* **Diff Between endl & \n in c++ :**

1. **\n**
   * معناها "new line" = سطر جديد.
   * وظيفتها إنها **تنزل المؤشر للسطر الجديد فقط**.
   * **ما بتعمل flush (إرسال إجباري) للـ buffer**.

1. **endl**
   * معناها "end line" = نهاية السطر.
   * وظيفتها إنها **تنزل للسطر الجديد + تعمل flush للـ buffer** (يعني تجبر النظام يطبع المحتوى فورًا).

### 🔹 متى تستخدم كل واحد؟

* **استخدم \n**:
  + إذا كنت بتطبع نصوص عادية أو بتعمل حلقات طباعة كثيرة (أسرع من endl   
    لانه مافي flush
  + مثال: طباعة بيانات كبيرة بسرعة.
* **استخدم endl**:
  + إذا كنت **بدك تتأكد إن النص انطبع مباشرة** (مهم في الـ debugging أو عند التعامل مع ملفات/شبكات).
  + مثال: طباعة رسالة خطأ لازم تبين فورًا.
* **استخدم endl + \n** :
  + إذا كان في على فرض 100 جملة بدي اطبعها لو استخدمت \n ==🡺حيكون بطيء
  + إذا كان في على فرض 100 جملة بدي اطبعها لو استخدمت \endl ==🡺حيكون بطيء
  + الحل 🡺 اعمل اول 50 \n ->بعدها اعمل endl -> عشان يعمل fluch buffer

وال 50 الباقيين اعمل نفس الحركة

.

* **Literals and Escape sequences literals**
* **Literals** are fixed values written directly in a program. They represent constant data.
  + **Numeric literals:** 10, 3.14
  + **Character literals:** 'A', '\n'
  + **String literals:** "Hello", "123"
  + **Boolean literals:** true, false
  + **Null literal:** null (in some languages)
* **Escape Sequences** are special codes used inside **character or string literals** to represent characters that cannot be typed directly. They always start with \.
  + Examples:
    - \n → newline
    - \t → tab
    - \" → double quote
    - \\ → backslash
* **Note:** Escape sequences are also considered **literals** (character or string literals written in a special form).

.

* **Variables**

**C++ Data Types**

**│**

**├── 1. Fundamental (Primitive)**

**│ │**

**│ ├── Integral**

**│ │ ├── int**

**│ │ ├── short int**

**│ │ ├── long int**

**│ │ ├── unsigned int**

**│ │ ├── char**

**│ │ └── wchar\_t (wide character)**

**│ │**

**│ ├── Floating**

**│ │ ├── float**

**│ │ └── double**

**│ │**

**│ ├── Boolean**

**│ │ └── bool**

**│ │**

**│ └── Void**

**│ └── void**

**│**

**├── 2. Derived**

**│ ├── Array**

**│ ├── Pointer**

**│ ├── Reference**

**│ └── Function**

**│**

**└── 3. User-defined**

**├── struct**

**├── class**

**├── union**

**├── enum**

**└── typedef / using**

.

* **Datatype sizes**

| **Data Type** | **Size (in bytes)** | **Range (approximate)** |
| --- | --- | --- |
| **bool** | 1 | true / false |
| **char** | 1 | -128 to 127 (signed) 0 to 255 (unsigned) |
| **wchar\_t** | 2 or 4 (depends on compiler) | 0 to 65,535 (if 2 bytes) |
| **char16\_t** | 2 | 0 to 65,535 |
| **char32\_t** | 4 | 0 to 4,294,967,295 |
| **short** | 2 | -32,768 to 32,767 |
| **unsigned short** | 2 | 0 to 65,535 |
| **int** | 4 | -2,147,483,648 to 2,147,483,647 |
| **unsigned int** | 4 | 0 to 4,294,967,295 |
| **long** | 4 (on 32-bit), 8 (on 64-bit systems) | -2,147,483,648 to 2,147,483,647 (if 4 bytes) |
| **unsigned long** | same as long | 0 to 4,294,967,295 (if 4 bytes) |
| **long long** | 8 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| **unsigned long long** | 8 | 0 to 18,446,744,073,709,551,615 |
| **float** | 4 | ~ ±3.4e38 (7 decimal digits precision) |
| **double** | 8 | ~ ±1.7e308 (15 decimal digits precision) |
| **long double** | 8, 12, or 16 (depends on compiler) | even larger range (~18-19 digits precision) |
| **void** | no storage | no value |
| **string** | 12 |  |

* **Type Modifiers**

**We can modify some of the fundamental data types by using type modifiers . There are 4 type modifiers in C++ , they are :**

**1-signed**

**2-unsigned**

**3-short**

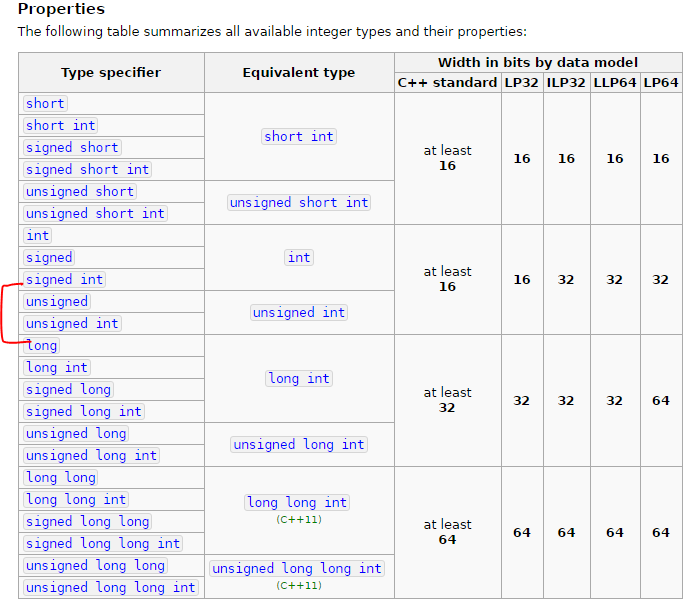
**4-long**

* **Note :** 
  + - **We can only modify the following datatypes with the type modifiers:**

**1 – int**

**2- double**

**3-char**



منطقيا : عمر الشخص 🡺 مش رح يتجاوز 200 سنة  
 فاذا عرفته ب int  
 بتكون تعمل جريمة بحق البشرية

* **Struct & Enums**

**1️⃣ enum (Enumeration)**

* Represents a **set of named constants**.
* Improves **code readability** and **type safety**.
* Values default to integers starting from **0** .

enum Color { green, blue, white, red };

Color myColor = Color::blue;  // myColor has value 1

**2️⃣ struct (Structure)**

* A **user-defined type** that groups **variables** (and functions) under a single name.
* Members are **public by default**.
* Can have **nested structs**, arrays, and member functions.

struct Student {

    string name;

    short age;

    Course course;  // nested struct

    void printFullInfo() {

        cout << name << ", " << age << ", " << course.coursename << "\n";

    }

};

Student s1;

s1.name = "Rahaf";

## **3️⃣** class

* A **user-defined type** similar to struct but with **private members by default**.
* Supports **encapsulation, inheritance, and polymorphism**.
* Can have **constructors, destructors, functions, static members**, etc.

| **Feature** | **enum** | **struct** | **class** |
| --- | --- | --- | --- |
| Members | Named constants | Variables + functions | Variables + functions |
| Default Access | N/A | Public | Private |
| OOP Support | ❌ | Limited | ✅ (full) |
| Use Case | Fixed set of values | Simple data containers | Full OOP design |
| Inheritance | ❌ | Limited (can inherit) | ✅ |
| Constructor | ❌ | ✅ (C++11) | ✅ |

* **Datatype casting**

When you convert a value from one type to another:

* **High → Low** (e.g., double → int) → **Data Loss** (fractional part lost)
* **Low → High** (e.g., int → double) → **No Data Loss**

## **2. How to Cast**

### **a) Implicit Casting**

Automatically done by the compiler.

int x = 10;

double y = x; // int → double (low → high), no data loss

cout << y; // 10.0

### **b) Explicit Casting (C-style)**

double pi = 3.14;

int a = (int)pi; // double → int, fractional part lost

cout << a; // 3

### **c) Explicit Casting (Functional style)**

double pi = 3.14;

int a = int(pi); // same as (int)pi

cout << a; // 3

## **3. String Conversion Functions**

### **a) String → Number**

| **Function** | **Description** | **Example** | **Output** |
| --- | --- | --- | --- |
| stoi | String → int | stoi("123") | 123 |
| stof | String → float | stof("3.14") | 3.14 |
| stod | String → double | stod("3.1415") | 3.1415 |

**Example:**

string s1 = "123";

int x = stoi(s1);

cout << x + 1; // 124

string s2 = "3.14";

float f = stof(s2);

cout << f + 1; // 4.14

string s3 = "3.1415";

double d = stod(s3);

cout << d \* 2; // 6.283

### **b) Number → String**

Use to\_string() to convert numbers to strings:

int n = 42;

double pi = 3.1415;

string strFromInt = to\_string(n); // "42"

string strFromDouble = to\_string(pi); // "3.141500"

cout << strFromInt << endl;

cout << strFromDouble << endl;

**String Methods**

* **Problem : Multi string input 🡪 cin read first one ex :**

int main() {

    string name;

    cout << "Enter your full name: ";

    cin >> name;   // <-- هذا سيأخذ فقط أول كلمة قبل الفراغ

    cout << "You entered: " << name << endl;

    return 0;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

🔹 لو كتبت:

Enter your full name: Rahaf Ahmad

سيطبع:

You entered: Rahaf

* **Solution : الحل استخدام getline(cin , string) --- >لقراءة السطر كامل**
* **Functions and Procedures**
* **Functions =🡺 it’s derived Data type.**
* **Functiion :It was created for reusablity ,and it’s a block of code . ( it return )**
* **Procedure : It’s a function but doesn’t return (void)**
* **Function 🡺**

**Problem :**

getline(cin, p1.FullName);

cin >> p1.Age;

getline(cin, p1.address.Country);

* When you entered Age by cin and then press Enter 🡺 getline() بتعتبر ال => enter هي ادخال فزي كانك ادخلت البلد بالتالي نزل عاللي بعديه لهيك الحل   
  solve**: cin.ignore(1,’\n’) 🡺هي معناها طنشنلي الnewline**
* **Variable scope : دورة حياة المتغير**

**1. Introduction**

In C++, every variable has:

* **Scope** → Where the variable can be accessed.
* **Lifetime** → How long the variable exists in memory.

Understanding these concepts is essential for writing safe and efficient programs.

**2. Types of Scope**

**🔹 a) Local Scope**

* Declared inside a block { }, function, or loop.
* Accessible only inside that block.
* Destroyed when the block ends.

void demo() {

int x = 10; // local variable

cout << x; // valid here

}

// x is destroyed outside this block

**🔹 b) Global Scope**

* Declared **outside all functions**.
* Accessible from any function in the same file.
* Created when the program starts, destroyed when it ends.

int g = 100; // global variable

void show() {

cout << g; // accessible here

}

**🔹 c) Block Scope**

* Variables defined inside if, for, or while are visible only within that block.

for (int i = 0; i < 3; i++) {

cout << i; // i exists only inside loop

}

// i not accessible here

**🔹 d) Function Parameter Scope**

* Parameters act as **local variables** inside the function.

void sum(int a, int b) { // a and b are local

cout << a + b;

}

**3. Lifetime of Variables**

**🔸 Local Variables**

* Created when the function is called.
* Destroyed when the function exits.

void test() {

int x = 5; // exists only during function call

}

**🔸 Static Local Variables**

* Declared with static.
* Created only once, and keep their values between function calls.

void counter() {

static int c = 0; // created once

c++;

cout << c << endl;

}

int main() {

counter(); // 1

counter(); // 2

counter(); // 3

}

**🔸 Global Variables**

* Exist for the entire program’s execution.
* Stored in a special memory section.

**4. Accessing Global Variables**

**✅ Direct Access**

Functions can access global variables directly:

int g = 42;

void print() {

cout << g; // direct access

}

**✅ When Shadowed by Local Variable**

If a local variable has the same name, the **scope resolution operator ::** is used:

int g = 50;

void test() {

int g = 10; // shadows global

cout << g; // prints local

cout << ::g; // prints global

}

* **By Val & By reference**

**1. Function Parameters in C++**

When you pass a variable to a function, C++ offers **two main ways**:

1. **Pass by Value** → a **copy** of the variable is passed.
2. **Pass by Reference** → the function gets **direct access** to the original variable.

**2. Pass by Value**

* The **function receives a copy** of the argument. //استنسخ نسخة وليس الاصلي
* Changes inside the function **do not affect** the original variable.
* Safe, but less efficient for large objects (like struct, class).

**Example:**

#include <iostream>

using namespace std;

void squareByValue(int x) {

x = x \* x; // modifies local copy

cout << "Inside function (by value): " << x << endl;

}

int main() {

int num = 5;

squareByValue(num);

cout << "Outside function: " << num << endl;

return 0;

}

**Output:**

Inside function (by value): 25

Outside function: 5

🔎 Notice that num in main is **unchanged**.

**3. Pass by Reference**

* The function receives a **reference (alias)** & to the original variable.//بعدل على الاصلي نفسه
* Changes inside the function **affect the original variable**.
* Efficient for large objects, but must be used carefully.

**Example:**

#include <iostream>

using namespace std;

void squareByReference(int &x) { // & = reference

x = x \* x; // modifies original variable

cout << "Inside function (by reference): " << x << endl;

}

int main() {

int num = 5;

squareByReference(num);

cout << "Outside function: " << num << endl;

return 0;

}

**Output:**

Inside function (by reference): 25

Outside function: 25

* **Structures & Functions (لتحقيق ال reusability )**