**C points**

**Basic Data Types and Format Specifiers in C**

**1. Basic Data Types in C**

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Size (in bytes) | Declaration Example | Format Specifier |
| int | 4 bytes | int x = 10; | %d |
| char | 1 byte | char c = 'A'; | %c |
| float | 4 bytes | float f = 3.14; | %f |
| double | 8 bytes | double d = 3.14159; | %lf |
| short | 2 bytes | short s = 100; | %hd |
| long | 4 or 8 bytes | long l = 100000; | %ld |
| long long | 8 bytes | long long ll = 123456; | %lld |
| unsigned int | 4 bytes | unsigned int u = 10; | %u |
| unsigned char | 1 byte | unsigned char uc = 'A'; | %c |
| unsigned long | 4 or 8 bytes | unsigned long ul = 100; | %lu |

Note: For unsigned long, format specifier %lu is used.

**2. Format Specifiers (Full List)**

|  |  |  |
| --- | --- | --- |
| Specifier | Used For | Example |
| %d | int (decimal) | printf("%d", 10); |
| %i | int (same as %d) | printf("%i", 10); |
| %u | unsigned int | printf("%u", 10); |
| %f | float | printf("%f", 3.14); |
| %lf | double | printf("%lf", 3.1415); |
| %c | char | printf('%c', 'A'); |
| %s | string (char array) | printf("%s", "Hello"); |

**Other Concepts**

**3. Pointers:**

A pointer is a variable that stores the memory address of another variable as its value.

Use %p to print is value:

**Example:**

int myAge = 43;     // An int variable  
**int\* ptr = &myAge;**  // A pointer variable, with the name ptr, that stores the address of myAge  
  
// Output the value of myAge (43)  
printf("%d\n", myAge);  
  
// Output the memory address of myAge (0x7ffe5367e044)  
printf("%p\n", &myAge);  
  
// Output the memory address of myAge with the pointer (0x7ffe5367e044)  
printf("%p\n", ptr);

**4. Diff between Initialization , Defination and Decleration :**

**Summary Table:**

| **Term** | **What It Means** | **Example** | **Notes** |
| --- | --- | --- | --- |
| Declaration | Introduce variable/function name | extern int x; | No memory allocated for variables |
| Definition | Allocate memory or provide body | int x; or int func(){} | Must appear once in program |
| Initialization | Assign initial value at definition | int x = 10; | Can only happen during definition |

**Signed and Unsigned Types — Java vs C**

* In java all datatypes are signed to byte having range -128 to 127
* In C datatypes are as both signed and unsigned so
  + Signed range : -128 to 127
  + Unsigned range : 0 to 255

**1. Java Integer Types**

* All integer types (byte, short, int, long) in Java are **signed** by default.
* Java **does not provide unsigned versions** of these types (except for char).
* The char type in Java is **unsigned 16-bit**, representing Unicode characters from 0 to 65,535.
* **Ranges**:

| **Type** | **Bits** | **Range** |
| --- | --- | --- |
| byte | 8 | –128 to 127 |
| short | 16 | –32,768 to 32,767 |
| int | 32 | –2,147,483,648 to 2,147,483,647 |
| long | 64 | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |

* **Unsigned values in Java**:
  + No native unsigned byte, int, or long.
  + You can **simulate unsigned behavior** by using larger types or utility methods:
    - Example: Byte.toUnsignedInt(byteValue) converts signed byte to unsigned int.
  + For example, storing 128 in a byte:

java

CopyEdit

byte b = (byte)128; // Overflow; actual value is -128

int unsignedValue = Byte.toUnsignedInt(b); // unsignedValue = 128

**2. C Integer Types**

* C supports both **signed** and **unsigned** types explicitly.
* Types can be declared as:
  + signed int, unsigned int
  + signed char, unsigned char
  + signed short, unsigned short
  + etc.
* **Ranges** depend on signedness:
  + signed char: –128 to 127
  + unsigned char: 0 to 255
* C char type signedness is implementation-defined:
  + It can be either signed or unsigned depending on the compiler.
  + To be explicit, use signed char or unsigned char.

**Example: Storing 128 in a Byte (8 bits)**

| **Language** | **Type** | **Can store 128?** | **Behavior when assigning 128** |
| --- | --- | --- | --- |
| Java | byte (signed) | No | Overflows to -128 |
| C | signed char | No | Overflows (undefined behavior or implementation-defined) |
| C | unsigned char | Yes | Stores 128 correctly |

**Nan , Null and Undefined : C vs Java vs Js**

**1. NaN (Not a Number)**

| **Language** | **When It Occurs** | **How It Works** | **How to Check** |
| --- | --- | --- | --- |
| **C** | Invalid float operations (e.g., 0.0/0.0) | IEEE 754 special float value | isnan() function |
| **Java** | Invalid float/double operations | Float.NaN or Double.NaN constants | Float.isNaN(), Double.isNaN() |
| **JavaScript** | Invalid numeric operations (e.g., 0/0) | Special numeric value | isNaN() or Number.isNaN() |

**2. Undefined**

| **Language** | **When It Occurs** | **How It Works** | **Notes** |
| --- | --- | --- | --- |
| **C** | *No concept of undefined* | Uninitialized vars have garbage values | No keyword or value |
| **Java** | *No concept of undefined* | Variables must be initialized | Compiler error if used uninitialized |
| **JavaScript** | Variable declared but not assigned | Auto-assigned special undefined value | Distinct primitive value |

**3. Null**

| **Language** | **When It Occurs** | **How It Works** | **Notes** |
| --- | --- | --- | --- |
| **C** | Pointer set to no valid address | NULL macro ((void\*)0) | Used only with pointers |
| **Java** | Reference variable points to nothing | null keyword | Used with object references |
| **JavaScript** | Variable explicitly assigned null | Primitive value representing "no value" | Different from undefined |

**One Complimant using ~**

**Yes, in Java, the ~ operator does give the one's complement of a number.**

**🔁 What ~ does:**

The ~ (bitwise NOT) operator **flips all bits** of a number:

* 0 becomes 1
* 1 becomes 0

This is exactly what **one’s complement** means.

**✅ Example in Java:**

int x = 5; // Binary: 00000000 00000000 00000000 00000101

int result = ~x; // 11111111 11111111 11111111 11111010 (in two's comp = -6)

System.out.println(result); // Output: -6

**Important Note:**

Java uses **two’s complement** to store integers internally. So:

* ~x gives the **one’s complement**
* But the **value printed** is interpreted in **two’s complement form**

So ~5 gives binary ...11111010, which equals -6 in decimal.

**Step-by-Step Breakdown:**

**1. Binary of 5:**

5 in binary (32-bit):

00000000 00000000 00000000 00000101

**2. Apply ~ (bitwise NOT → one’s complement):**

Flip all bits (0 becomes 1, 1 becomes 0):

~5:

11111111 11111111 11111111 11111010

This is the binary result stored in result.

**3. How Java interprets this binary number:**

Java uses **two’s complement** representation for signed integers.

So when you see:

11111111 11111111 11111111 11111010

This is a **negative number** in two’s complement. Let’s convert it to decimal:

**✅ Convert Two's Complement to Decimal:**

To find the decimal of a two’s complement negative binary:

1. Invert the bits (get one’s complement):

00000000 00000000 00000000 00000101

1. Add 1:

+1 = 00000000 00000000 00000000 00000110 (which is 6)

1. Final result: -6

**✅ Therefore:**

int x = 5;

int result = ~x; // flips bits = -6