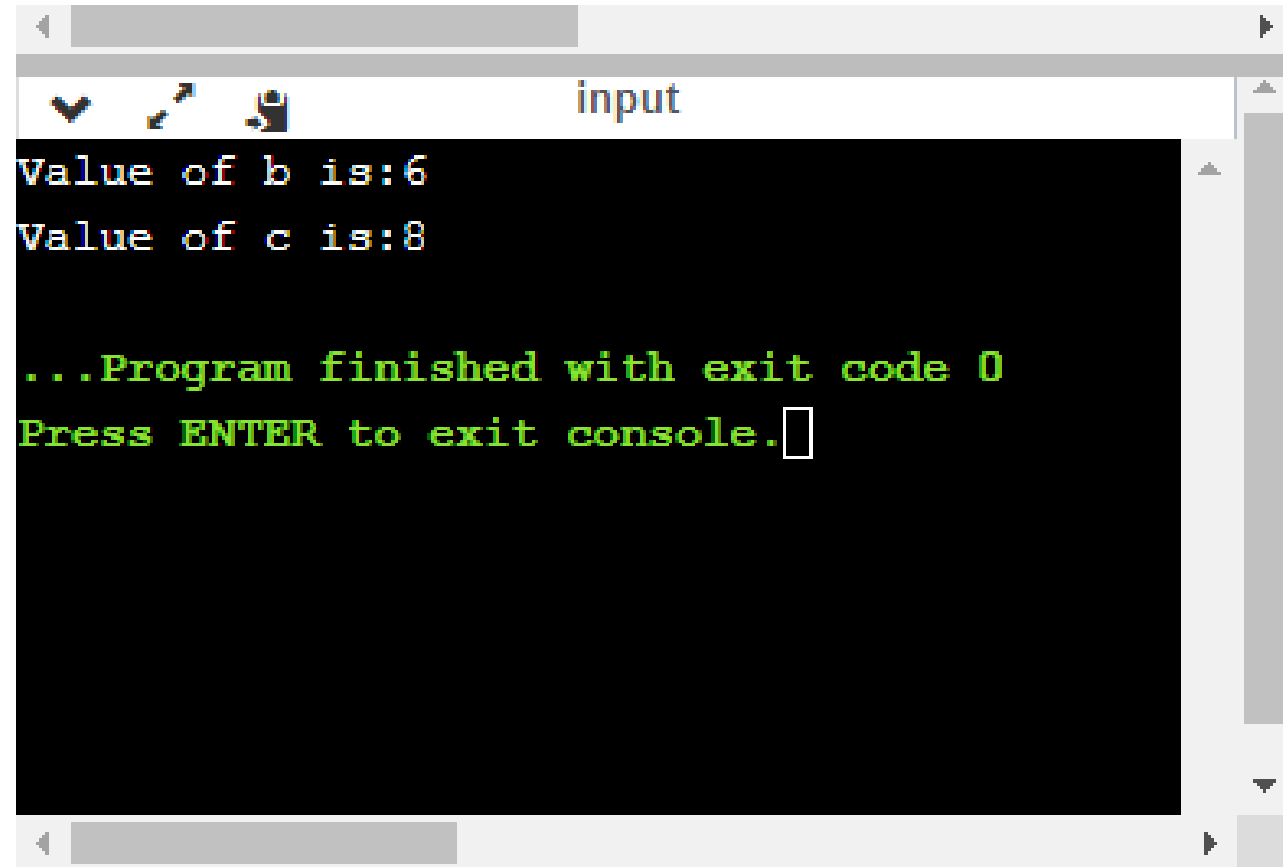
Format Specifiers in C

Format specifier	Description
%d or %i	It is used to print the signed integer value
%u	It is used to print the unsigned integer value
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value
%e/%E	It is used for scientific notation. It is also known as Mantissa or Exponent.
%g	It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value after the decimal in input would be exactly the same as the value in the output.
%p	It is used to print the address in a hexadecimal form.
%c	It is used to print the unsigned character.
%s	It is used to print the strings.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'

Let's understand the format specifiers in detail through an example. (example 1)

```
int main()
{
    int b=6;
    int c=8;
    printf("Value of b is:%d", b);
    printf("\nValue of c is:%d",c);

    return 0;
}
```

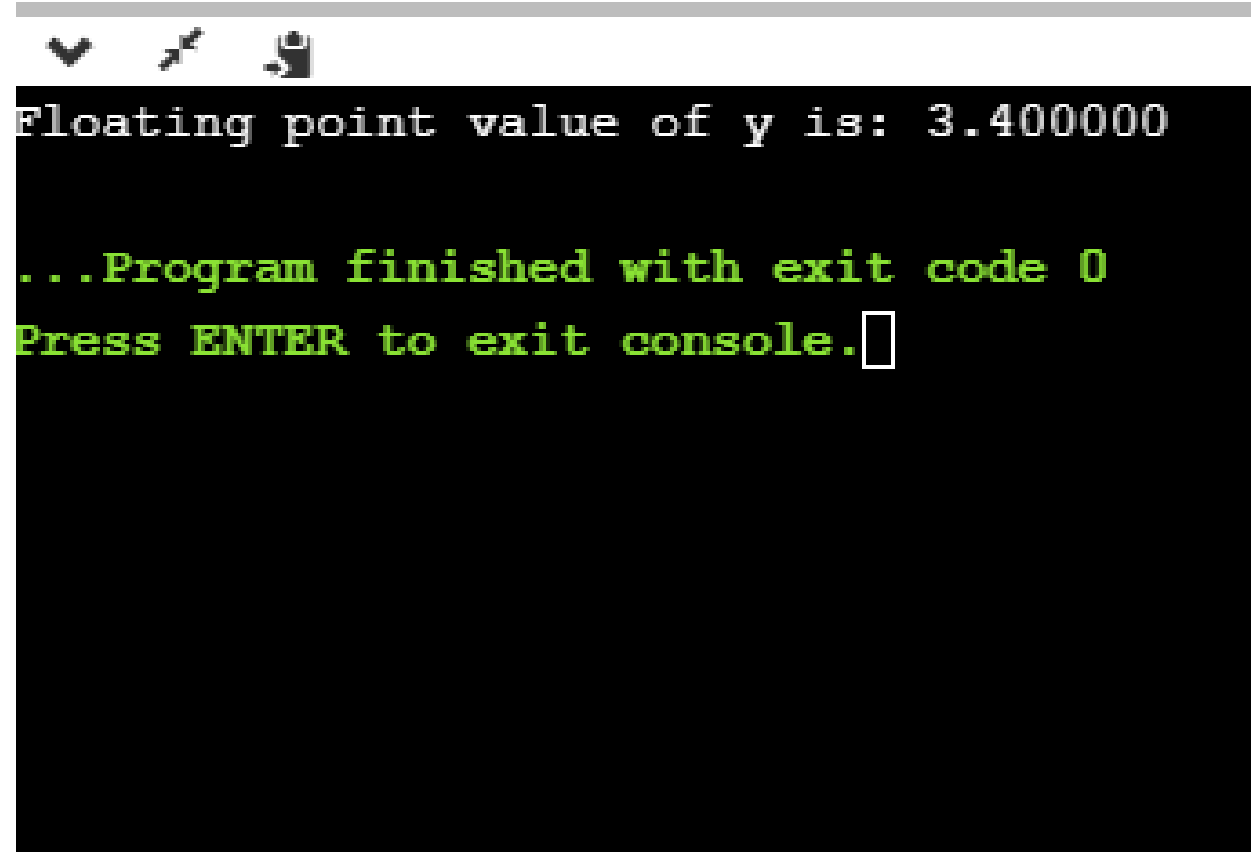


```
input
Value of b is:6
Value of c is:8

...Program finished with exit code 0
Press ENTER to exit console.█
```

Let's understand the format specifiers in detail through an example. (example 2)

```
int main()
{
    float y=3.4;
    printf("Floating point value of y is
: %f", y);
    return 0;
}
```

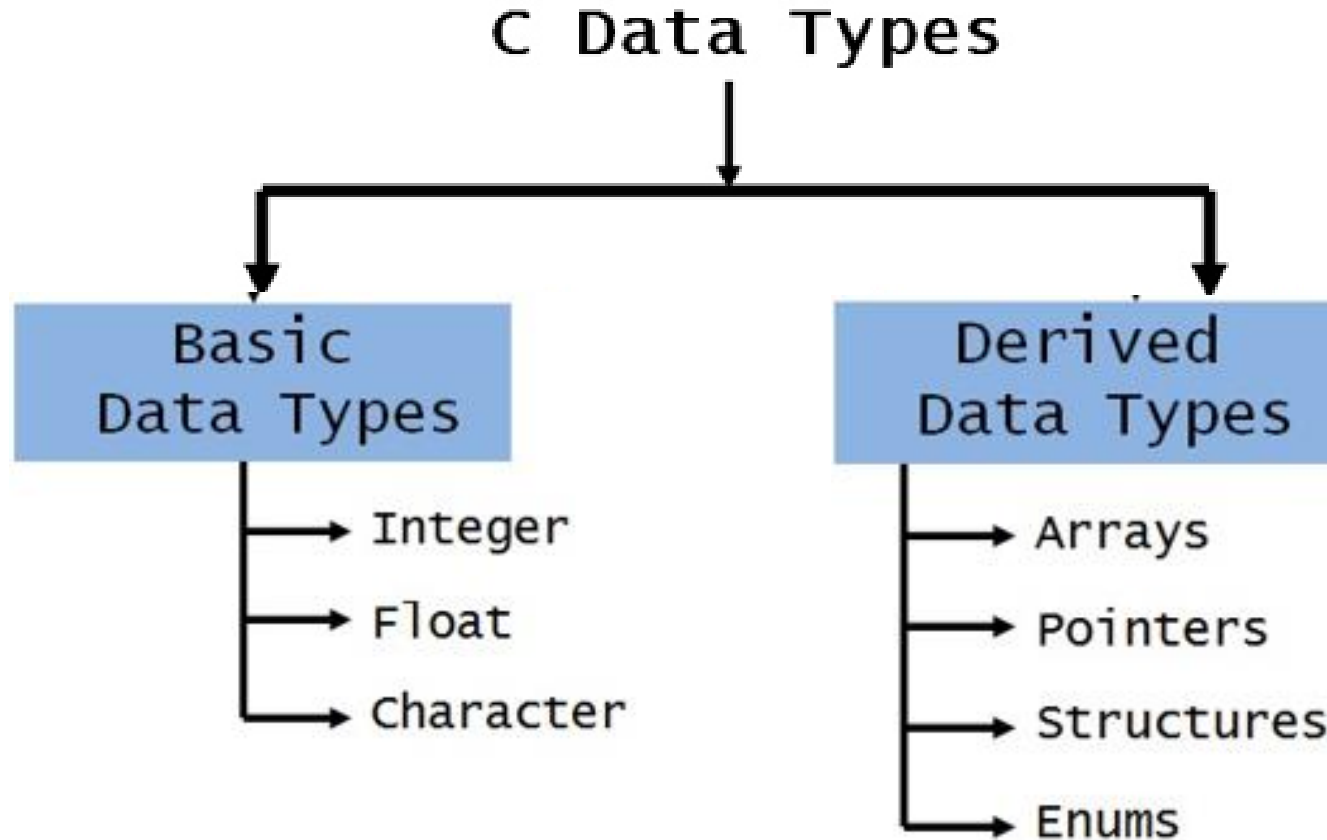
A screenshot of a terminal window with a dark background. At the top, there are three small icons: a downward arrow, a magnifying glass, and a document. The terminal displays the output of the C program: "Floating point value of y is: 3.400000" in white text. Below this, it shows "...Program finished with exit code 0" and "Press ENTER to exit console." in green text. A white cursor is visible at the end of the last line.

```
Floating point value of y is: 3.400000
...Program finished with exit code 0
Press ENTER to exit console.
```

Comments in C language

- Comments can be used to explain code, and to make it more readable.
 - It can also be used to prevent execution when testing alternative code.
 - Single-line comments start with two forward slashes (//)
 - Any text between // and the end of the line is ignored by the compiler (will not be executed).
 - Multi-line comments start with /* and ends with */
 - Any text between /* and */ will be ignored by the compiler:
- **Example 1**
 - `// This is a comment`
`printf("Hello World!");`
 - **Example 2**
 - `printf("Hello World!"); // This is a comment`
 - **Example 3**
 - `/* The code below will print the words Hello World!`
`to the screen, and it is amazing */`
`printf("Hello World!");`

C language data types



Basic data types

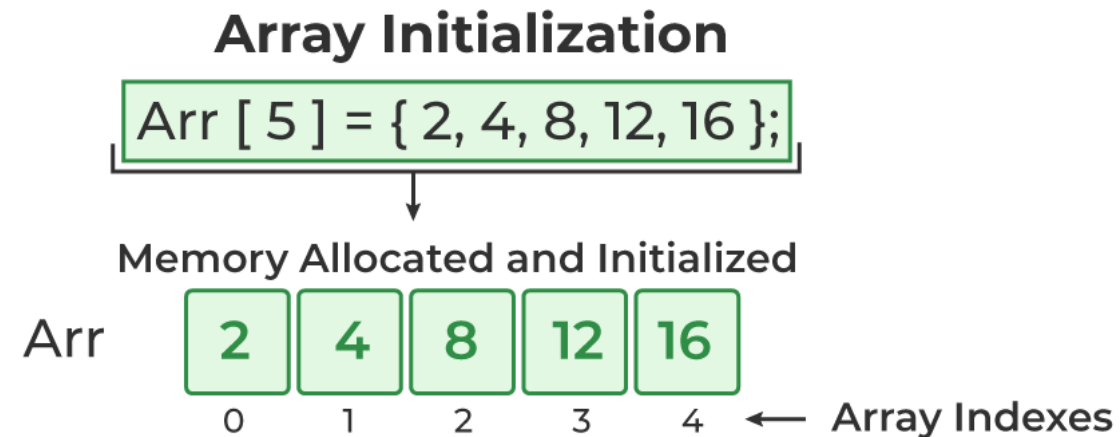
Data Types	Description
int	The most natural size of integer for the machine. An is a whole that can be positive, negative, or zero. Examples of integers are: -5, 1, 5, 8, 97, and 3,043
float	.A single-precision floating point value. A floating point number, is a positive or negative whole number with a decimal point. For example, 5.5, 0.25, and -103.342 are all floating point numbers, while 91, and 0 are not.
double	A double-precision floating point value. The double in C is a data type that is used to store high-precision floating-point data or numbers (up to 15 to 17 digits). It is used to store large values of decimal numbers
void	Represents the absence of type
char	Typically a single octet(one byte) char is the data type that represents a single character. The char values are encoded with one byte code, by the ASCII table

Basic data Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or - 2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

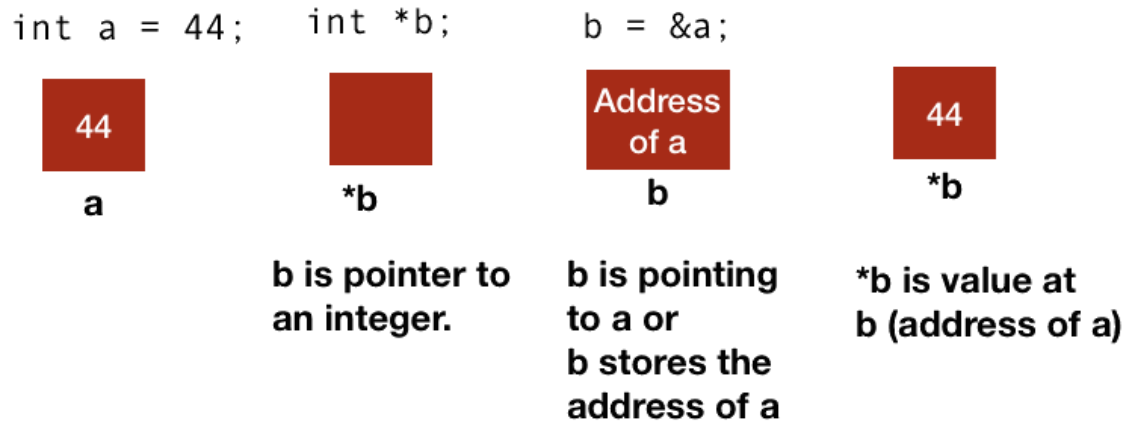
Derived data Types

- The C language supports a few derived data types:
- **Arrays** – The *array* basically refers to a sequence (ordered sequence) of a finite number of data items from the same data type sharing one common name.
- We can form arrays by collecting various primitive data types such as int, float, char, etc.
- We create a collection of these data types and store them in the contiguous memory location.



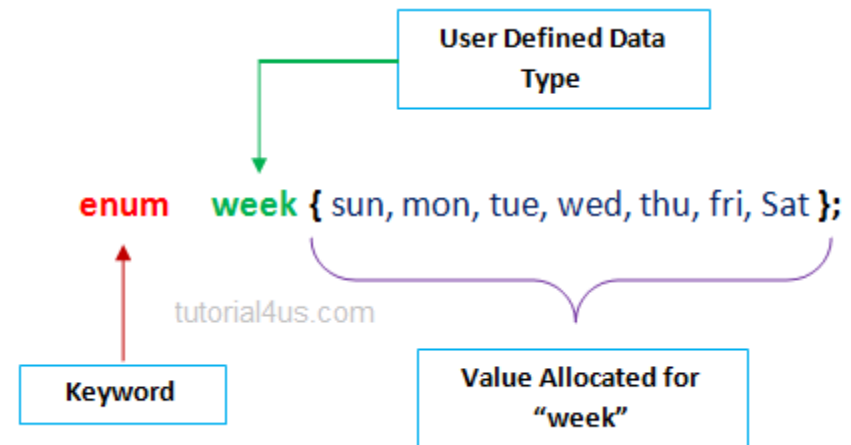
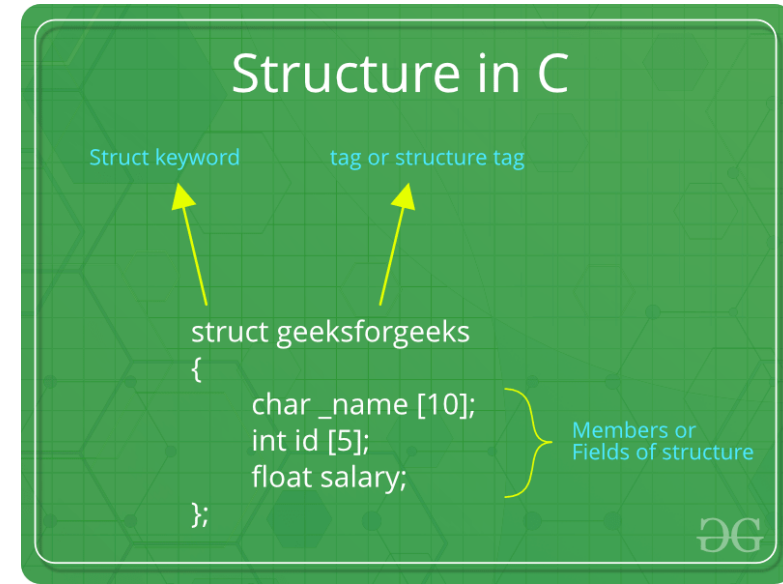
Derived data Types

- **Pointers:** The *Pointers in C* language refer to some special form of variables that one can use for holding other variables' addresses



Derived data Types

- **Structures** – A collection of various different types of data type items that get stored in a contiguous type of memory allocation is known as *structure in C*.
- **Enumeration** or Enum in C is a special kind of data type defined by the user. It consists of constant integrals or integers that are given names by a user



1st rule of Programming:

If it works.... don't touch it!..



Keywords

- Keywords are predefined, reserved words used in programming that have special meanings to the compiler.
- Keywords are part of the syntax and they cannot be used as an identifier.
- example:

```
int money;
```

Here, **int** is a keyword that indicates 'money' is a variable of type integer.

Keywords

- As C is a case sensitive language, all its 32 keywords must be written in lowercase.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Identifiers

- Identifier refers to name given to entities such as variables, functions, structures etc.
- Identifier must be unique. They are created to give unique name to a entity to identify it during the execution of the program.
- For example:

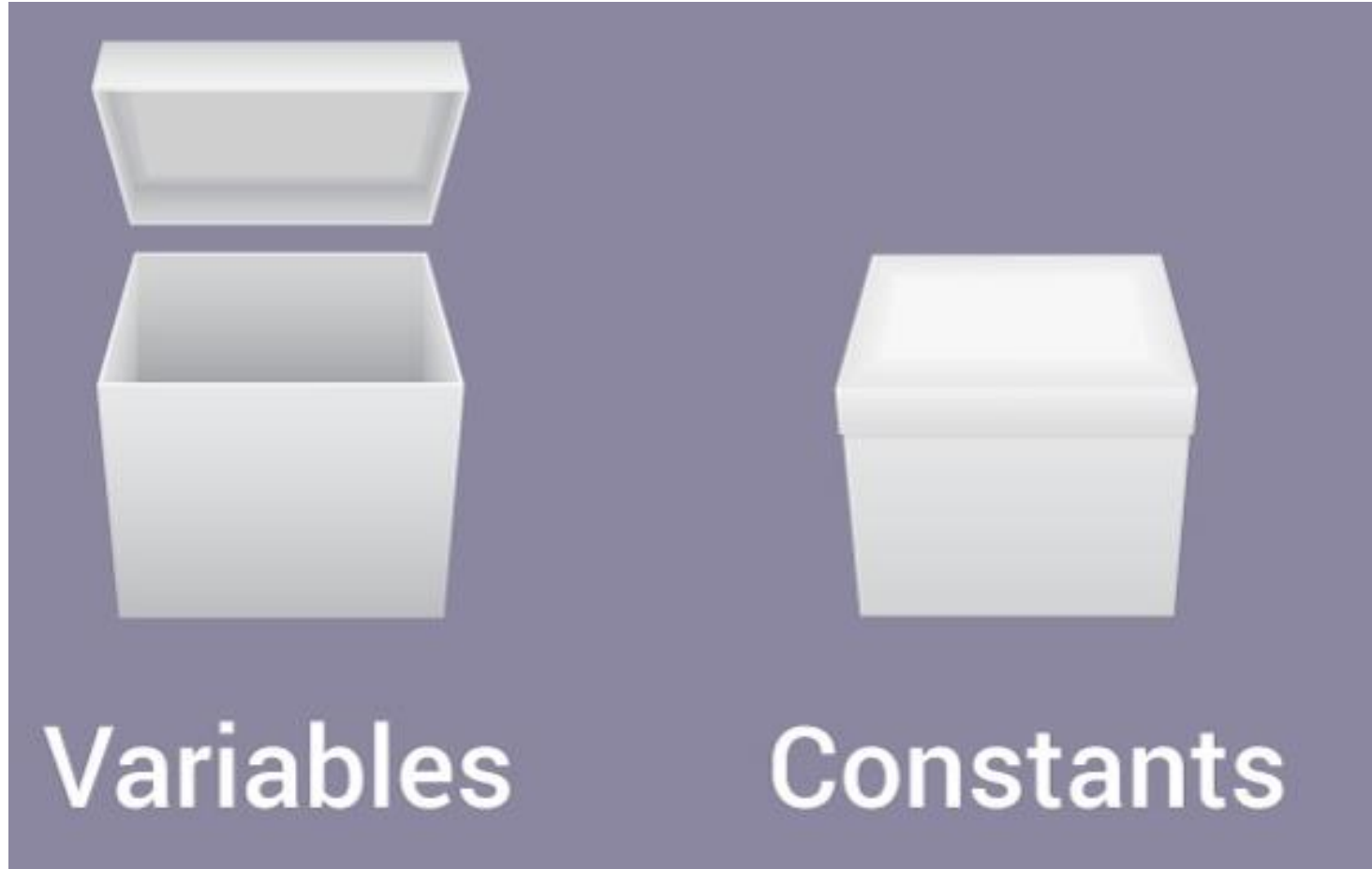
```
int money;  
double accountBalance;
```

- Here, `money` and `accountBalance` are identifiers.
- Identifier names must be different from keywords. You cannot use `int` as an identifier because `int` is a keyword.

Rules for define an Identifier

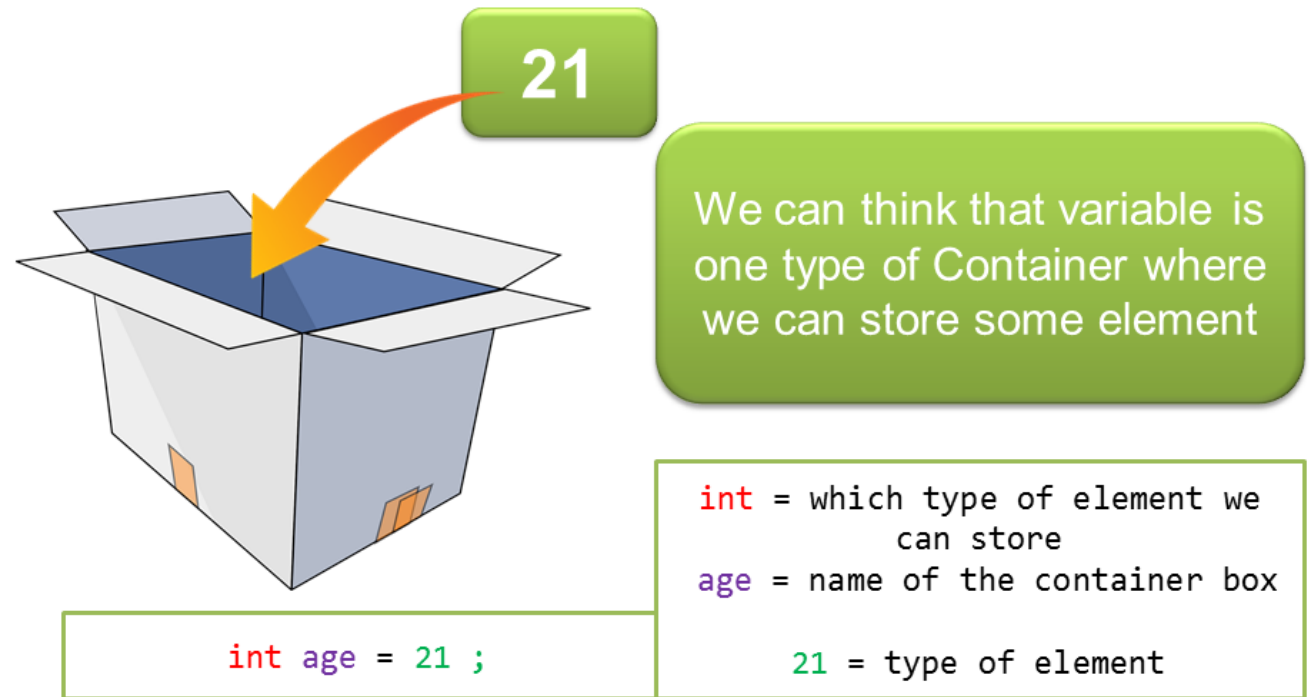
- 1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscore only.**
- 2. The first letter of an identifier should be either a letter or an underscore.**
 - However, it is discouraged to start an identifier name with an underscore.
- 3. There is no rule on length of an identifier.**
 - However, only the first 31 characters of a identifier are checked by the compiler.

Variables & Constants



Variables

- In programming, a variable is a container (storage area) to hold data
- To indicate the storage area, each variable should be given a unique name (identifier).
- A variable is something that may change in value.
- Variable names are just the symbolic representation of a memory location
- Example : page number, the air temperature each day, or the exam marks given to a class of school children.



Rules for naming a variable in C

1. A variable name can have letters (both uppercase and lowercase letters), digits and underscore only.
2. The first letter of a variable should be either a letter or an underscore.
 - However, it is discouraged to start an identifier name with an underscore.
3. There is no rule on length of an identifier.
 - However, only the first 31 characters of a identifier are checked by the compiler.
4. C is a strongly typed language. What this means it that, **the type of a variable cannot be changed.**

Sample programming C -Variables and Math

```
/*  
    Variables and Calculations.c  
  
    Add two integer values together and display the result.  
*/  
  
#include "simpletools.h"                // Include simpletools  
  
int main()                             // main function  
{  
    int a = 25;                        // Initialize a variable to 25  
    int b = 17;                        // Initialize b variable to 17  
    int c = a + b;                     // Initialize c variable to a + b  
    print("c = %d ", c);              // Display decimal value of c  
}
```

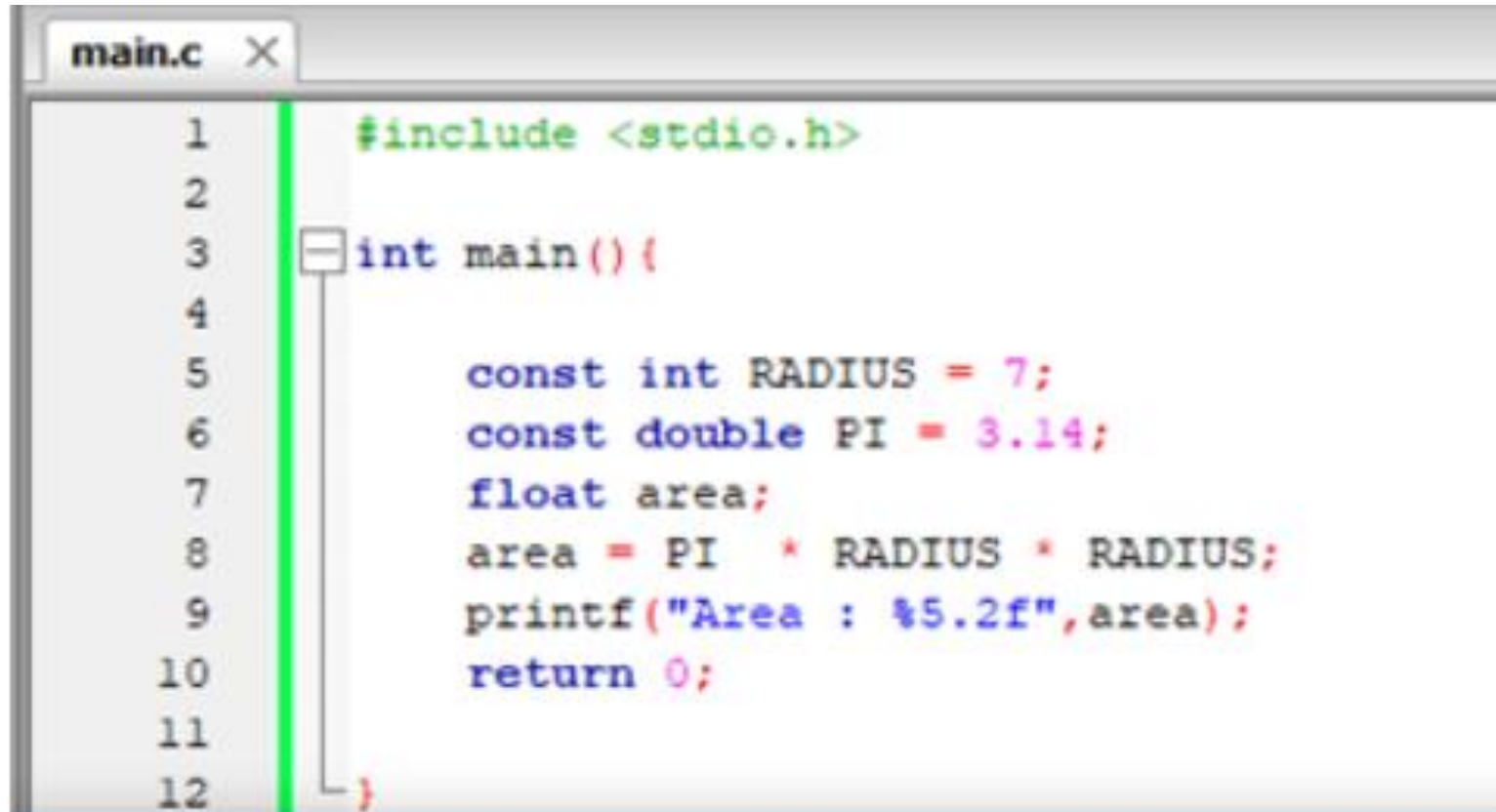
Constants/Literals

- A constant is a value or an identifier whose value cannot be altered in a program.
- For example; `const double PI = 3.14;`
 - Here, PI is a constant.
 - Basically what it means is that, PI and 3.14 is same for this program.

`const int x = 5;`

Constant Literal

Sample program with use of constants



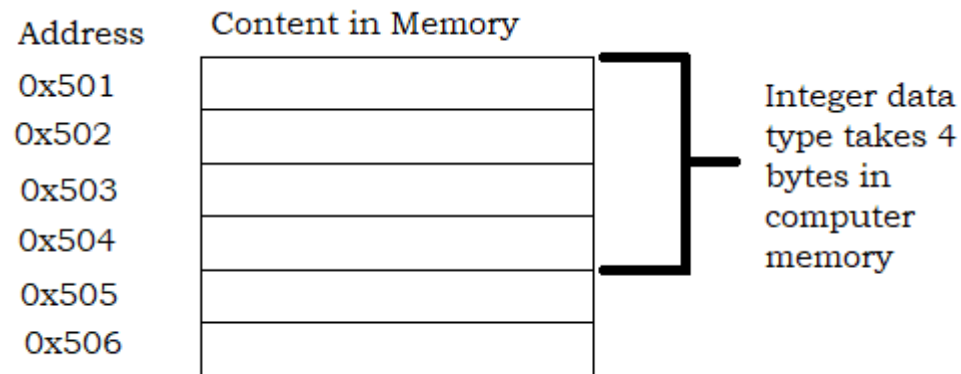
```
1  #include <stdio.h>
2
3  int main() {
4
5      const int RADIUS = 7;
6      const double PI = 3.14;
7      float area;
8      area = PI * RADIUS * RADIUS;
9      printf("Area : %5.2f", area);
10     return 0;
11
12 }
```

Answer :

Area: 153.86

sizeof()

- A computer's memory is a collection of byte-addressable chunks.



- **sizeof()** is a built-in function that is used to calculate the size (**in bytes**) that a data type occupies in the **computer's memory**.
 - Suppose that a variable x is of type integer and takes four bytes of the computer's memory, then sizeof(x) would return four.

sizeof()

- This function is a unary operator (i.e., it takes in one argument).
- This argument can be a;
 - Data type: The data type can be primitive (e.g., `int`, `char`, `float`) or user-defined (e.g., `struct`).
 - Expression
- IMPORTANT NOTICE:
 - The result of the `sizeof()` function is machine-dependent since the sizes of data types in C varies from system to system.

sizeof operator in C

- **Usage of sizeof() operator**
- *sizeof()* operator is used in different ways according to the operand type.
- **When the operand is a Data Type:** When *sizeof()* is used with the data types such as int, float, char... etc
- it simply returns the amount of memory allocated to that data types

- Sample program – size()

```
// C Program To demonstrate
// sizeof operator
#include <stdio.h>
int main()
{
    printf("%lu\n", sizeof(char));
    printf("%lu\n", sizeof(int));
    printf("%lu\n", sizeof(float));
    printf("%lu", sizeof(double));
    return 0;
}
```

Output

1
4
4
8

sizeof()

```
#include<stdio.h>

int main() {
    int x = 20;
    char y = 'a';
    //Using variable names as input
    printf("The size of int is: %d\n", sizeof(x));
    printf("The size of char is %d\n", sizeof(y));
    printf("The size of x + y is: %d\n", sizeof(x+y));
    //Using datatype as input
    printf("The size of float is: %d\n", sizeof(float));
    printf("The size of double is: %d\n", sizeof(double));
    return 0;
}
```

```
The size of int is: 4
The size of char is 1
The size of x + y is: 4
The size of float is: 4
The size of double is: 8
```

Size qualifiers

- Size qualifiers alters the size of a basic type.
- There are two size qualifiers, **long** and **short**.
- For example:

```
long double x;      double x;  
                     long double lx;  
  
printf("size of x: = %d\n", sizeof(x));  
printf("size of lx: = %d", sizeof(lx));
```

- The size of double is 8 bytes.
- However, when **long** keyword is used, that variable becomes 16 bytes.

Size qualifiers

- Size qualifiers alters the size of a basic type.
- There are two size qualifiers, `long` and `short`.
- The size of `int` is 4 bytes.
- However, when `short` keyword is used, that variable becomes 2 bytes.

```
int x;  
short sx;  
  
printf("size of x: = %d\n", sizeof(x));  
printf("size of sx: = %d", sizeof(sx));
```

Sign qualifiers

- Integers and floating point variables can hold both negative and positive values.
- However, if a variable needs to hold positive value only, unsigned data types are used.
- For example: `unsigned int x = 3;`
- An `int` is signed by default, meaning it can represent both positive and negative values.
- An unsigned is an integer that **can never be negative**.

Operators

- C programming has various types of operators to perform tasks including arithmetic, conditional and bitwise operations.
- Operators in C programming are;
 - Arithmetic Operators
 - Increment & Decrement Operators
 - Assignment Operators
 - Relational Operators
 - Logical Operators
 - Conditional Operators
 - Bitwise Operators
 - Special Operators

Arithmetic Operators

- An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multifaction
/	Division
%	Remainder after division (modulo division)

Arithmetic Operators

```
int 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 = %d\n", c);  
}
```

```
a+b = 13  
a-b = 5  
a*b = 36  
a/b = 2  
Remainder = 1
```

Increment & Decrement Operators

- C programming has two increment and decrement operators to change the value of an operand (constant or variable) by 1.
 - Increment **++** increases the value by 1
 - Decrement **--** decreases the value by 1
- These two operators are unary operators, meaning they only operate on a single operand.
- The operators ++ and -- can be used as prefix or postfix.

Increment & Decrement Operators

```
int main() {  
    int a = 7;  
    float b = 5.5;  
    printf("++a = %d\n", ++a);  
    printf("--b = %.2f\n", --b);  
    printf("a++ = %d\n", a++);  
    printf("b-- = %.2f\n", b--);  
    printf("Final Values: a = %d, b = %.2f\n", a, b);  
}
```

Increment & Decrement Operators

```
int main() {  
    int a = 7;  
    float b = 5.5;  
    printf("++a = %d\n", ++a);  
    printf("--b = %.2f\n", --b);  
    printf("a++ = %d\n", a++);  
    printf("b-- = %.2f\n", b--);  
    printf("Final Values: a = %d, b = %.2f\n", a, b);  
}
```

```
++a = 8  
--b = 4.50  
a++ = 8  
b-- = 4.50  
Final Values: a = 9, b = 3.50
```

Assignment Operators

- An assignment operator is used for assigning a value to a variable.
 - The most common assignment operator is =

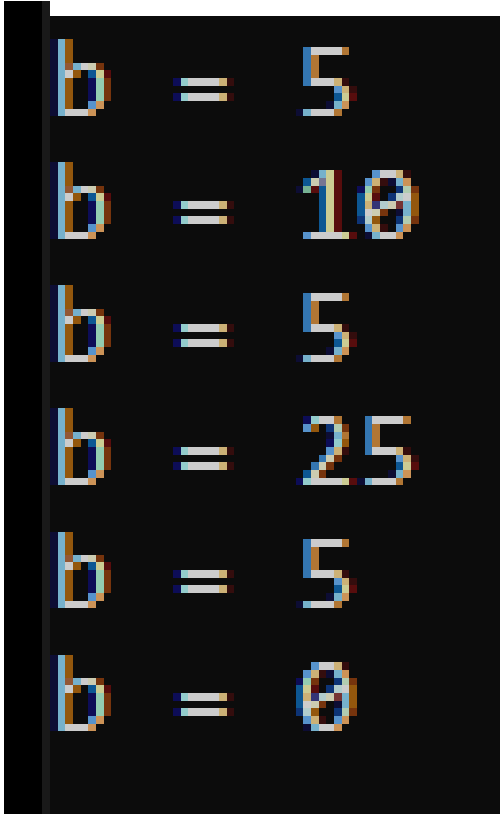
Operator	Example	Same as...
=	a = b	a = b
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

Assignment Operators

```
int main() {  
    int a = 5, b;  
    b = a;  
    printf("b = %d\n", b);  
    b += a;  
    printf("b = %d\n", b);  
    b -= a;  
    printf("b = %d\n", b);  
    b *= a;  
    printf("b = %d\n", b);  
    b /= a;  
    printf("b = %d\n", b);  
    b %= a;  
    printf("b = %d\n", b);  
}
```

Assignment Operators

```
int main() {  
    int a = 5, b;  
    b = a;  
    printf("b = %d\n", b);  
    b += a;  
    printf("b = %d\n", b);  
    b -= a;  
    printf("b = %d\n", b);  
    b *= a;  
    printf("b = %d\n", b);  
    b /= a;  
    printf("b = %d\n", b);  
    b %= a;  
    printf("b = %d\n", b);  
}
```



b = 5
b = 10
b = 5
b = 25
b = 5
b = 0

Relational Operators

- A relational operator checks the relationship between two operands
- If the relation is **true**, it returns **1**; if the relation is **false**, it returns value **0**
- Relational operators are used in decision making and loops.

Operator	Meaning	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 returns 0

Relational Operators

```
int main() {  
  
    int a = 5, b = 5, c = 10;  
  
    printf("%d == %d = %d\n", a, b, a == b);  
    printf("%d == %d = %d\n", a, c, a == c);  
  
    printf("%d > %d = %d\n", a, b, a > b);  
    printf("%d > %d = %d\n", a, c, a > c);  
  
    printf("%d < %d = %d\n", a, b, a < b);  
    printf("%d < %d = %d\n", a, c, a < c);  
  
    printf("%d != %d = %d\n", a, b, a != b);  
    printf("%d != %d = %d\n", a, c, a != c);  
  
    printf("%d >= %d = %d\n", a, b, a >= b);  
    printf("%d >= %d = %d\n", a, c, a >= c);  
  
    printf("%d <= %d = %d\n", a, b, a <= b);  
    printf("%d <= %d = %d\n", a, c, a <= c);  
  
}
```

Relational Operators

```
int main() {  
  
    int a = 5, b = 5, c = 10;  
  
    printf("%d == %d = %d\n", a, b, a == b);  
    printf("%d == %d = %d\n", a, c, a == c);  
  
    printf("%d > %d = %d\n", a, b, a > b);  
    printf("%d > %d = %d\n", a, c, a > c);  
  
    printf("%d < %d = %d\n", a, b, a < b);  
    printf("%d < %d = %d\n", a, c, a < c);  
  
    printf("%d != %d = %d\n", a, b, a != b);  
    printf("%d != %d = %d\n", a, c, a != c);  
  
    printf("%d >= %d = %d\n", a, b, a >= b);  
    printf("%d >= %d = %d\n", a, c, a >= c);  
  
    printf("%d <= %d = %d\n", a, b, a <= b);  
    printf("%d <= %d = %d\n", a, c, a <= c);  
  
}
```

```
5 == 5 = 1  
5 == 10 = 0  
5 > 5 = 0  
5 > 10 = 0  
5 < 5 = 0  
5 < 10 = 1  
5 != 5 = 0  
5 != 10 = 1  
5 >= 5 = 1  
5 >= 10 = 0  
5 <= 5 = 1  
5 <= 10 = 1
```

Logical Operators

- An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND , true only if all operands are true	If c = 5 & d = 2, then, expression ((c==5)&&(d==5)) equals to 0
	Logical OR , true only if either operand is true	If c = 5 & d = 2, then, expression ((c==5)&&(d==5)) equals to 1
!	Logical NOT , true only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0

Logical Operators

```
int main() {  
  
    int a = 5, b = 5, c = 10, result;  
  
    result = (a == b) && (c > b);  
    printf("(a == b) && (c > b) equals to %d\n", result);  
  
    result = (a == b) && (c < b);  
    printf("(a == b) && (c < b) equals to %d\n", result);  
  
    result = (a == b) || (c > b);  
    printf("(a == b) || (c > b) equals to %d\n", result);  
  
    result = (a != b) || (c > b);  
    printf("(a != b) || (c > b) equals to %d\n", result);  
  
    result = !(a != b);  
    printf("!(a != b) equals to %d\n", result);  
  
    result = !(a == b);  
    printf("!(a == b) equals to %d\n", result);  
}
```

Logical Operators

```
int main() {  
  
    int a = 5, b = 5, c = 10, result;  
  
    result = (a == b) && (c > b);  
    printf("(a == b) && (c > b) equals to %d\n", result);  
  
    result = (a == b) && (c < b);  
    printf("(a == b) && (c < b) equals to %d\n", result);  
  
    result = (a == b) || (c > b);  
    printf("(a == b) || (c > b) equals to %d\n", result);  
  
    result = (a != b) || (c > b);  
    printf("(a != b) || (c > b) equals to %d\n", result);  
  
    result = !(a != b);  
    printf("!(a != b) equals to %d\n", result);  
  
    result = !(a == b);  
    printf("!(a == b) equals to %d\n", result);  
}
```

```
(a == b) && (c > b) equals to 1  
(a == b) && (c < b) equals to 0  
(a == b) || (c > b) equals to 1  
(a != b) || (c > b) equals to 1  
!(a != b) equals to 1  
!(a == b) equals to 0
```