



# Lesson #01 - What is Debugging & Why

## مفهوم كلمة Debug في البرمجة

عند تشغيل أي برنامج تجد أن الأوامر التي كتبتها فيه يتم تنفيذها بسرعة فائقة بدون أي توقف لدرجة أنه قد يتم تنفيذ كل أوامر حتى وإن كان يحتوي على أكثر من ألف أمر في أقل من ثانية واحدة.

عندما يصبح الكود كبيراً و يحتوي على أوامر متداخلة و تبدأ المشاكل بالظهور فيه يصبح إكتشاف الأخطاء البرمجية (Bugs) أو المنطقية (Logical Errors) الموجودة فيه أمر صعب للغاية إذا كنا سنعتمد فقط على النتيجة النهائية التي يعطينا إياها.

لأجل ذلك كان لا بد من إيجاد طريقة تسمح لنا بمراقبة الكود أثناء عمله حتى نعرف ما هو الكود الذي تحدث المشكلة عندما يتم تنفيذه.

في عالم البرمجة كلمة Debugging أو Debug تعني التحكم في تنفيذ أوامر البرنامج بهدف معرفة مكان وجود الأخطاء بالضبط في الكود بهدف تصحيحها.

## Erros in C / C++

Syntax error  
①

Run-time error  
③

Linker error

Logical error  
②

Semantic error

EG

# Lesson #02 - Breakpoint & Memory Values

تعين نقاط التوقف **breakpoint** تسمح نقاط التوقف للمطوريين بإيقاف تنفيذ التعليمات البرمجية مؤقتاً عند نقطة معينة وفحص حالة البرنامج. وهذا مفيد بشكل خاص عند محاولة تحديد السبب الجذري للمشكلة

الـ **breakpoint** هي نقطة يضعها المبرمج عند أحد أسطر البرنامج ، تخبر الـ **debugger** أن يوقف البرنامج عند هذا السطر ويمكن معرفة حالة وقيمة المتغيرات عند هذا الوقت أي قبل تنفيذ الأسطر التالية؛ لها أكثر من فائدة مثلاً معرفة هل تم المرور على هذا الـ **Block** مثل **if** أو **Block** ، أي هل تم تنفيذ ما بداخله أو ما قيمة هذا المتغير في هذا الوقت من البرنامج كذلك معرفة قيم الـ **Array** كاملة وأيضاً معرفة قيمة العدد والرقم الذي توقف عنده.

إذا لم يقف البرنامج يعني أنه لم يمر على هذا السطر إما بسبب شرط **if** أو **loop** أو غير ذلك من الأسباب.

إحدى طرق إضافتها في الكود: توقف عند السطر الذي تريده، زر الفأرة الأيمن، ثم اختيار **insert break-point**

```
12 int main()
13 {
14     int arr1[5] = { 200,100,50,25,30 };
15     int a, b, c;
16
17
18     a = 10;
19     b = 20;
20     a++;
21     ++b;
22 }
```

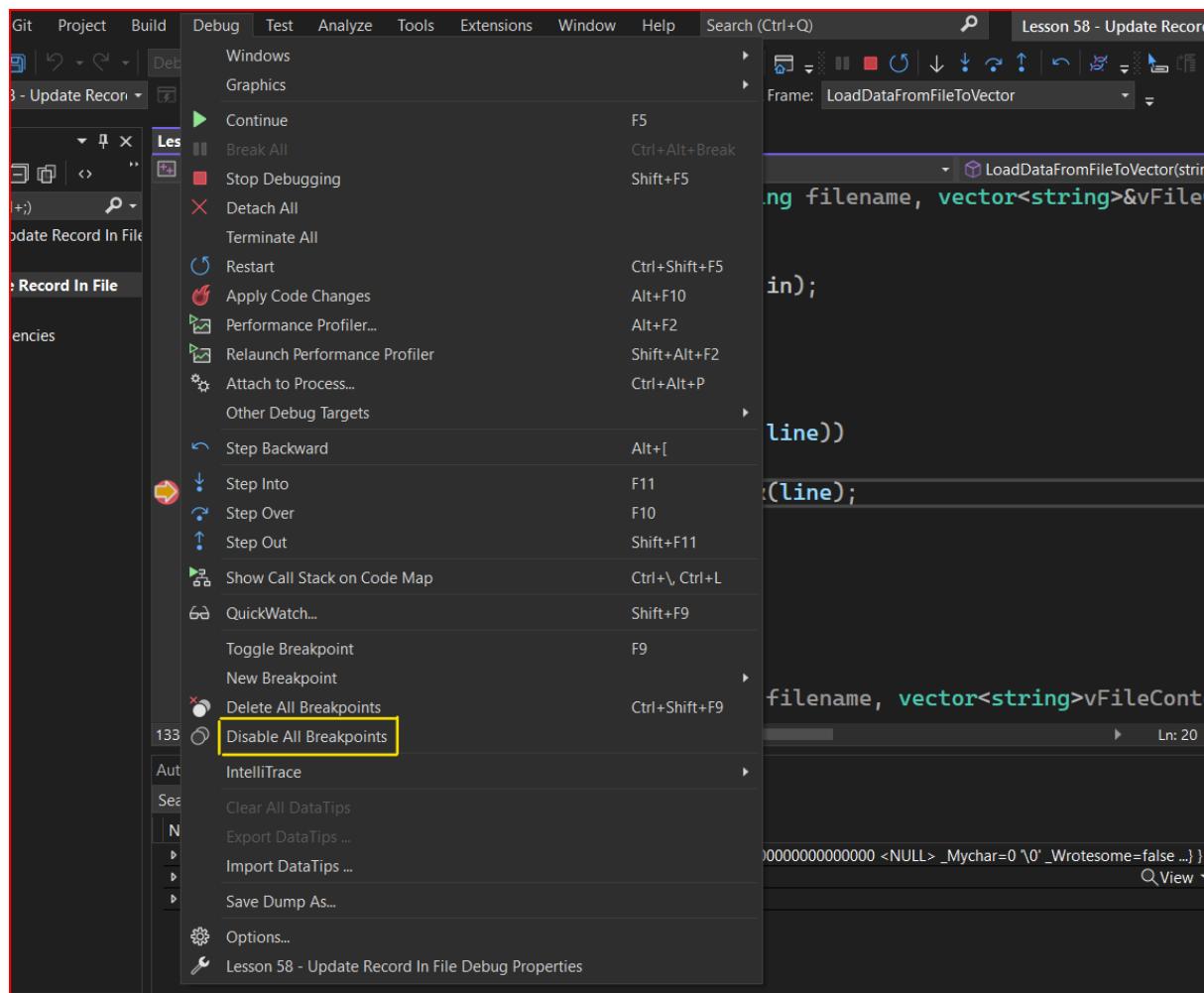
A screenshot of a code editor showing a C program. A red dot on the left margin indicates a breakpoint is set on line 18. The code defines an array arr1 and initializes variables a, b, and c. Lines 18 and 19 show assignments to a and b respectively. Lines 20 and 21 show increment operations on a and b.

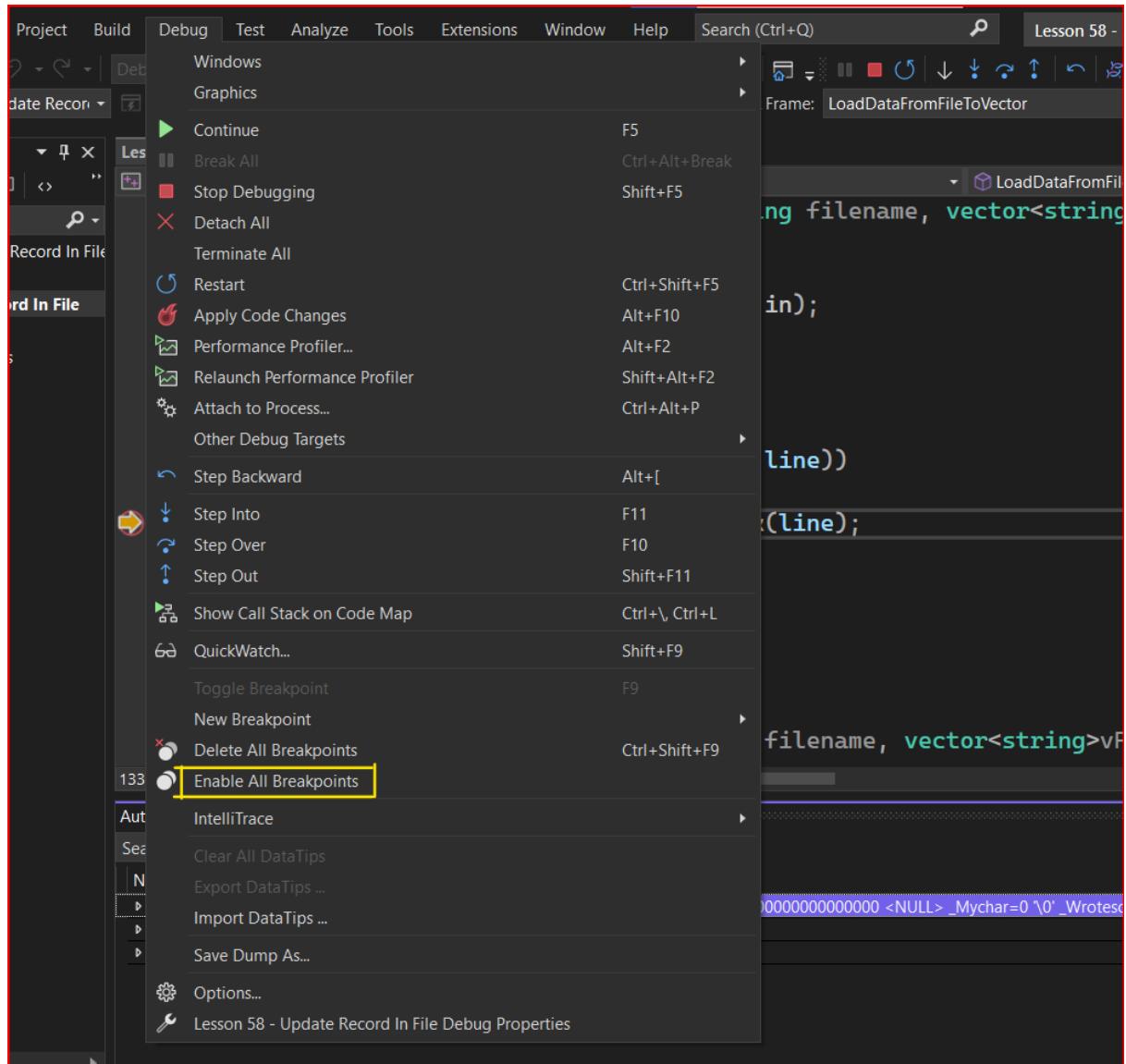


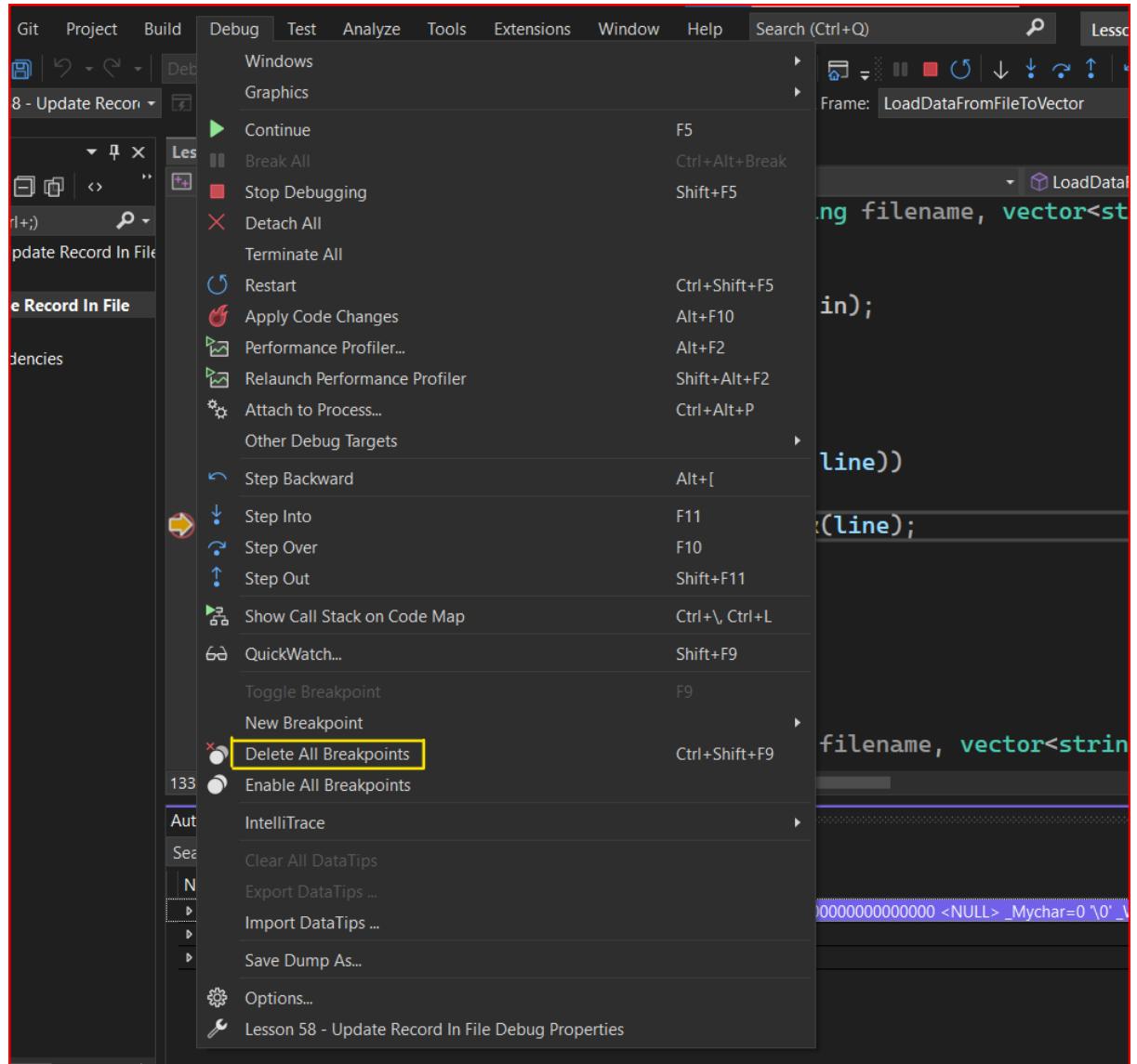
# Lesson #03 - More about /Breakpoints

يمكن تعين اكثر من نقطة توقف.. وعند تخطي نقطة توقف معينة نضغط على **F5 (Run)** لتخطي هذه النقطة.

(تعطيل كافة نقاط التوقف).. يمكن تعطيل كافة نقاط التوقف باستخدام [Disable All Breakpoints](#) هذا الامر من قائمة [Debug](#). وعند تمكينها مرة اخرى نضغط على [Enable All Breakpoints](#) (تمكين كافة نقاط التوقف) من قائمة [Debug](#) مرة اخرى.  
ولحذف جميع نقاط التوقف مرة واحدة نستخدم امر [Delete All Breakpoints](#) من قائمة [Debug](#).







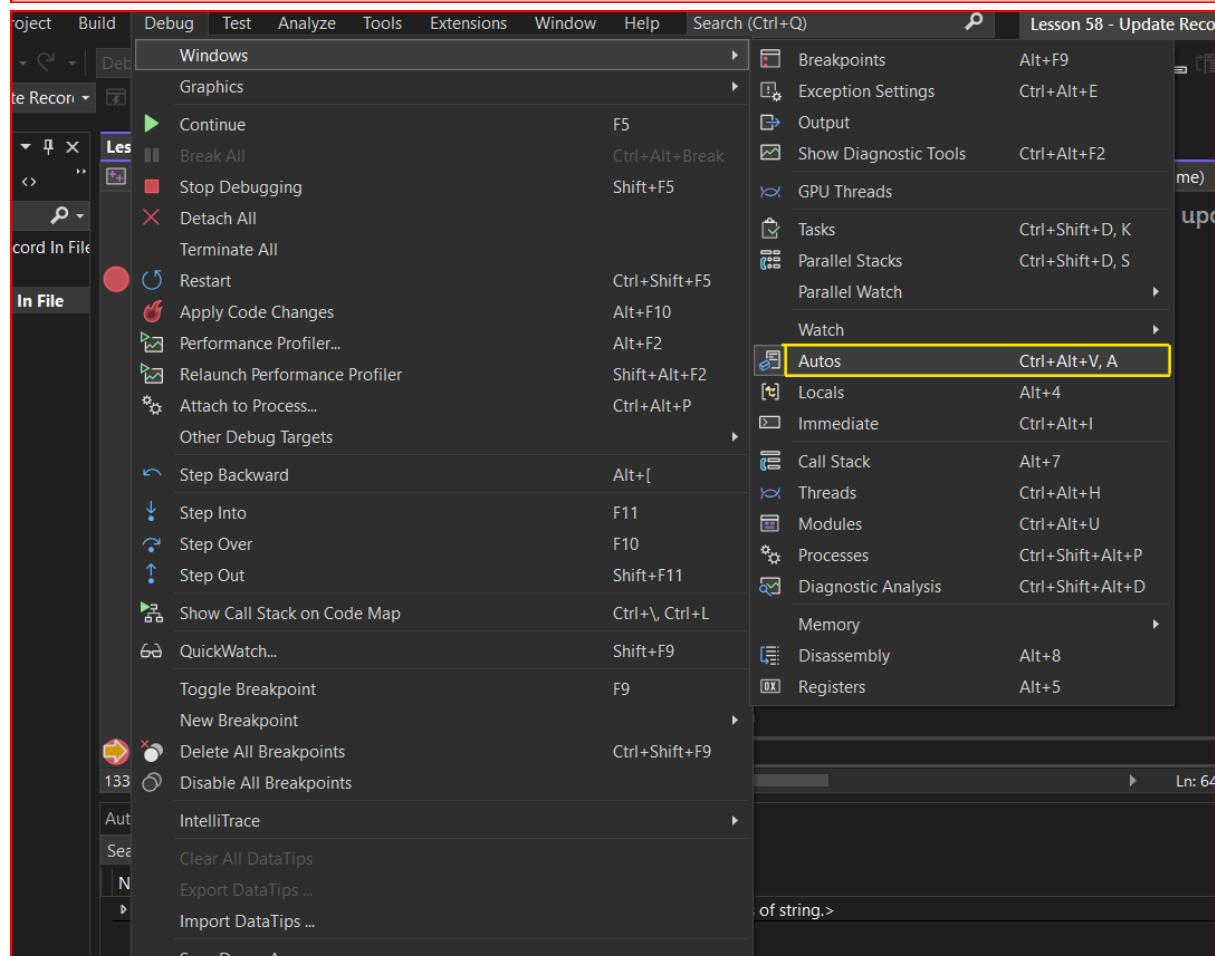


# Lesson #04 - Autos Window

لاظهار المتغيرات وقيمها وتكون عادة اسفل نافذة الكود.

وتظهر عندما يكون هناك Breakpoints (نقط توقف) عند عمل تشغيل (Run) للكود.

وتكون في قائمة Debug ثم اول عنصر في القائمة Windows ومن هذه القائمة نذهب الى Autos



The screenshot shows the Visual Studio IDE with the code editor displaying a C++ file named 'Lesson 04 - ...os window.cpp'. The code contains a main function with variable declarations and assignments. Below the code editor is the 'Autos' window, which is also highlighted with a yellow box. The Autos window displays the current values of variables 'a', 'arr', 'b', and 'c'. The variable 'arr' is shown as an array of integers with memory address 0x0000007153cfae8 and elements (200, 100, 50, 25, 30). The other variables have their values set to -858993460.

Name	Value	Type
a	-858993460	int
arr	0x0000007153cfae8 (200, 100, 50, 25, 30)	int[5]
b	-858993460	int
c	-858993460	int



# Lesson #05 - Quick Watch Window

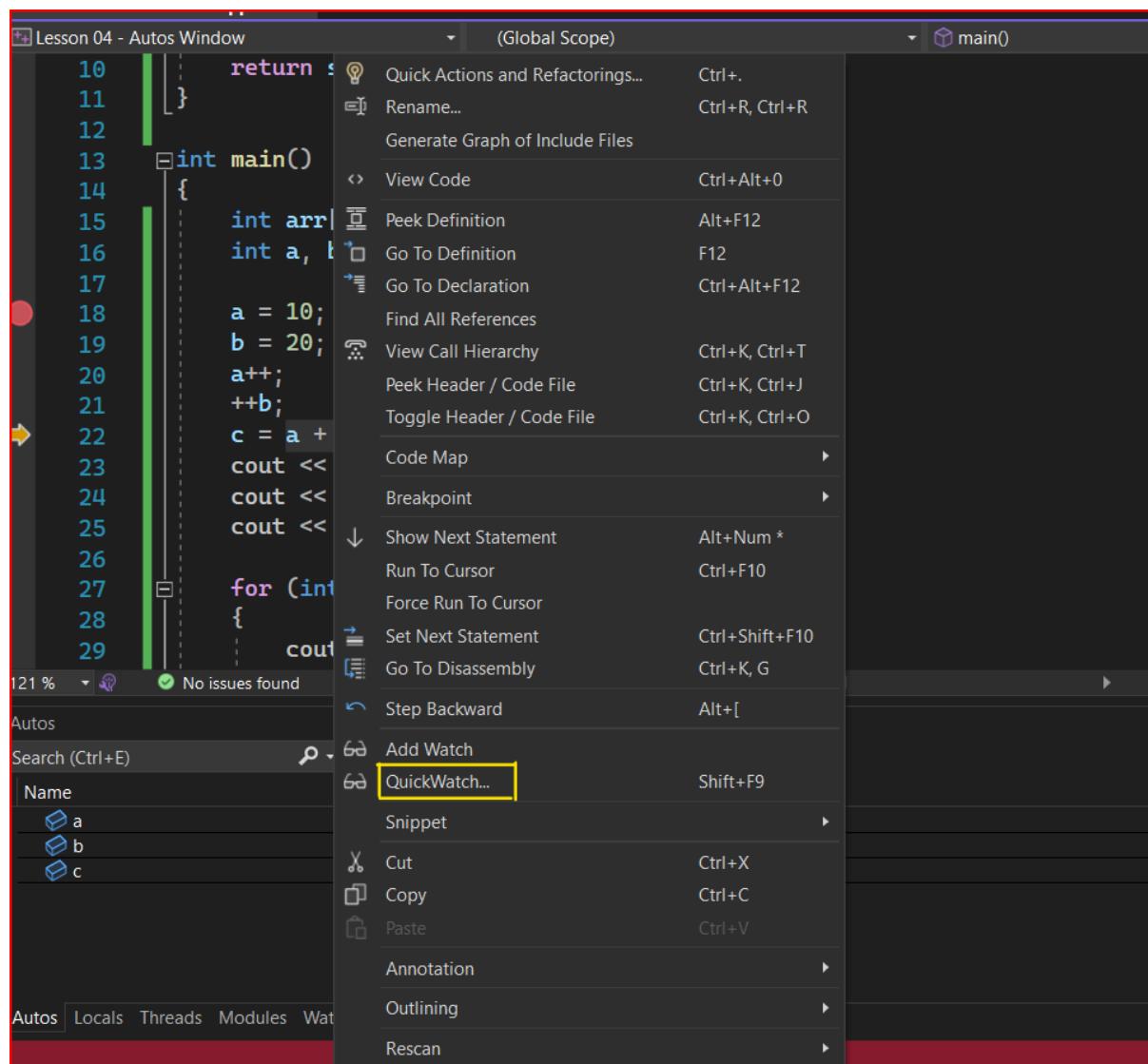
نراقب به كل ما يحدث في الكود

Lesson 04 - Autos Window (Global Scope)

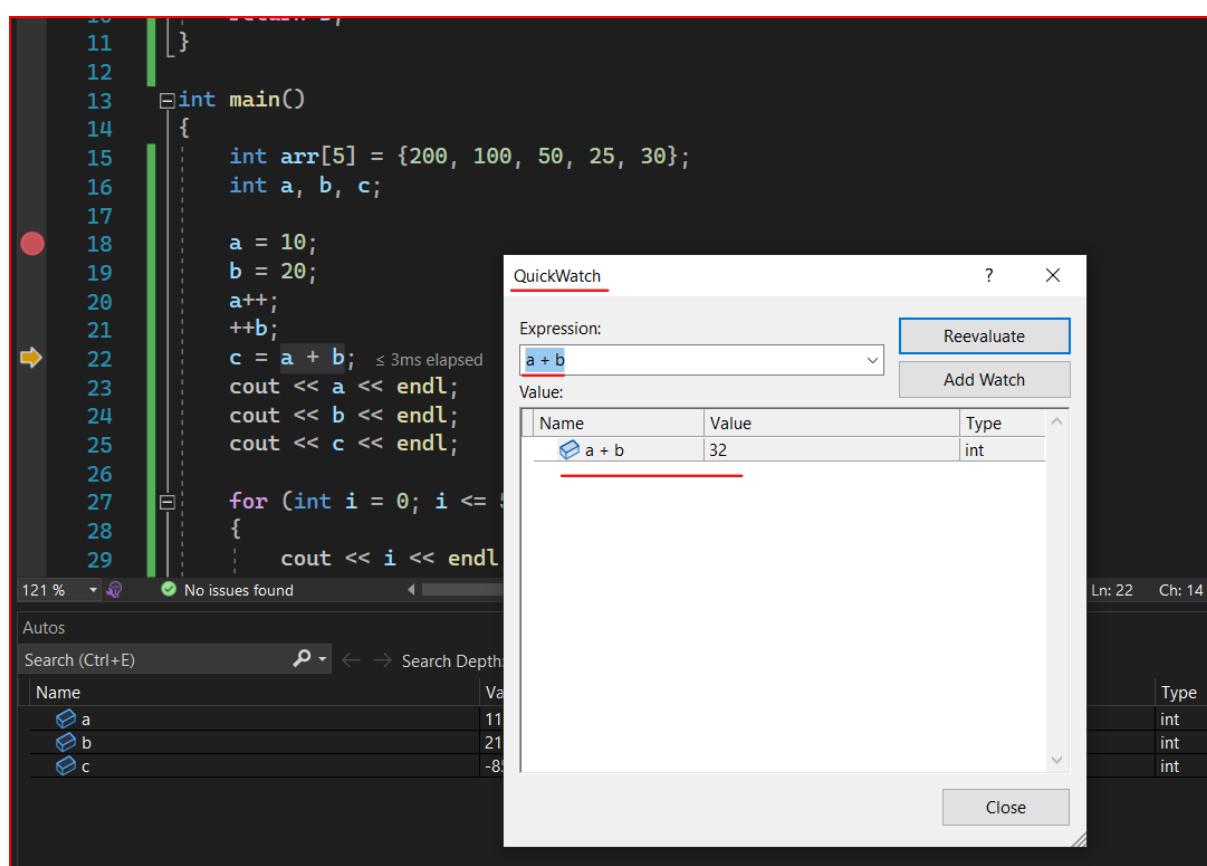
```
10     return s;
11 }
12
13 int main()
14 {
15     int arr[5] = {200, 100, 50, 25, 30};
16     int a, b, c;
17
18     a = 10;
19     b = 20;
20     a++;
21     ++b;
22     ▶| c = a + b; ≤ 3ms elapsed
23     cout << a << endl;
24     cout << b << endl;
25     cout << c << endl;
26
27     for (int i = 0; i <= 5; i++)
28     {
29         cout << i << endl;
```

Lesson 04 - Autos Window (Global Scope)

```
10     return s;
11 }
12
13 int main()
14 {
15     int arr[5] = {200, 100, 50, 25, 30};
16     int a, b, c;
17
18     a = 10;
19     b = 20;
20     a++;
21     ++b;
22     ▶| c = a + b; ≤ 3ms elapsed
23     cout << a << endl;
24     cout << b << endl;
25     cout << c << endl;
26
27     for (int i = 0; i <= 5; i++)
28     {
29         cout << i << endl;
```



## نراقب قيمة متغير



# نراقب قيمة متغير

The screenshot shows a C++ code editor with a breakpoint at line 18. The code defines an array `arr` and calculates the sum of its elements. A Quick Watch window is open, showing the expression `arr` with a value of `0x00000071533cfae8` and type `int[5]`. The array elements are listed as `[0]: 200`, `[1]: 100`, `[2]: 50`, `[3]: 25`, and `[4]: 30`.

```
10     return s;
11 }
12
13 int main()
14 {
15     int arr[5] = {200, 100, 50, 25, 30};
16     int a, b, c;
17
18     a = 10;
19     b = 20;
20     a++;
21     ++b;
22     c = a + b; // 3ms elapsed
23     cout << a << endl;
24     cout << b << endl;
25     cout << c << endl;
26
27     for (int i = 0; i <= 5; i++)
28     {
29         cout << i << endl;
30     }
31
32     cout << a + b << endl;
33     cout << b << endl;
34     cout << c << endl;
35
36     for (int i = 0; i <= 5; i++)
37     {
38         cout << i << endl;
39         a = a + a * i;
40     }
41
42     c = MySum(a, b); // 3ms elapsed
43     cout << c << endl;
44
45     return 0;
46 }
```

QuickWatch  
Expression: arr  
Value:  
Name Value Type  
arr 0x00000071533cfae8 [200, 100, ... int[5]  
[0] 200 int  
[1] 100 int  
[2] 50 int  
[3] 25 int  
[4] 30 int

Watch 1  
Search (Ctrl+E) ⚡ ← → Search Depth: 3

# نراقب قيمة المعلمات

The screenshot shows a C++ code editor with a breakpoint at line 32. The code includes a user-defined function `MySum`. A Quick Watch window is open, showing the expression `MySum(a, b)` with a value of `7941` and type `int`.

```
22     cout << a + b << endl;
23     cout << b << endl;
24     cout << c << endl;
25
26     for (int i = 0; i <= 5; i++)
27     {
28         cout << i << endl;
29         a = a + a * i;
30     }
31
32     c = MySum(a, b); // 3ms elapsed
33     cout << c << endl;
34
35     return 0;
36 }
```

QuickWatch  
Expression: MySum(a, b)  
Value:  
Name Value Type  
MySum(a, b) 7941 int

Watch 1  
Search (Ctrl+E) ⚡ ← → Search Depth: 3



# Lesson #06 - Changing Values in Debugging Mode

## Changeing Values in Debugging Mode

تغيير القيم أثناء عملية الديبجنج

The screenshot shows a debugger interface with the following details:

**Code Editor:** Shows the C++ code for `main()`. The variable `b` is highlighted with a yellow border and has its value set to 55. The expression `a + b` is also highlighted.

```
16     int a, b;
17
18     a = 10;
19     b = 20;
20     a + b // Value: 55
21     ++b;
22     c = a + b;
23     cout << a << endl;
24     cout << b << endl;
25     cout << c << endl;
26
27     for (int i = 0; i <= 5; i++)
28     {
29         cout << i << endl;
30         a = a + a * i;
31     }
32     c = MySum(a, b);
33     cout << c << endl;
34
35     return 0;
36
```

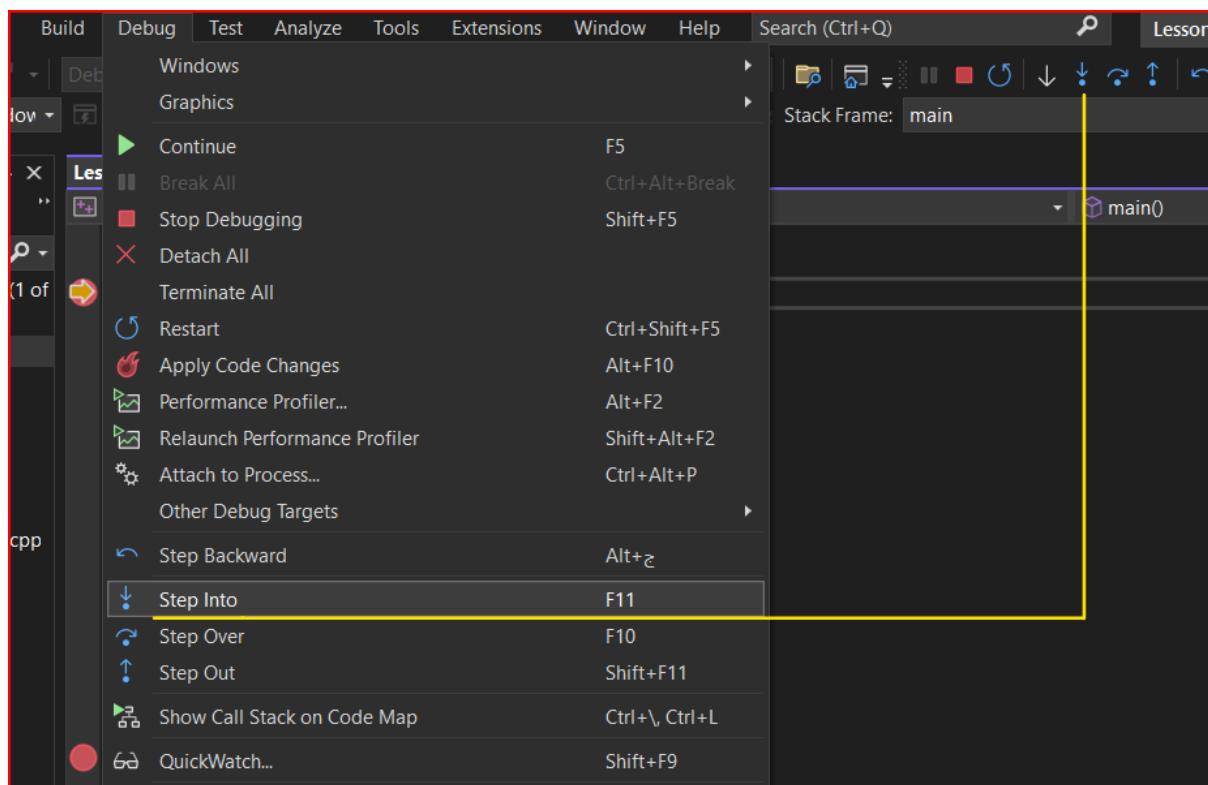
**Watch Window:** Shows the current values of variables and expressions.

Name	Type
<code>a + b</code>	int
<code>arr</code>	int[5]
<code>MySum(a, b)</code>	This expression has side effects and will not be evaluated.

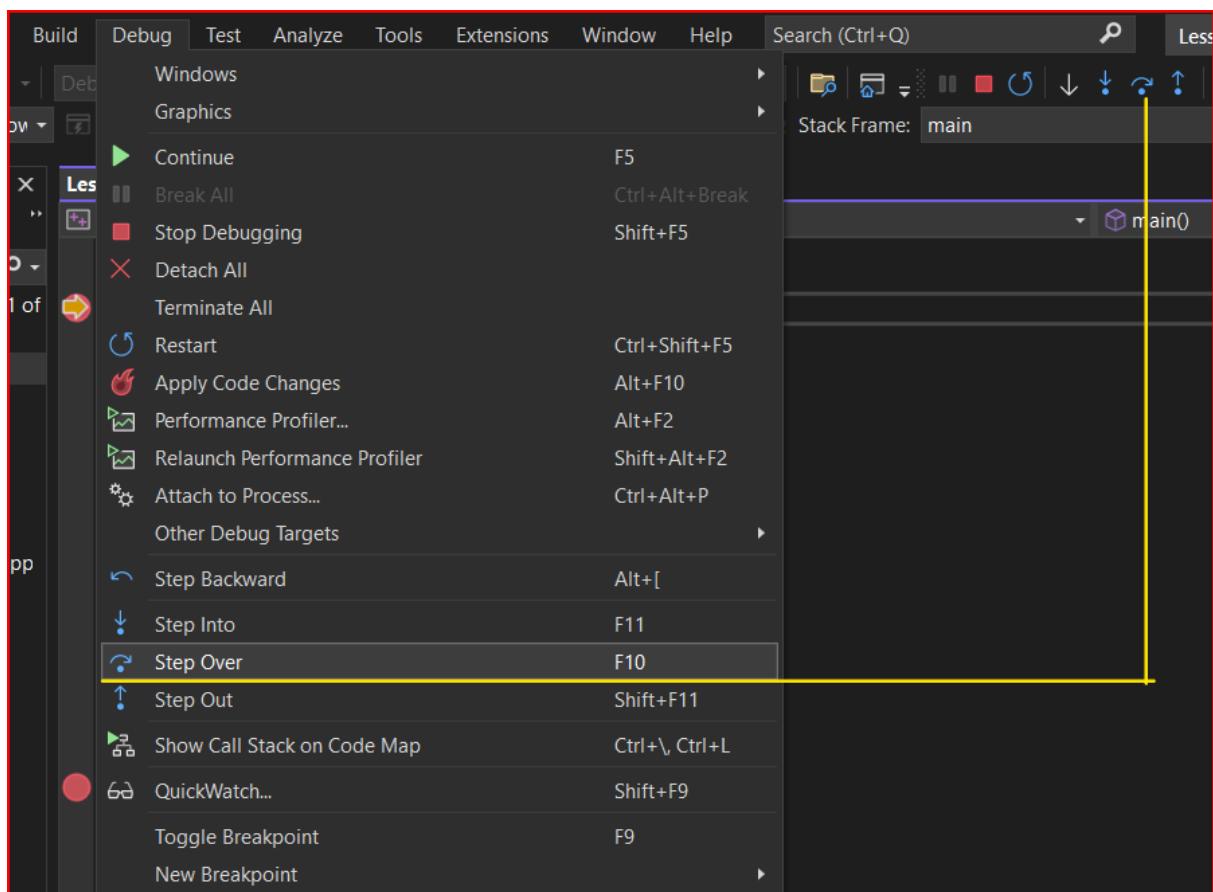


# Lesson #07 - Step Into/Over/Out

## Step Into خطوة خطوة

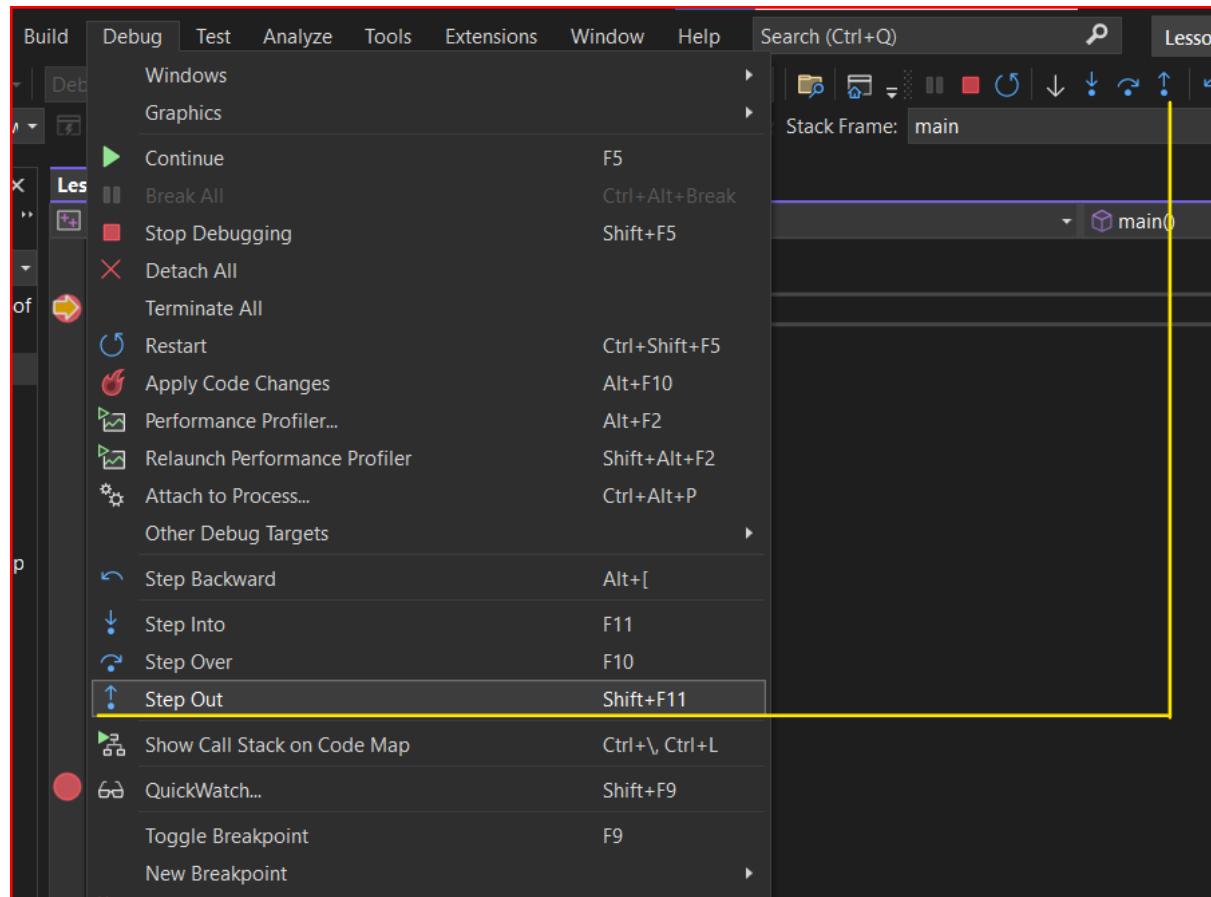


## Step Over تخطي السطر التالي



## Step Out

## اخْرَجْ مِنْ السُّطُرِ الْتَالِيِّ





# Lesson #08 - Library: Create Your Own Library

## مفهوم المكتبات البرمجية

المكتبة (Library) عبارة عن ملف يمكن أن يحتوي على كلاسات و دوال جاهزة بمجرد تضمينها في المشروع تصبح قادرة على استخدام كل ما هو موجود فيها و كأنها جزء من المشروع. في العادة عندما يوجد المطور نفسه دائمًا ما يستخدم نفس الكلاسات والدوال في مشاريعه يقوم بوضعهم داخل مكتبة واحدة وكلما احتاج إليها يقوم بتضمينها في مشروعه.

الآن، عليك معرفة أنه يوجد نوعين من المكتبات التي يمكن تضمينها في المشروع:

- مكتبات ثابتة (Static Libraries) و يكون امتدادها .a. على نظام ويندوز و .lib. على نظام لينكس و ماك.
- مكتبات ديناميكية (Dynamic Libraries) و يكون امتدادها .dll. على نظام ويندوز و .so. على نظام لينكس و ماك.

المكتبة التي تنشئها بنفسك أو التي تقوم بتحميلها من النت، يمكنك وضعها في أي مكان تريده على حاسوبك ونقصد بذلك أنك لست مجرر على وضع المكتبات في مكان محدد حتى يسمح لك باستخدامها. لاحقًا عند الحاجة لاستخدام المكتبة في أي مشروع، يجب إعلام المترجم بمكان وجودها حتى تصبح قادرة على تضمين أي ملف موجود فيها والبدء باستخدام الكود الموجود فيه.

**الأفضل والأسهل لك دائمًا هو أن تخصص مكان في حاسوبك لوضع فيه كل المكتبات التي قد تستخدمها.**

## الفرق بين المكتبات الثابتة والمكتبات الديناميكية

من ناحية التعامل مع الكود الموجود في المكتبات سواء كنت تستخدم مكتبة ثابتة أو مكتبة ديناميكية فإنه لا يوجد أي اختلاف من هذه الناحية.

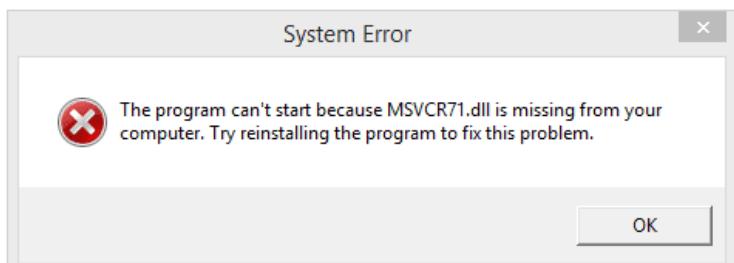
**الاختلاف الأساسي بين النوعين** هو أن المكتبات الثابتة يتم دمج الكود الموجودة فيها مع كود المشروع نفسه حين تقام ببناء المشروع. في حين أن المكتبات الديناميكية **تظل موضوعة بشكل منفصل عن كود المشروع** حين تقام ببنائه.

في كل مرة تقوم فيها بإجراء تعديل على كود مكتبة ثابتة تستعملها في مشروعك، يجب إعادة بناء كود المكتبة و من ثم كود المشروع الذي يستعملها لجعل المشروع يستعمل آخر نسخة تم تحديثها من منها.

أما في حال إجراء تعديل على كود مكتبة ديناميكية في مشروعك، يجب إعادة بناء كود المكتبة فقط بدون الحاجة لإعادة بناء كود المشروع.

## معلومات تقنية:

أحياناً عندما تحاول تشغيل برنامج ما على حاسوبك يظهر لك خطأ يخبرك بأنه يوجد ملف .dll. ناقص يحتاجه البرنامج حتى يشتغل كالتالي:



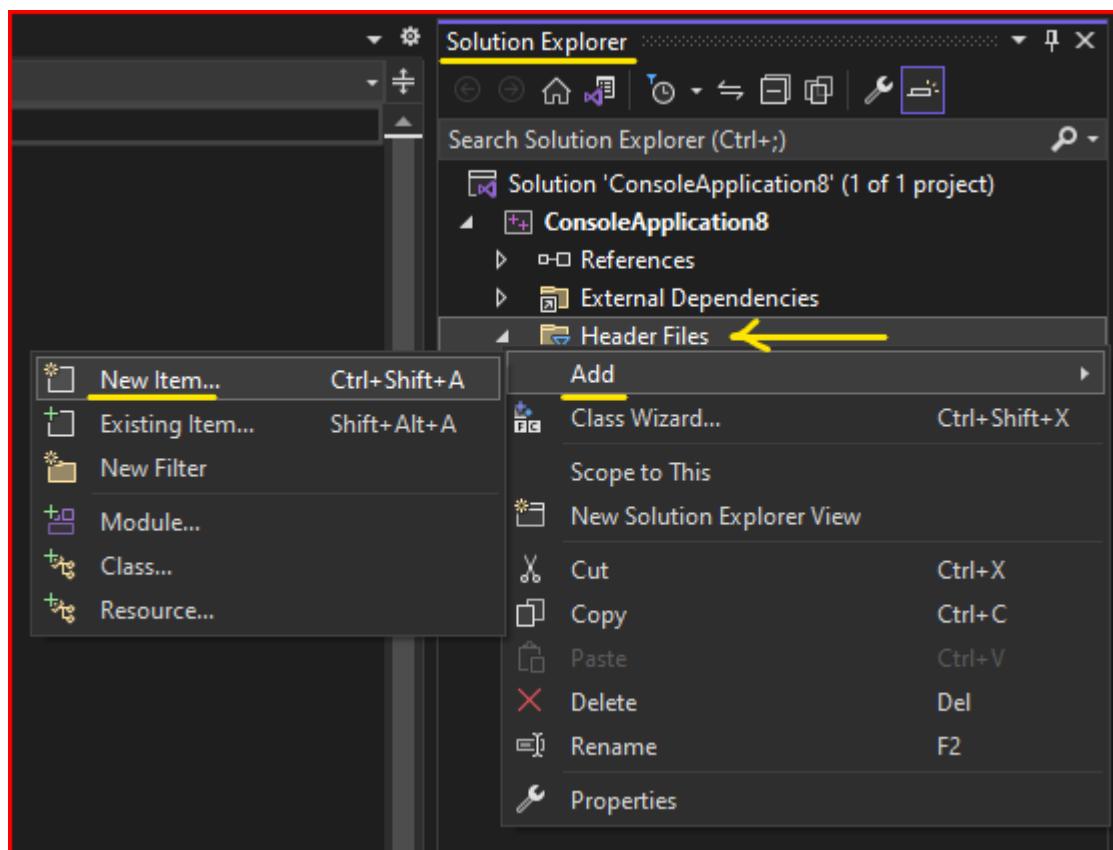
هذا الخطأ معناه بكل بساطة أن البرنامج الذي تستعمله في الأساس يستعمل المكتبة الديناميكية **MSVCR71.dll** التي تم ذكر إسمها والتي لم يستطع إيجادها.

لحل هذه المشكلة في العادة تقوم بالبحث في النت عن إسم الملف المذكور (أي المكتبة) الذي يحتاجه التطبيق حتى يعمل.

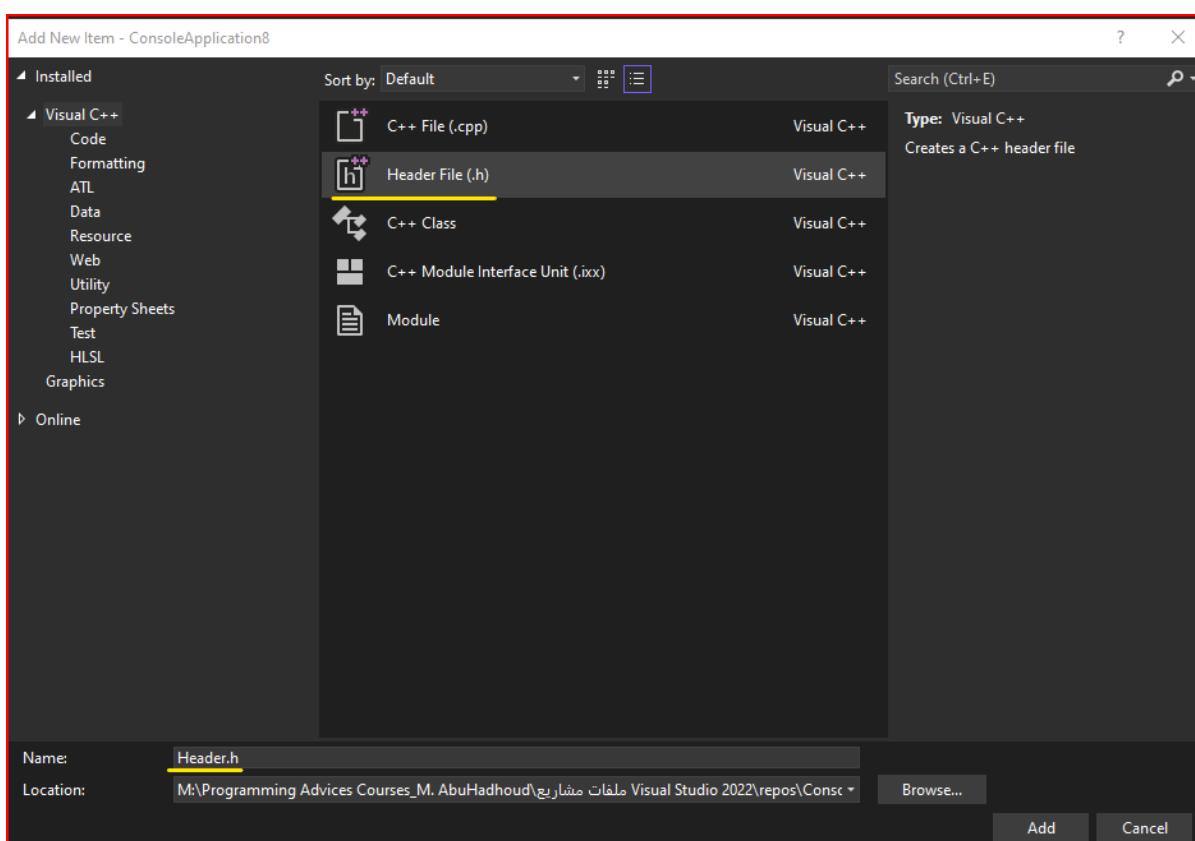
في حالتنا نبحث عن الملف **MSVCR71.dll** وبعد إيجاده نقوم بإضافته في ضمن ملفات البرنامج فقط.

## لعمل مكتبة داخل ال Project

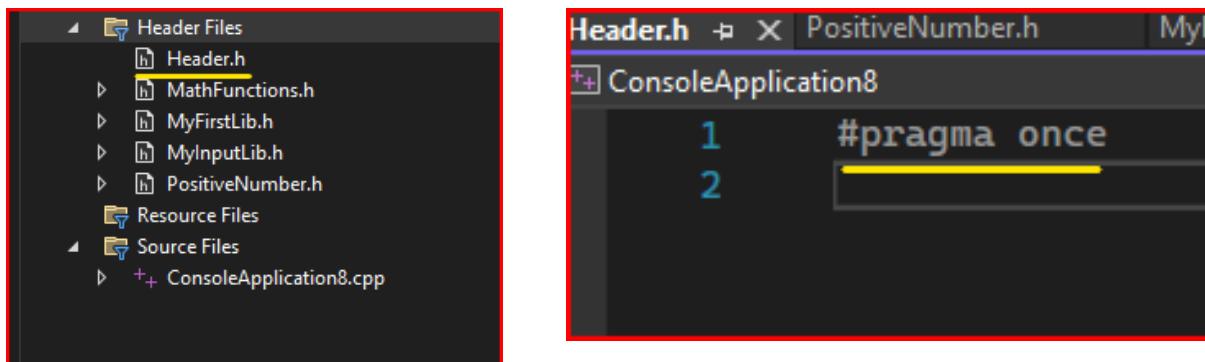
من داخل ال Project نذهب الى Solution Explorer ومنها للفolder اسمه Header File نؤشر عليه بالماوس ثم كليك يمين ومن القائمة المنسدلة نأشر على Add ومنها نضغط على New Item ثم تفتح نافذة... كالصورة التالية:



من هذه النافذة Add New Item نختار Header File(.h) وعند ال Name نضع اسم للمكتبة متبوع بـ .h. وبعد ذلك ... Add كالصورة التالية:



سيظهر بعد ذلك اسم المكتبة اسفل تبويب **Header File** نضغط عليه مرتين يظهر لنا ملف فارغ إلا من سطر في الأعلى به **#pragma once** (وهذا يسمى **Compiler Directive**) وظيفته ان لا يعمل **reloaded** ([إعادة تحميل]) لهذه المكتبة في كل مرة **لذا** نترك السطر كما هو موجود.



في هذه المكتبة يمكن ان نضع **Procedure** و **Function** داخل **Category** باسم **namespace** يتبعها الاسم الذي تريده لها... **كالمثال التالي:-**

هنا عملنا مكتبة اسمها **Header.h** ثم قمنا بعمل **Category** باسم **MyLib8** جمعنا تحته بروسيدجر (**void PrintText()**) وفانكشن (**int SumNumbers(a, b, c)**)

The screenshot shows the code editor with 'ConsoleApplication8' selected. The code is as follows:

```

1 #pragma once
2 using namespace std;
3
4 namespace MyLib8
5 {
6     void PrintText()
7     {
8         cout << "\nThis is my first local library" << endl;
9     }
10
11     int SumNumbers(int a, int b, int c)
12     {
13         int Sum = 0;
14         return Sum = a + b + c;
15     }
16 }
17
18

```

The code includes a '#pragma once' directive, a 'using namespace std;' statement, and a 'namespace MyLib8' block. Inside this block are two functions: 'PrintText()' and 'SumNumbers(int a, int b, int c)'. The 'PrintText()' function outputs a message to the console. The 'SumNumbers()' function takes three integers as parameters and returns their sum.

```

Output ..... Show output from: Build | ⌂ | ⌄ | ⌅ | ⌆ | ⌇ | ⌈ | ⌉
1>E:\Courses\Programming Advices Courses_M. AbuHadhoud\Visual Studio 2022\repos\ConsoleApplication8\repos\ConsoleApplication8\Visual Studio 2022\repos\ConsoleApplication8\repos\ConsoleApplication8.vcxproj -> E:\Courses\Programming Advices Courses_M. AbuHadhoud\Visual Studio 2022\repos\ConsoleApplication8\repos\ConsoleApplication8.vcxproj
1>Done building project "ConsoleApplication8.vcxproj".
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====

```

(Ctrl + B) Build : المقصود من بناء المشروع هو قيام المترجم بالتأكد من أن كود المترجم لا يوجد فيه أي مشكلة من ناحية كتابة الأوامر (Syntax Error).

يمكن ان نعمل مكتبة تجمع عدة فانكشن او عدة بروسيدر مثل مكتبة MyInputLibrary نجمع بها العديد من الفانكشنز او البروسيدر جرز ( readpositivenumbers() و readarray() و readfloatnumbers() و readnumberinrange() ) وليس هناك ما يمنع ان يكون هناك اكثرا من مكتبة

نقوم باستدعاء المكتبة في الملف الرئيسي بـ #include "LibraryName.h" لانها مكتبة محلية (Local Library)

```

ConsoleApplication8 (Global Scope)
1 #include <iostream>
2 #include "Header.h";
3
4 using namespace std;
5
6 int main()
7 {
8     MyLib8::PrintText();
9
10    cout << "\nSum of numbers : " << MyLib8::SumNumbers(10, 20, 30) << endl;
11
12 }
13
14
15
16
17

```

اذا احتجنا Function من مكتبة اخرى نعمل لها include داخل المكتبة الحالية  
ممكن ان ننادي الفانكشن مباشرة من غير ما نكتب ( MyLib8::PrintTxt(); ) وذلك باستخدام using namespace MyLib8؛  
الهيدر ، لكن هذا لا يفضل لانه يمكن ان يحدث مشاكل في اسماء الفانكشنز

```

#include <iostream>

#include "Header.h";

using namespace std;

using namespace MyLib8;

int main()
{
    PrintText();

    cout << "\nSum of numbers : " << SumNumbers(10, 20, 30) << endl;
}

```



# Lesson #09 - Ternary Operator:

## Short Hand IF

There is also a **short-hand if else**, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements.

وهو الذي يُعرف باسم **العامل الثلاثي** لأنه يتكون من ثلاثة معاملات. يمكن استخدامه لاستبدال عدة أسطر من التعليمات البرمجية بسطر واحد. غالباً ما يتم استخدامه لاستبدال عبارات if البسيطة.

### Syntax

```
variable = (condition) ? expression1 : expression2;
```

True  
↓

False  
↓

### Using normal IF

```
int Mark = 90;
string result;

if (Mark >= 50)
{
    result = "PASS" ;
}

else
{
    result = "FAIL" ;
}

cout << result << endl;
```

### Short Hand IF

```
int Mark = 90;
string result;

result = (Mark >= 50) ? "PASS" : "FAIL";
cout << result << endl;
```

```
//simple if else statements.
```

```
int time = 20;
if (time < 18)
{
    cout << "Good day.";
}
else
{
    cout << "Good evening.";
}
```

```
// short-hand if else
```

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
cout << result;
```

- لا يمنع ان يكون هناك **Expressions** بدل الـ **Functions**

- والشرط يمكن ان يحتوي على **Or** او **And**

```
string result = (time < 18 && time == 18 || time == 20) ? Function1 : Function2;
```

- ويمكن ان تحتوي على العديد من الـ **expressions** (الشروط) وأيضاً من **conditions** (التعابيرات، المصطلحات)

```
((Number == 0) ? cout << "\nThis number is Zero\n" : ((Number > 0) ? cout
<< "\nThis number is positive\n" : cout << "\nThis number is negative\n"));
```

```
string CheckNumber()
```

```
{
```

```
    string Result;
```

```
    int Number;
```

```
    cout << "Please enter a number: ";
```

```
    cin >> Number;
```

```
    (Number == 0) ? cout << "\nThis number is Zero\n" :
```

```
    ((Number > 0) ? cout << "\nThis number is positive\n" : cout
    << "\nThis number is negative\n");
```

```
    return Result;
```

```
}
```



# Lesson #10 - Ranged Loop

## Ranged Loop

This **for loop** is specifically used with **collections** such as **arrays** and **vectors**.  
تُستخدم حلقة **for** هذه خصيصاً مع **مجموعات** مثل **المصفوفات** و **المتجهات**.

## Syntax

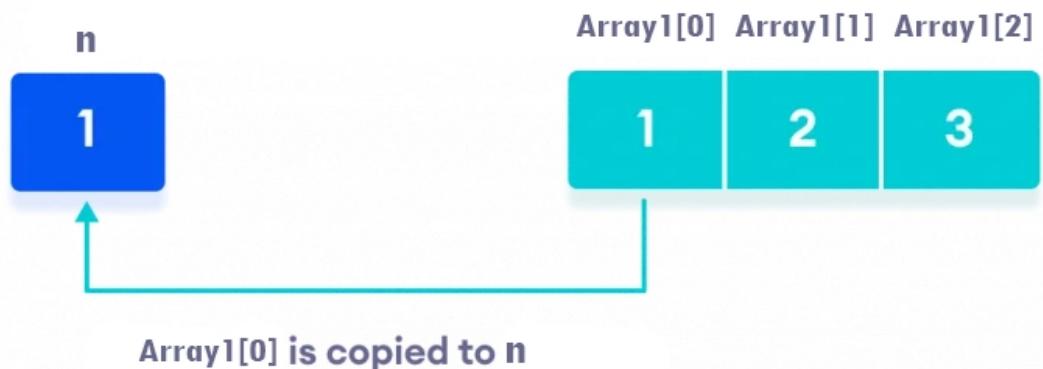
```
for (      int n      :      Array1      )  
for (rangeDeclaration : rangeExpression)  
{  
// code  
}
```

## Example:

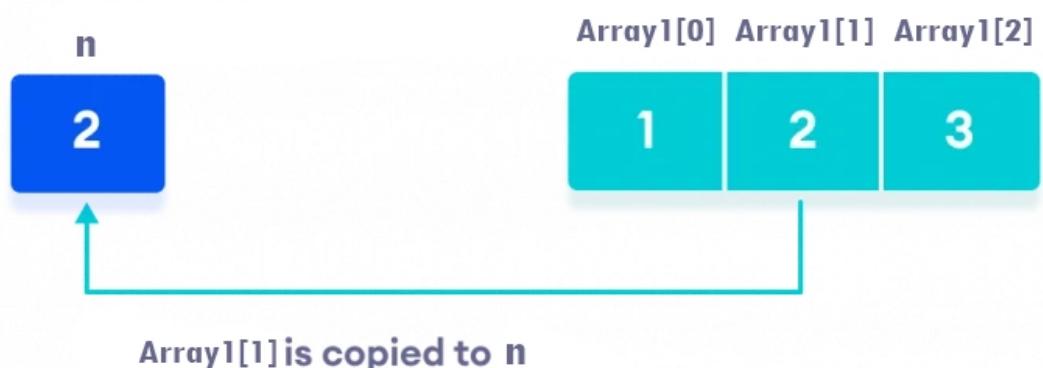
```
int Array1[] = { 1, 2, 3, 4 };  
  
for (int n : Array1)  
{  
  
    cout << n << endl;  
  
}
```

يجب العنصر الاول ويحطها في **n** ويطبعها  
وبعدين العنصر اللي بعده ويحطها في **n** وهكذا لعد الانتهاء من اخر عنصر

## 1st Iteration



## 2nd Iteration



## 3rd Iteration



Here, the ranged for loop iterates the array **Array1** from beginning to end. The int variable **n** stores the value of the array element in each iteration.

هنا، تقوم **ranged for loop** بتكرار المصفوفة **Array1** من البداية إلى النهاية. يقوم المتغير **int n** بتخزين قيمة عنصر المصفوفة في كل تكرار.

However, it's better to write the ranged based for loop like this:

ومع ذلك، فمن الأفضل كتابة النطاق بناءً على الحلقة مثل هذا:

```
// access memory location of elements of num  
for (int &n : Array1)  
{  
    // code  
}
```

Notice the use of **&** before **n** Here,

لاحظ استخدام **&** قبل **n** هنا،

**int n : Array1** - Copies each element of **Array1** to the **n** variable in each iteration.

This is not good for computer memory.

ينسخ كل عنصر من **Array1** إلى المتغير **n** في كل تكرار. **هذا ليس جيداً لذاكرة الكمبيوتر**.

**int &n : Array1** - Does not copy each element of **Array1** to **n**. Instead, accesses the elements of **Array1** directly from **Array1** itself. This is more efficient.

لا يتم نسخ كل عنصر من **Array1** إلى **n** بدلاً من ذلك، يصل إلى عناصر **Array1** مباشرةً من **Array1** نفسه. **هذا أكثر كفاءة**.

**Ranged Loop** : تستخدم للكوكلشن مثل الاري والفاكتور والأوبجكت

**Array is** : collection of items

**Ranged loop** : For: ليست بديل للفور لوب العادي

```
int main()
{
    // Iterating over array
    int array1[] = { 1, 2, 3, 4, 5 };

    for (int& n : array1)
    {
        cout << n << ' ';
    }

    cout << "\n\n";

    // the initializer may be a braced-init-list
    for (int& n : { 1, 2, 3, 4, 5 })
    {
        cout << n << ' ';
    }

    cout << "\n\n";

    // Just running a loop for every array
    // element
    for (int& n : array1)
    {
        cout << "In loop" << ' ';
    }

    cout << "\n\n";

    // Printing string characters
    string str = "Ahmad ElSayed";
    for (char& c : str)
    {
        cout << c << ' ';
    }

    cout << "\n\n";
}
```



# Lesson #11 - Validate Number

تحقق : التحقق من ان قيم المدخلات يجب ان تكون رقم فقط  
**while(cin.fail())**: يكتشف ما إذا كانت القيمة المدخلة تناسب القيمة المحددة في المتغير وهذا من خلال لooop

فإذا كان **(cin.fail() صحيحاً (true)**, فهذا يعني:

أ) القيمة المدخلة لا تناسب المتغير

ب) لن يتأثر المتغير

ج) لا يزال الدفق مكسوراً

د) القيمة المدخلة لا تزال في المخزن المؤقت وسيتم استخدامها لعبارة "cin >>variable" التالية.

ومن ثم عليك القيام بما يلى:

أ) إصلاح البث (مسح الخطأ) عبر **(cin.clear()**)

ب) امسح المخزن المؤقت باستخدام: **cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');**  
(يلغي حجم المدخلات لعند لما يجد ادخال سطر جديد)(طشت باقي الـ Input كله)

ج) ثم نطلب من المستخدم ادخال رقم مرة اخرى باستخدام رسالة للادخال بـ **cout** وبعدها **cin**

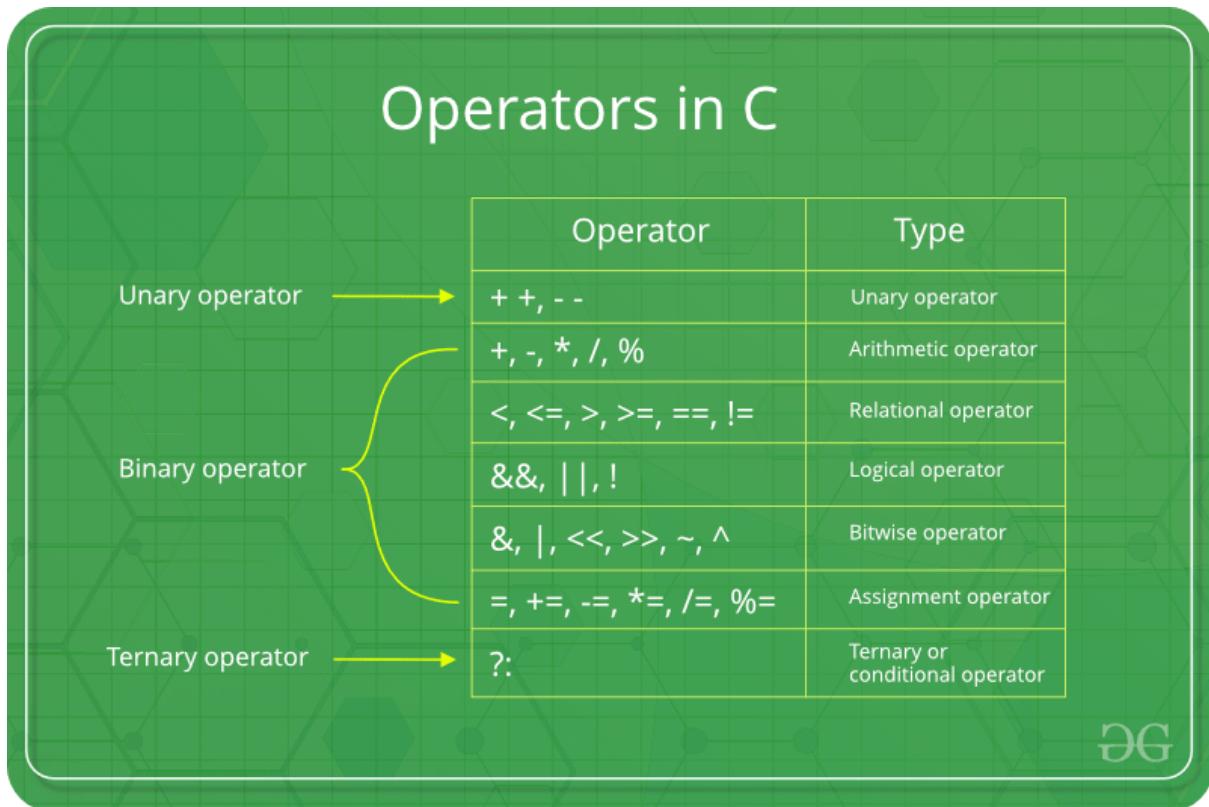
ثم يتأكد مرة اخرى إذا كان **(cin.fail() خطأ (false)** يطلع من اللوب ثم يرجع الرقم على الشاشة  
كالمثال التالي:-

```
int ValidateNumber()
{
    int Number;
    cout << "Please enter a number: ";
    cin >> Number;
    while (cin.fail())
    {
        cin.clear();
        cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

        cout << "Invalid Number, Enter a valid one: ";
        cin >> Number;
    }
    return Number;
}
```



# Lesson #12 - Bitwise AND operator



EG

تُشَدِّد العمليات **البَيْنِيَّة** (أو الثانية) على مستوى **البَيْت** من البيانات وذلك باستعمال العامل التالي:

**ـ عامل Bitwise AND البَيْنِيَّ (&)**

**12** = 0 0 0 0 1 1 0 0    (In Binary)

**25** = 0 0 0 1 1 0 0 1    (In Binary)

//Bitwise **AND** Operation of 12 and 25

0 0 0 0 1 1 0 0

& 0 0 0 1 1 0 0 1

—————

0 0 0 0 1 0 0 0 = **8**    (In decimal)

سبب ذلك أن عامل AND يعمل على مستوى البٌit (bit level) ويستخدم جدول Bitwise AND Truth Table التالي:

## Bitwise AND Truth Table

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

```
// Example : Bitwise AND
```

```
#include <iostream>
using namespace std;

int main() {
    // declare variables
    int a = 12, b = 25;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a & b = " << (a & b) << endl;

    return 0;
}
```

Microsoft Visual Studio Debug Console

```
a = 12
b = 25
a & b = 8
```



# Lesson #13 - Bitwise OR operator

تُنفذ العمليات **البٌتية** (أو الثانية) على مستوى **البٌت** من البيانات وذلك باستعمال العامل التالي:

| - عامل **OR** البٌتـي (Bitwise OR)

**12** = 0 0 0 0 1 1 0 0                                  (In Binary)

**25** = 0 0 0 1 1 0 0 1                                  (In Binary)

//Bitwise **OR** Operation of 12 and 25

0	0	0	0	1	1	0	0
	0	0	0	1	1	0	1
<hr/>							
0 0 0 1 1 1 0 1 = <b>29</b> (In decimal)							

سبب ذلك أن عامل **OR** يعمل على مستوى **البٌت** (**bit level**)، ويستخدم جدول (**Bitwise OR Truth Table**) التالي:

## Bitwise OR Truth Table

a	b	a   b
0	0	0
0	1	1
1	0	1
1	1	1

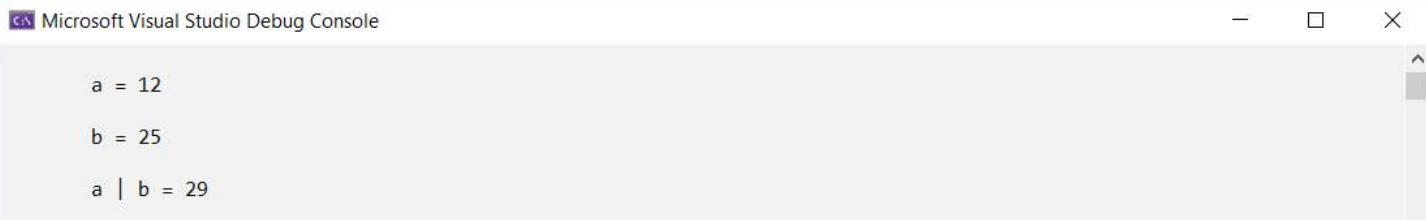
```
// Example : Bitwise OR
```

```
#include <iostream>
using namespace std;

int main()
{
    int a = 12, b = 25;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a | b = " << (a | b) << endl;

    return 0;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console displays three lines of text output:

```
a = 12
b = 25
a | b = 29
```



# Lesson #14 - Declaration Vs Definition

المكونات الضرورية لإضافة دالة إلى البرنامج؟ **هناك ثلاثة مكونات:**

- إعلان **Declaration** الدالة.
- استدعاءات **Calls** الدالة.
- وتعريف **Definition** الدالة.

## 1- إعلان الدالة The Function Declaration

تماماً كما لا يمكنك استخدام متغير دون إخبار المترجم أولاً بما هو عليه، لا يمكنك أيضاً استخدام دالة دون إخبار المترجم عنها. هناك طريقتان ل القيام بذلك. الطريقة (**النهج**) هي إعلان **declare** الدالة قبل أن يتم استدعاؤها. يتم الإعلان عن الدالة (**starline()** في السطر:

```
void starline();
```

يخبر الإعلان **declaration** المترجم أننا في مرحلة لاحقة نخطط لتقديم دالة تسمى **starline**. تحدد الكلمة المفتاحية **void** أن الدالة لا تحتوي على (ليس لها) قيمة إرجاع ، وأن الأقواس الفارغة () تشير إلى أنها لا تأخذ أي وسائط **arguments**.

لاحظ أنه يتم إنهاء إعلان **declaration** الدالة بفاصلة منقوطة ; .

## 2- استدعاء الدالة Calling the Function

يتم استدعاء **called** الدالة (أو يتم استحضارها **invoked** أو تنفيذها **executed**) ثلث مرات من الدالة **main()** يبدو كل استدعاء من الاستدعاءات الثلاثة كما يلي:

```
starline();
```

الكود:

هذا هو كل ما نحتاجه لاستدعاء الدالة: اسم الدالة متبوعة بالأقواس. يشبه بناء جملة الاستدعاء تلك الموجودة في الإعلان، باستثناء أنه لا يتم استخدام نوع الإرجاع.

يتم إنهاء الاستدعاء بفواصل منقوطة. يؤدي تنفيذ عبارة الاستدعاء إلى تنفيذ الدالة؛ بمعنى، يتم نقل التحكم إلى الدالة، ويتم تنفيذ العبارات في تعريف الدالة (التي سنقوم بفحصها بعد لحظة)، ثم يتم عودة التحكم إلى العبارة التالية لاستدعاء الدالة.

### The Function Definition 3

أخيراً ، نأتي إلى الدالة نفسها ، والتي يشار إليها باسم تعريف **definition** الدالة . يحتوي التعريف على الكود الفعلي للدالة . إليك تعريف الدالة : **starline()**

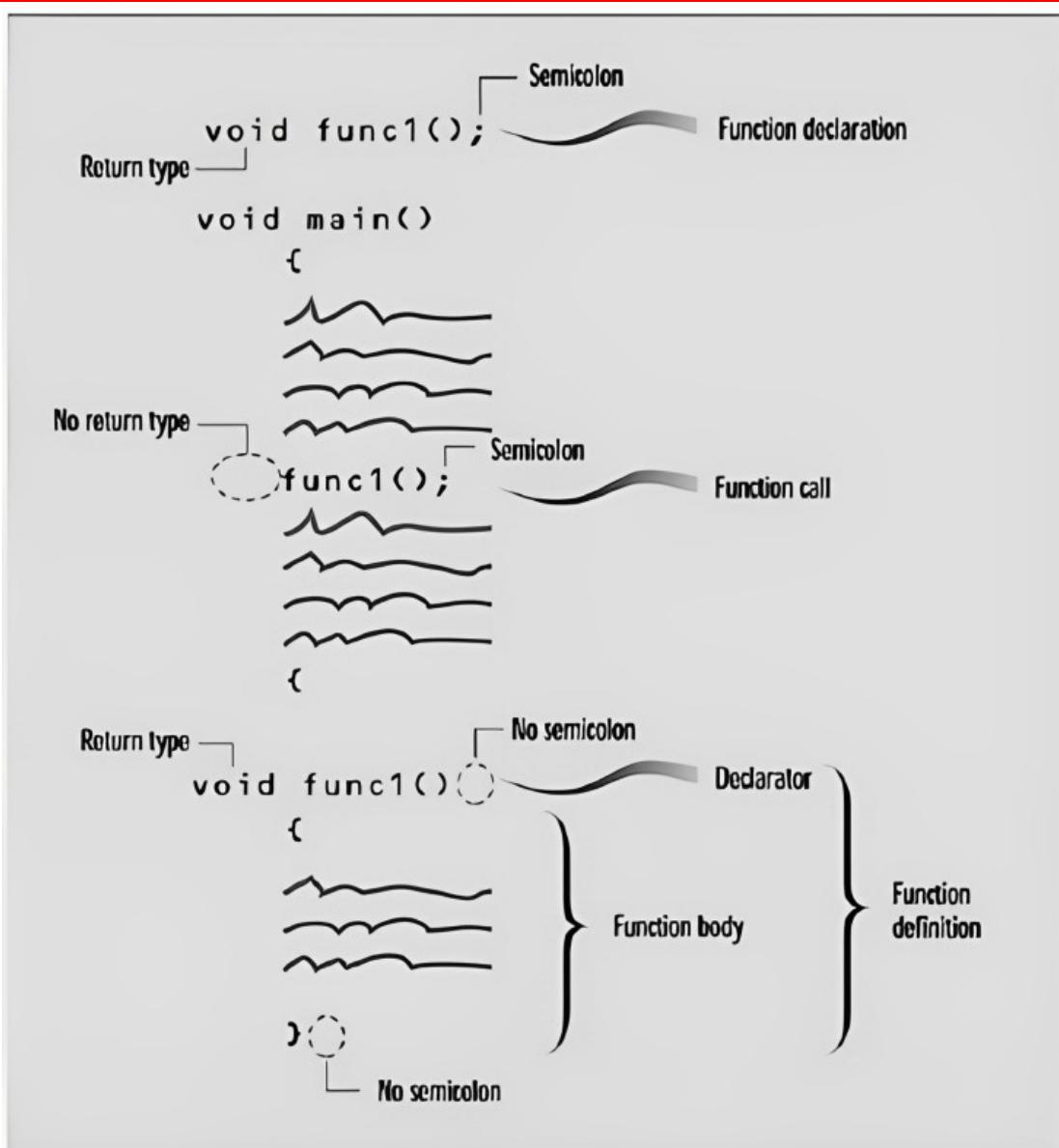
ال코드

```
void starline() //declarator
{
    for(int j=0; j<45; j++) // function body
        cout >> '*' << endl;
}
```

يتكون التعريف من سطر المُعلن **declarator** ، متبوعاً بجسم **body** الدالة . يتكون جسم الدالة من العبارات التي تشكل الدالة ، وتحددتها الأقواس {} .

يجب أن يتوافق المُعلن مع الإعلان **declaration** : يجب أن يستخدم نفس اسم الدالة ، وأن يكون له نفس أنواع الوسائط بنفس الترتيب (إذا كانت هناك وسائط) ، وأن يكون له نفس نوع الإرجاع .

لاحظ أن المُعلن لا ينتهي بفاصلة منقطة . يوضح الشكل التالي صيغة جملة **syntax** إعلان الدالة واستدعاء الدالة **call** وتعريف **definition** الدالة .



 نصيحة:

الأفضل دائمًا وضع تعاريف الدوال ( Functions Declarations ) قبل الدالة **main()** وتعريف محتوى هذه الدوال ( Function Definition ) بعد الدالة **main()** لأن قراءة الكود ستصبح أسهل.

```
#include <iostream>
using namespace std;

void startline();

int main()
{
    startline();
}

void startline()
{
    for (int i = 0; i < 45; i++)
        cout << "*";
}
```



# Lesson #15 - Default Parameters

تتيح لك وضع قيم إفتراضية للباراميترات مما يجعلك عند إستدعاء الدالة مخير على تمرير قيم مكان الباراميترات بدل أن تكون مجبأً على ذلك.



## مصطلحات تقنية:

القيمة الإفتراضية التي نضعها للباراميتر يقال لها **Default Argument**.



في المثال التالي قمنا بتعريف دالة إسمها **printLanguage**. هذه الدالة فيها باراميتر واحد إسمه **language** يملك النص "English" كقيمة إفتراضية. كل ما تفعله هذه الدالة عند إستدعاءها هو طباعة قيمة الباراميتر **language**.

```
#include <iostream>
```

```
using namespace std;
```

هنا قمنا بتعريف دالة إسمها **printLanguage** عند إستدعائها يمكنك تمرير قيمة لها مكان الباراميتر **language** ويمكنك عدم تمرير قيمة لأنه أصلاً يملك قيمة.

```
void printLanguage(string language="English")  
{  
    cout << "Your language is " << language << endl;  
}
```

```
int main()
```

```
{
```

هنا قمنا باستدعاء الدالة **printLanguage()** بدون تمرير قيمة مكان الباراميتر **language** وبالتالي ستظل قيمته "English"

```
printLanguage();
```

هنا قمنا باستدعاء الدالة **printLanguage()** مع تمرير القيمة 'Arabic' للباراميتر **language** وبالتالي ستصبح قيمته "Arabic"

```
printLanguage("Arabic");
```

```
return 0;
```

```
}
```

```
Microsoft Visual Studio Debug Console
```

```
Your language is English
```

```
Your language is Arabic
```

**ملاحظة:** بما أن الباراميتر **language** يملك قيمة بشكل إفتراضية، فهذا يعني أنه لم تعد مجبـر على تمرير قيمة له عند إستدعاء الدالة لأنـه أصلـاً يملك قيمة.



إذا كانت الدالة تملك أكثر من باراميـتر وترـيد وضع قيمة إفتراضـية لأحد الباراميـترات التي تمـكـلـها فقط فيـجب وضع الـبارـاميـترـاتـ الـتـيـ تـمـلـكـ قـيمـ إـفـتـرـاضـيـةـ فيـ الآـخـرـ. إنـ لمـ تـرـدـ ذـلـكـ سـتـكـونـ مـجـبـرـ عـلـىـ وـضـعـ قـيمـ إـفـتـرـاضـيـةـ لـجـمـيعـ الـبـارـاميـترـاتـ الـمـوـجـودـةـ بـعـدـ أـوـلـ بـارـاميـترـ وـضـعـتـ لهـ قـيمـةـ إـفـتـرـاضـيـةـ.



## ⚠ أخطاء قد تظهر بسبب وضع قيم إفتراضية للباراميـترـاتـ

في المـثالـ التـالـيـ قـمـنـاـ بـاعـطـاءـ **c**ـ قـيمـةـ إـفـتـرـاضـيـةـ وـهـذـاـ لـنـ يـسـبـبـ مشـكـلـةـ لـأـنـهـ لـاـ يـوـجـدـ بـعـدـ أـيـ بـارـاميـترـ.

```
void printMax(int a, int b, int c = 0)
{
}
```

المـثالـ التـالـيـ فـيـهـ مشـكـلـةـ حـيـثـ أـنـنـاـ قـمـنـاـ بـاعـطـاءـ **b**ـ قـيمـةـ إـفـتـرـاضـيـةـ وـلـمـ نـعـطـيـ قـيمـةـ إـفـتـرـاضـيـةـ لـلـبـارـاميـترـ **c**ـ الـمـوـجـودـ بـعـدـهـ.

هـذـهـ الدـالـلـةـ سـتـسـبـبـ خـطـأـ فـيـ الـكـوـدـ وـالـذـيـ يـعـنـيـ أـنـ الـمـشـكـلـةـ هـيـ نـسـيـانـ وـضـعـ قـيمـةـ إـفـتـرـاضـيـةـ لـلـبـارـاميـترـ الـثـالـثـ.

```
void printMax(int a, int b = 0, int c)
{
}
```

في المـثالـ التـالـيـ قـمـنـاـ بـاعـطـاءـ **b**ـ وـ **c**ـ قـيمـةـ إـفـتـرـاضـيـةـ وـهـذـاـ لـنـ يـسـبـبـ مشـكـلـةـ لـأـنـهـ لـاـ يـوـجـدـ بـعـدـهـمـاـ أـيـ بـارـاميـترـ.

```
void printMax(int a, int b = 0, int c = 0)
{
}
```

المثال التالي فيه مشكلة حيث أننا قمنا بإعطاء **a** قيمة إفتراضية ولم نعطي قيمة إفتراضية للباراميترین **b** و **c** الموجودين بعده.

هذه الدالة ستسبب خطأ في الكود والذي يعني أن المشكلة هي نسيان وضع قيمة إفتراضية للباراميترین الثاني و الثالث.

```
void printMax(int a = 0, int b, int c)
{
}
```

في المثال التالي قمنا بإعطاء **a** و **b** و **c** قيم إفتراضية، أي كل الباراميترات وبالتالي لا يوجد أي مشكلة هنا.

```
void printMax(int a = 0, int b = 0, int c = 0)
{
}
```



# Lesson #16 - Function Overloading

تتيح لغة البرمجة (C++) للمبرمجين **تحديد أكثر من تعريف واحد لاسم دالة أو عامل داخل نطاق معين**، وهذا ما يسمى بالتحميل الزائد للوظيفة وهو (**Overloading**)، حيث يصبح بإمكان المبرمج أن يقوم بتعريف أكثر من عامل (**Constructor**) أو (**Function**) أو (**Operator**) لهم نفس الاسم ولكنهم يختلفون في عدد أو نوع الـ (**Parameters**).

## مفهوم الـ Overloading

**Overloading** تعني تعريف أكثر من **عامل أو دالة أو كونسٹرکتور** (هو عبارة عن دالة، يتم استدعائها أثناء إنشاء كائن من الكلاس لإعطاء قيم أولية للخصائص الموجودة فيه) **لهم نفس الاسم ولكنهم يختلفون في عدد أو نوع الباراميترات.**

- عند تعريف أكثر من كونسٹرکتور لهم نفس الاسم يكون الهدف أنه عند إنشاء كائن يكون هناك أكثر من طريقة متاحة لتمرير قيم أولية للخصائص الموجودة فيه.
- عند تعريف أكثر من دالة لهم نفس الاسم يكون الهدف منهم إمكانية تنفيذ نفس العملية مع مراعاة عدد وأنواع القيم التي يتم تمريرها للدالة عند استدعائهما.
- عند تعريف أكثر من عامل لهم نفس الرمز يكون الهدف منهم تصغير حجم الكود عند التعامل مع الكائنات.

## شروط الـ Overloading

يطبق فقط على **العامل و الدوال و الكونسٹرکتورات**.

- يجب أن يملكون نفس الاسم.
- يجب أن يختلفوا في نوع أو عدد الباراميترات.
- بالنسبة للدوال، نوع الإرجاع غير مهم لأن المترجم لا يستطيع التفريق بين الدوال إذا كانوا مختلفين في نوع الإرجاع فقط.

في المثال التالي قمنا بتعريف دالتين اسمهما **sum** ونوعهما **void** ولكنها مختلفتان عن بعضهما بأنواع الباراميترات.

- الدالة الأولى مهمتها جمع أي عددين نوعهما **int** نمررها لها عند استدعاءها ومن ثم طباعة الناتج.
- الدالة الثانية مهمتها جمع أي عددين نوعهما **float** نمررها لها عند استدعاءها ومن ثم طباعة الناتج.

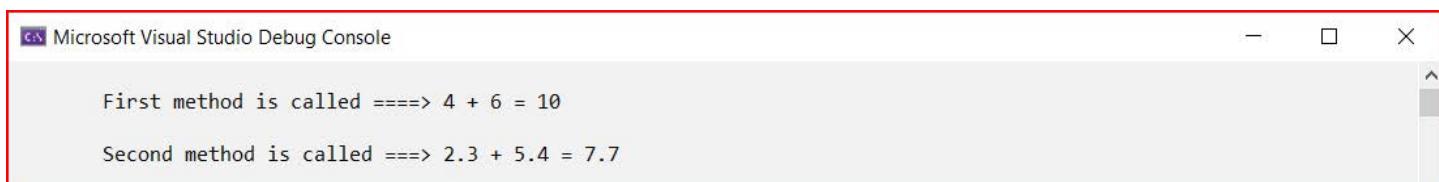
```
#include <iostream>
using namespace std;

void sum(int a, int b)
{
    cout << "First method is called ===> " << a << " + " << b << " = " << (a + b) << endl;
}

void sum(double a, double b)
{
    cout << "Second method is called ===> " << a << " + " << b << " = " << (a + b) << endl;
}

int main()
{
    sum(4, 6);           // باراميتر نوعهم 2 هنا سيتم استدعاء الدالة التي تأخذ int
    sum(2.3, 5.4);      // باراميتر نوعهم 2 هنا سيتم استدعاء الدالة التي تأخذ double
    return 0;
}
```

سنحصل على النتيجة التالية عند التشغيل.



The screenshot shows the Microsoft Visual Studio Debug Console window. It displays two lines of text output:

```
First method is called ===> 4 + 6 = 10
Second method is called ==> 2.3 + 5.4 = 7.7
```

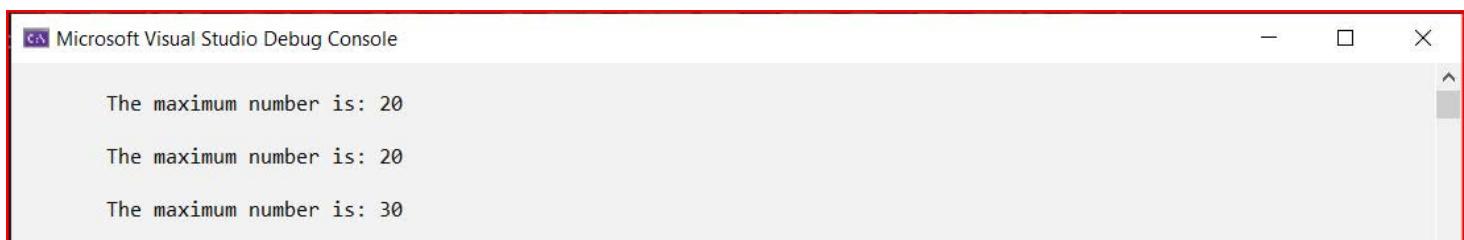
كما لاحظنا هنا، في كل مرة قمنا فيها باستدعاء الدالة **sum()** وجدنا أن المترجم قام باستدعاء الدالة **sum()** المناسبة لنوع الباراميترات التي كنا نمرر لها.

في المثال التالي قمنا بتعريف ثلاثة دوال إسمهم **maximum()** و نوعهم **double**

- الدالة الأولى تأخذ 2 باراميتر عبارة عن أرقام، و هي تعطينا العدد الأكبر بينهما.
- الدالة الثانية تأخذ 3 باراميترات عبارة عن أرقام، و هي تعطينا العدد الأكبر بينهم.
- الدالة الثالثة تأخذ 4 باراميترات عبارة عن أرقام، و هي تعطينا العدد الأكبر بينهم.

```
#include <iostream>
using namespace std;
// هذه الدالة تعطيها رقمين فترجع لك العدد الأكبر بينهما
double maximum(double a, double b)
{
    if (a > b)
        return a;
    else
        return b;
}
// هذه الدالة تعطيها ثلاثة أرقام فترجع لك العدد الأكبر بينهم
// و هي تعتمد على الدالة السابقة لمقارنة أول عددين مع العدد الثالث
double maximum(double a, double b, double c)
{
    if (maximum(a, b) > c)
        return maximum(a, b);
    else
        return c;
}
// هذه الدالة تعطيها ثلاثة أرقام فترجع لك العدد الأكبر بينهم
// و هي تعتمد على الدالتين السابقتين لمقارنة أول ثلاثة أعداد مع العدد الرابع
double maximum(double a, double b, double c, double d)
{
    if (maximum(a, b, c) > d)
        return maximum(a, b, c);
    else
        return d;
}
// هنا قمنا بتعريف الدالة
int main()
{
    // هنا سيتم استدعاء الدالة التي تأخذ 2 باراميتر
    cout << "The maximum number is: " << maximum(5, 20) << endl;
    // هنا سيتم استدعاء الدالة التي تأخذ 3 باراميتر
    cout << "The maximum number is: " << maximum(5, 20, 15) << endl;
    // هنا سيتم استدعاء الدالة التي تأخذ 4 باراميتر
    cout << "The maximum number is: " << maximum(5, 20, 15, 30);
    return 0;
}
```

سنحصل على النتيجة التالية عند التشغيل.



```
The maximum number is: 20
The maximum number is: 20
The maximum number is: 30
```

كما لاحظنا هنا، في كل مرة قمنا فيها باستدعاء الدالة **maximum()** وجدنا أن المترجم قام باستدعاء الدالة **maximum()** التي تحتوي على نفس عدد الباراميترات الذي كنا نمرره لها، وداخلياً ربطنا الدوال ببعضها.



# Lesson #17 - Call Stack/Call Hierarchy

ال **Stack** يستخدم لإنشاء **كائن** يمثل حاوية تخزن العناصر التي نضيفها فيها بشكل متسلسل فوق بعضها البعض مما يجعلك قادر فقط على التعامل مع العنصر الموجود في أعلىها.

أو هي عبارة عن خط انتظار لمجموعة من البيانات، ما يميز هذا الخط بأنه مفتوح من اتجاه واحد فقط طريقة تخزين العناصر أو البيانات (أي أنها تدخل وتخرج من بوابة واحدة) في هذه الحاوية (**Stack**) تعتمد أسلوب **LIFO** الذي هو اختصار لجملة **Last In First out** والتي تعني أن **العنصر الأخير الذي يتم إضافته في الحاوية هو أول عنصر يمكن إخراجه منها** - وهذا بديهي بما أنه لا يوجد لدينا إلا فتحة واحدة لإدخال وإخراج العناصر - كالتالي:

## Program

```
#include <iostream>
using namespace std;

void Function4()
{
    cout << "Hi I'm function4 " << endl;
}

void Function3()
{
    Function4();
}

void Function2()
{
    Function3();
}

void Function1()
{
    Function2();
}

int main() {
    Function1();
    return 0;
}
```

## Call Stack ➔

ويمكن اجراء عمليتان على **Stack** وهما **Push** و **Pop** وتعني **Push** لادخال عنصر جديد في ال **Stack** فوق أعلى عنصر و **Pop** لخروج أعلى عنصر من **Stack**.

## مصطلحات:

- **Call Stack = Call Hierarchy** = تدرج او تسلسل استدعاء الفانكشن
- **Function Push()** = دخول العناصر (وظيفتها تخزن العناصر - يدخل عنصر عنصر - الى داخل ال **Stack** )
- **Function Pop()** = خروج العناصر (وظيفتها تخرج او تزحف العناصر - حذف عنصر عنصر - من داخل ال **Stack** )
- **FILO = first in last out** = اول عنصر يدخل هو اخر عنصر يخرج
- **LIFO = last in first out** = اخر عنصر يدخل هو اول عنصر يخرج
- **view Call Hierarchy** = نستخدم هذا الامر من القائمة المنسدلة التي تظهر عند الضغط كليك يمين على الفانكشن

```
1 // Lesson 17 - Call Stack - Call Hierarchy.cpp : This
2
3 #include <iostream>
4 #include<stack>
5
6 using namespace std;
7
8
9 void Function4()
10 {
11     cout << "Function4"
12 }
13
14 void Function3()
15 {
16     Function4()
17 }
18
19 void Function2()
20 {
21     Function3()
22 }
23
24 void Function1()
25 {
26 }
```

120 % No issues found

Call Hierarchy

My Solution

- Function4()
  - Calls To 'Function4'
  - Calls From 'Function4'

Quick Actions and Refactorings... Ctrl+.

Rename... Ctrl+R, Ctrl+R

Generate Graph of Include Files

View Code Ctrl+Alt+O

Peek Definition Alt+F12

Go To Definition F12

Go To Declaration Ctrl+Alt+F12

Find All References

**View Call Hierarchy** Ctrl+K, Ctrl+T

Peek Header / Code File Ctrl+K, Ctrl+J

Toggle Header / Code File Ctrl+K, Ctrl+O

Breakpoint

Run To Cursor Ctrl+F10

Force Run To Cursor

Snippet

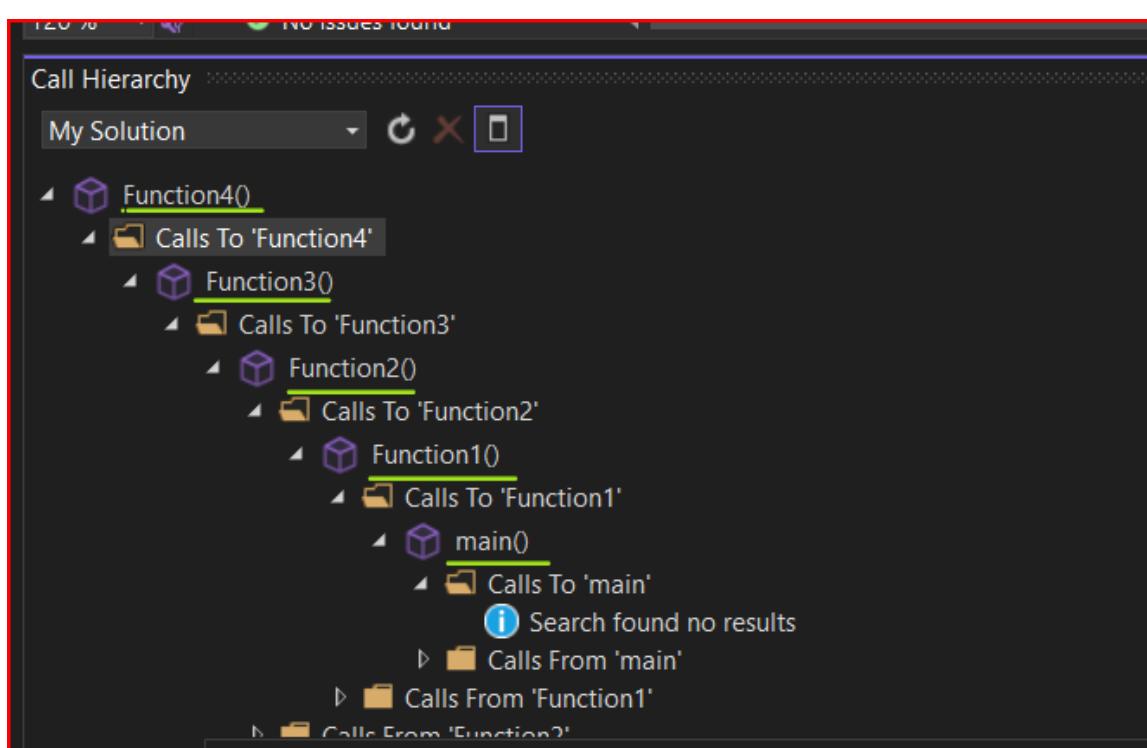
Cut Ctrl+X

Copy Ctrl+C

Paste Ctrl+V

Annotation

Outlining



```
#include <iostream>
#include<stack>

using namespace std;

void Function4()
{
    cout << "This is Function 4" << endl;
}

void Function3()
{
    Function4();
}

void Function2()
{
    Function3();
}

void Function1()
{
    Function2();
}

int main()
{
    Function1();

    return 0;
}
```



# Lesson #18 - Visual Studio Enterprise 2022

## لمحات

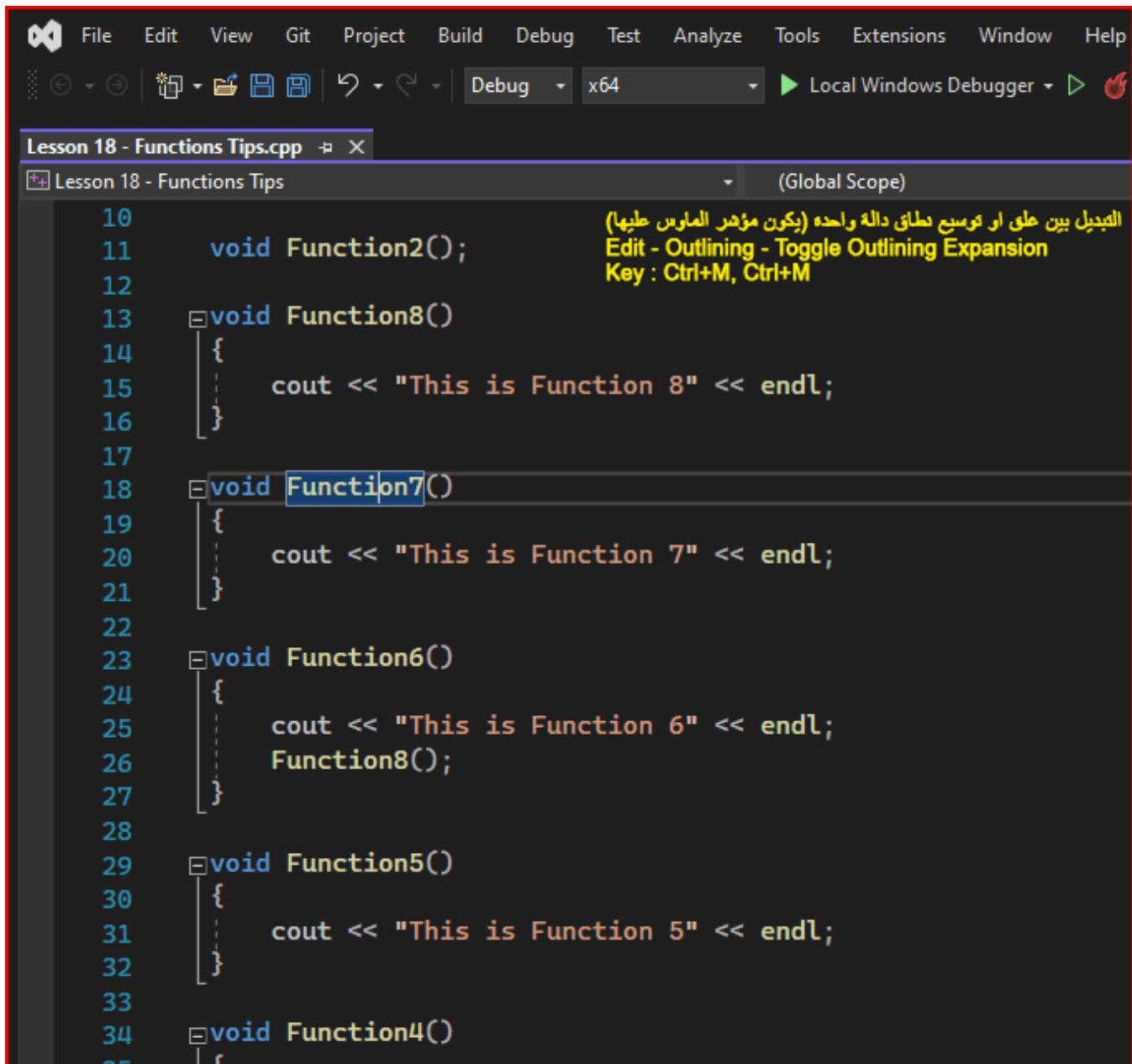


```
4     #include <iostream>
5
6
7     using namespace std;
8
9     void Function6();
10
11    void Function2();
12
13    void Function8()
14    {
15        cout << "This is Function 8" << endl;
16    }
17
18    void Function7()
19    {
20        cout << "This is Function 7" << endl;
21    }
22
23    void Function6()
24    {
25        cout << "This is Function 6" << endl;
26        Function8();
27    }
28
```

- التبديل بين غلق او توسيع نطاق دالة واحدة (يكون مؤشر الماوس عليها)

Edit - Outlining - Toggle Outlining Expansion

Key : Ctrl+M, Ctrl+M



The screenshot shows the Visual Studio IDE interface with the following details:

- Menu Bar:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbar:** Includes icons for file operations like Open, Save, and Print, along with Debug and Build buttons.
- Status Bar:** Shows "Debug" and "x64" selected, and "Local Windows Debugger".
- Code Editor:** Displays the file "Lesson 18 - Functions Tips.cpp". The code contains several nested functions: Function2(), Function8(), Function7(), Function6(), Function5(), and Function4().
- Outlining Feature:** The code is outlined, with function names highlighted in blue and enclosed in square brackets. A tooltip on the right side of the editor area provides instructions for the outlining feature.

التبديل بين على او توسيع نطاق دالة واحدة (يكون مؤهلاً الموارس عليها)  
Edit - Outlining - Toggle Outlining Expansion  
Key : Ctrl+M, Ctrl+M

• التبديل بين غلق او توسيع نطاق جميع الدوال في الكود (في اي مكان في الكود)

Edit - Outlining - Toggle All Outlining

Key : Ctrl+M, Ctrl+L

The screenshot shows the Visual Studio Enterprise 2022 interface with the following details:

- Menu Bar:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbar:** Includes icons for Undo, Redo, Save, Open, and various project management tools.
- Status Bar:** Debug, x64, Local Windows Debugger.
- Code Editor:** The file "Lesson 18 - Functions Tips.cpp" is open. The code defines several nested functions: Function2, Function8, Function7, Function6, Function5, and Function4. Each function contains a cout statement printing its name followed by endl. The code editor uses syntax highlighting and displays line numbers from 10 to 37. A tooltip is visible in the top right corner providing information about the outlining feature.

البعض بين على او مرسخ نطاق جميع الدوال في الكود (في اي مكان في الكود)  
Edit - Outlining - Toggle All Outlining  
Key : Ctrl+M, Ctrl+L

• ايقاف الخطوط (الغاء علامات الطي والتوصيع على كامل الكود)

Edit - Outlining - Stop Outlining

Key : Ctrl+M, Ctrl+P

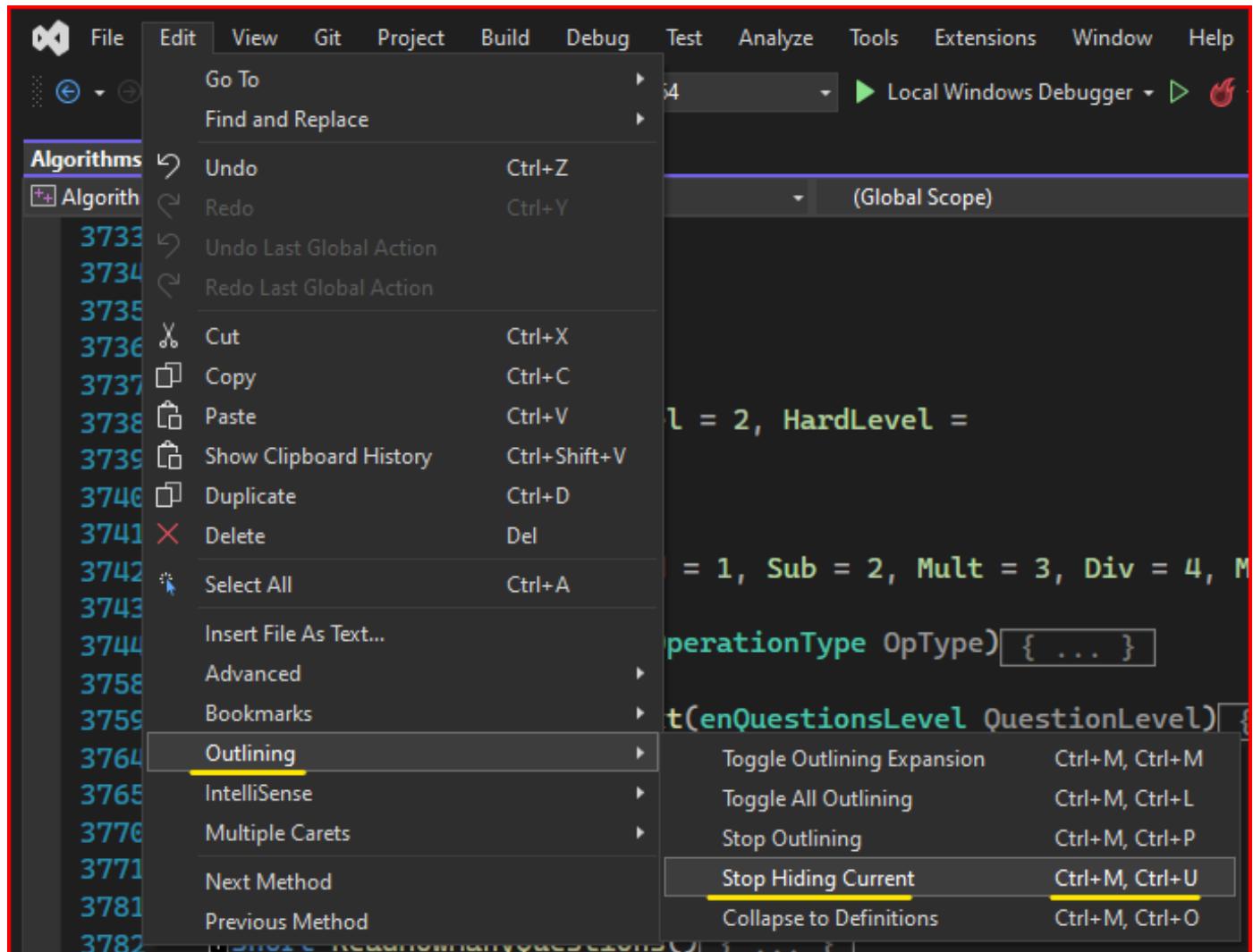
• إيقاف الخطوط (إلغاء علامات الطي والتوصيع على كامل الكود)  
Edit - Outlining - Stop Outlining  
Key : Ctrl+M, Ctrl+P

```
10     void Function2();  
11  
12     void Function8()  
13     {  
14         cout << "This is Function 8" << endl;  
15     }  
16  
17     void Function7()  
18     {  
19         cout << "This is Function 7" << endl;  
20     }  
21  
22     void Function6()  
23     {  
24         cout << "This is Function 6" << endl;  
25         Function8();  
26     }  
27  
28     void Function5()  
29     {  
30         cout << "This is Function 5" << endl;  
31     }  
32  
33     void Function4()  
34     {  
35         cout << "This is Function 4" << endl;  
36     }  
37 
```

• توقف عن إخفاء الحالي

Edit - Outlining - Stop Hiding Current

Key : Ctrl+M, Ctrl+U



• طي إلى التعريفات

Edit - Outlining - Collapse to Definitions

Key : Ctrl+M, Ctrl+O

The screenshot shows the Visual Studio Enterprise 2022 interface with the following details:

- File**, **Edit**, **View**, **Git**, **Project**, **Build**, **Debug**, **Test**, **Analyze**, **Tools**, **Extensions**, **Window**, **Help** menu items.
- Toolbar icons: Save, Undo, Redo, Build, Run, Stop.
- Build configuration dropdown: Debug, x64.
- Platform dropdown: Local Windows Debugger.
- Code editor window titled "Lesson 18 - Functions Tips.cpp".
- Code content:

```
10
11     void Function2();
12
13     void Function8()
14     {
15         cout << "This is Function 8" << endl;
16     }
17
18     void Function7()
19     {
20         cout << "This is Function 7" << endl;
21     }
22
23     void Function6()
24     {
25         cout << "This is Function 6" << endl;
26         Function8();
27     }
28
29     void Function5()
30     {
31         cout << "This is Function 5" << endl;
32     }
33
34     void Function4()
35     {
36         cout << "This is Function 4" << endl;
37     }
```
- Right-hand margin annotations:
  - Line 10: "ابحث عن التعاريف" (Search definitions) in Arabic.
  - Line 11: "Edit - Outlining - Collapse to Definitions" in English.
  - Line 11: "Key : Ctrl+M, Ctrl+O" in English.

: فانكشن موجود في الـ **main** .. اريد الذهاب **لتعريف** هذا الفانكشن في الكود بهذه الطريقة: **Go To Definition**

```
using namespace std;
void Function6();
void Function2();
void Function8() { ... }
void Function7() { ... }
void Function6() { ... }
void Function5() { ... }
void Function4() { ... }
void Function3() { ... }
void Function2() { ... }
void Function1() { ... }

int main()
{
    Function1();
    Function4();
    Function6();
}
```

: فانكشن موجود في ال **main** .. اريد الذهاب **لإعلان** هذا الفانكشن في الكود بهذه الطريقة:

```
Lesson 18 - Functions Tips.cpp

using namespace std;
void Function6();
void Function2();
void Function8() { ... }
void Function7() { ... }
void Function6() { ... }
void Function5() { ... }
void Function4() { ... }
void Function3() { ... }
void Function2() { ... }
void Function1() { ... }

int main()
{
    Function1();
    Function4();
    Function6();
}
```

: عرض التسلسل الهرمي للاستدعاءات - لمعرفة الذي استدعي هذا الفانكشن [View Call Hierarchy](#)

The screenshot shows the Visual Studio Enterprise 2022 interface with the following details:

- Top Bar:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help, Search (Ctrl+Q).
- Toolbars:** Standard and Debugging.
- Code Editor:** Displays the file "Lesson 18 - Functions Tips.cpp". The code defines a class with eight member functions (Function1 to Function8) and a main function that calls them. The code editor shows syntax highlighting and line numbers from 12 to 67.
- Status Bar:** Shows 121% zoom and "No issues found".
- Call Hierarchy Tool Window:** A floating window titled "Call Hierarchy" which lists the members of the class. It includes a dropdown menu set to "My Solution", a refresh button, and a close button. Below the list, it says: "Right-click on a member name in the Code Editor, and then click View Call Hierarchy to view the member's call hierarchy in this tool window."
- Bottom Navigation:** Call Hierarchy, Output, Error List.

البحث عن جميع المراجع - كم مرة تم الاشارة الى هذا الفانكشن ومكانه في الكود : [Find All References](#)

The screenshot shows the Visual Studio Enterprise 2022 interface. The top menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help, and Search (Ctrl+Q). The toolbar has icons for file operations like Open, Save, and Print. The status bar at the bottom shows 'Lesson #18 - Functions Tips.cpp' and '121 %'. The main code editor window displays the following C++ code:

```
Lesson 18 - Functions Tips.cpp
Lesson 18 - Functions Tips
18  +void Function7() { ... }
22
23  +void Function6() { ... }
28
29  +void Function5() { ... }
33
34  +void Function4() { ... }
38
39  +void Function3() { ... }
44
45  +void Function2() { ... }
49
50  +void Function1() { ... }
55
56
57
58
59  int main()
60  {
61      Function1();
62      Function4();
63      Function6();
64  }
65
66
67
```

The code editor highlights the call to `Function4()` with a blue rectangular box. A tooltip labeled '(Global Scope)' appears above the cursor, indicating the scope of the highlighted function. The bottom left corner of the interface shows the 'Output' tab.

**Peek Definition** : نظرة خاطفة على تعريف الفانكشن.. يظهر الفانكشن في نافذة زرقاء في نفس المكان.

```
Lesson 18 - Functions Tips.cpp
Lesson 18 - Functions Tips
(Global Scope)

40     {
41         Function4();
42         Function2();
43     }
44
45     void Function2() { ... }
46
47     void Function1()
48     {
49         Function2();
50         Function4();
51     }
52
53
54
55
56
57
58
59     int main()
60     {
61         Function1();
62
63         Function4();
64         Function6();
65     }
66
67
```

تغيير اسم الفانكشنز مرة واحدة : **Rename**

The screenshot shows the Visual Studio Enterprise 2022 interface. The top menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help, and a Search bar. The toolbar below has icons for file operations like Open, Save, and Print. The status bar at the bottom shows "120 %", a magnifying glass icon, and a green checkmark indicating "No issues found".

The main window displays a code editor for a file named "Lesson 18 - Functions Tips.cpp". The code contains several nested function definitions:

```
25     cout << "This is Function 6" << endl;
26     Function8();
27 }
28
29 void Function5()
30 {
31     cout << "This is Function 5" << endl;
32 }
33
34 void Function4()
35 {
36     cout << "This is Function 4" << endl;
37 }
38
39 void Function3()
40 {
41     Function4();
42     Function2();
43 }
44
45 void Function2()
46 {
47     Function3();
48 }
49
50 void Function1()
```

The code uses vertical dotted lines to indicate nesting levels. The "Function1" definition is currently selected.



# Lesson #19 - Recursion

## الـ Recursion (الاستدعاء الذاتي)

هي تقنية إجراء استدعاء دالة نفسها. توفر هذه التقنية طريقة لتقسيم المشكلات المعقدة إلى مشكلات بسيطة يسهل حلها. ويقصد به أيضاً : بناء كود يعيد تنفيذ نفسه بنفسه إلى حين إتمام مهمة معينة.

عملياً لتحقيق الاستدعاء الذاتي **نقوم ببناء دالة تستدعي نفسها بنفسها** وقد ترجع قيمة لنفسها أيضاً، وهذا النوع من الدوال يسمى **Recursive Function**.

### شروط الاستدعاء الذاتي

**أول ما عليك التفكير به** عند بناء دالة تستدعي نفسها هو **الشرط** الذي يجب وضعه **لتحديد** متى يجب أن **توقف** عن استدعاء نفسها لأنك إن لم تفعل ذلك فإن ذاكرة جهازك (RAM) ستمتلئ أو المعالج (Processor) سيعمل بأقصى سرعة ممكنة وبلا أي توقف الأمر الذي سيجعل الجهاز يعطل ولا يستطيع الإستجابة لأي أمر آخر تعطيه له.

```
#include<iostream>
using namespace std;

void PrintNumbersFromAtoB(int a, int b)
{
    if (a <= b) {
        cout << a << " ";
        PrintNumbersFromAtoB(a + 1, b);      // استدعاء الفانكتشن نفسها
    }
}

void PrintNumbersFromBtoA(int b, int a) {
    if (b >= a) {
        cout << b << " ";
        PrintNumbersFromBtoA(b - 1, a);
    }
}

int MyPower(int B, int P)
{
    if (P == 0) {
        return 1;
    }
    else {
        return (B * (MyPower(B, P - 1)));
    }
}

int fact(int n)
{
    if (n == 0 || n == 1) {
        return 1;
    }
    else {
        return n * fact(n - 1);
    }
}

int main() {
    cout << "\n\t Print Numbers From A to B : ";
    PrintNumbersFromAtoB(1, 10);
    cout << "\n\n\t Print Numbers From B to A : ";
    PrintNumbersFromBtoA(10, 1);
    cout << "\n\n\tPower of number : " << MyPower(2, 4) << " ";
    cout << "\n\n\tFactorial of number : " << fact(4) << " ";
    cout << endl;
    return 0;
}
```



# Lesson #20 - Static Variables

## (المتغيرات الساكنة) Static Variable

هو متغير يتم الاحتفاظ بقيمة على طول الاليف تايم بتابع البرنامج (وهو شبيه بال **Global Variable**) ولكن ضمن اسکوب الفانكشن ويدوم لفترة حياة البرنامج.

الفاريبول العادي يتم تدميره بمجرد انتهاء الاسکوب تبعه ... الاسکوب تبعه اللي هو الفانكشن.

**المتغيرات الساكنة داخل الدالة:** عندما يتم إعلان متغير على أنه ساكن (**static**)، يتم تخصيص مساحة له طوال فترة تشغيل البرنامج. حتى إذا تم استدعاء الدالة عدة مرات، يتم تخصيص مساحة للمتغير الساكن مرة واحدة فقط، ويتم الاحتفاظ بقيمة المتغير من الاستدعاء السابق خلال الاستدعاء التالي للدالة. هذا مفيد لتنفيذ الإجراءات الروتينية (**coroutines**) في لغات C/C++ أو أي تطبيق آخر يحتاج إلى تخزين الحالة السابقة للدالة.

على سبيل المثال ، نقوم بإنشاء متغير من النوع **static int val = 0**. غالبا ما تشير هذه العلامة إلى أن المتغير محجوز في ذاكرة مخصصة للمتغيرات الثابتة.

### إليك بعض النقاط الرئيسية حول المتغيرات الساكنة داخل الدوال:

يتم الإعلان عنها باستخدام الكلمة المفتاحية "**static**" قبل نوع المتغير.

يتم تخصيصها مرة واحدة فقط، عند أول استدعاء للدالة.

تحتفظ بقيمتها بين الاستدعاءات المتعاقبة للدالة.

يكون لها نطاقاً محلياً داخل الدالة، ولا يمكن الوصول إليها من خارجها.

### إليك بعض النقاط الرئيسية حول المتغيرات الساكنة داخل الدوال:

يتم الإعلان عنها باستخدام الكلمة المفتاحية "**static**" قبل نوع المتغير.

يتم تخصيصها مرة واحدة فقط، عند أول استدعاء للدالة.

تحتفظ بقيمتها بين الاستدعاءات المتعاقبة للدالة.

يكون لها نطاقاً محلياً داخل الدالة، ولا يمكن الوصول إليها من خارجها.

//Variable مثال باستعمال كلمة Static قبل الـ

```
// ثابت Static

#include <iostream>

using namespace std;

void Fun()
{
    static int x = 1; // يجعل المتغير x مشترك لاي دالة اخرى

    cout << "Vlaue of number : " << x << endl;
    x++; // x++ = x + 1
}

int main()
{
    Fun(); // = 1
    Fun(); // = 2
    Fun(); // = 3
}
```

// Variable مثال بعدم استعمال كلمة Static قبل الـ

```
#include <iostream>

using namespace std;

void Fun()
{
    int x = 1; // من غير كلمة static
                // حيث ان الـ x تكون داخل الفانكشن فقط لا تتطاوه

    cout << "Vlaue of number : " << x << endl;
    x++; // x++ = x + 1
}

int main()
{
    Fun(); // = 1
    Fun(); // = 1
    Fun(); // = 1
}
```



# Lesson #21 - Automatic Variables

= يضع نوع البيانات **Auto** للمتغير بناء على معرفة القيمة.

لا ينصح باستخدامه .. لأن من المهم معرفة نوع المتغير المناسب لل توفير في الذاكرة من حيث الحجم وسرعة البرنامج

ويعرف نوع البيانات من خلال قيمة المتغير كالمثال التالي:-

Auto X = 10;	Type Integer
Auto Y = 20.34;	Type Double
Auto Z = "Ahmad Abdelrahim";	Type String

```
#include <iostream>

using namespace std;

void AutoVar()
{
    auto x = 10;                                // Type int
    auto y = 5.45;                               // Type double
    auto w = "Ahmed Elsayed";                   // Type string
    cout << x << endl;
    cout << y << endl;
    cout << w << endl;
}

int main()
{
    AutoVar();
}
```



# Lesson #23 - Integer Format (printf)

الدالة `printf()` هو اختصار للمصطلح **Print Formatted** ، حيث تتيح الدالة `printf()` التحكم في طريقة طباعة الجمل النصية أو ما يعرف بـ**تنسيق الجمل النصية**، وإمكانية طباعة قيمة المتغيرات داخل النص المراد طباعته.

قبل التعرف على كيفية استخدام الدالة `printf()` يجب عليك أولاً أن تعرف أن الإستخدام الخاطئ لهذه الدالة قد يؤدي إلى إغلاق البرنامج بشكل مفاجئ أو ما يعرف بـ**Crash**، وقد يؤدي أيضاً إلى وجود العديد من الثغرات الأمنية البرمجية داخل كود البرنامج، لذلك وجب التنويه.

الدالة معرفة في المكتبة `cstdio` في لغة البرمجة `C++` ، لذلك يجب كتابة الأمر `#include<cstdio>` في بداية الكود قبل إستخدامها.

النموذج المستخدم لتعريف الدالة أو ما يعرف بـ **Function Definition** أو **Function Prototype** هو كالتالي:

```
int printf(const char * format, ...);
```

حيث يمثل المتغير `format` النص ، والثلاثة نقاط ... تتيح إضافة عدد غير محدد مسبقاً من المتغيرات أو الـ `arguments` ، حيث يعرف هذا النوع من الدوال بـ **Variadic function**.

تقوم الدالة بإرجاع `return` عدد الأحرف التي تم طباعتها في حالة تنفيذ العملية بنجاح أو تقوم بإعادة رقم سالب في حالة الفشل في تنفيذ عملية الطباعة.

عند إستدعاء الدالة يجب أن يتم تمرير متغير نصي واحد على الأقل، فلا يمكن إستدعاؤها بدون تمرير أي متغيرات.

## الأداء Performance

تستعمل `iostream` ليتم طباعة البيانات إلى الطرفية ، ومن المعروف أن `iostream` بطئ للغاية مقارنة باستعمال الدالة `printf` ، بالطبع فرق الأداء لن يكون ملحوظاً عند طباعة نصوص صغيرة أو عند حفظها إلى ملفات ، لكن عندما تريد إخراج نصوص كبيرة للغاية وفي أقل وقت ممكن فعليك أن تستعمل الدالة `printf`

بالرغم من أن `cout` أبطأ قليلاً وأن `iostream` ابطأ من `cstudio` لكن `cout` تتعارف تلقائياً على نمط المتغير الذي نريد طباعته ولا حاجة لتحديد نمطه ضمن عبارة الطباعة ، حيث ان تحديد النمط في `cout` يأخذ قليلاً من الوقت أثناء عمل البرنامج ، ولكنه يحل مشكلة **type safety** اي لن يحدث خطأ بسبب اختلاف نمط المتغير الذي يكتبه المبرمج عن نمطه الحقيقي مما يمنع الأخطاء وباللغاء مزامنة `iostream` مع المكتبة القياسية تصبح `cout` سريعة جداً.

## استخدامات الدالة (printf)

طباعة النصوص أو عملية مخرجات البرنامج تعتبر الإستخدام الأساسي لهذه الدالة، بالإضافة إلى إمكانية تنسيق النصوص، ودمج قيم المتغيرات بداخل النص المراد طباعته.

فلنفترض أن لديك متغير **x** يحتوي على قيمة عددية ناتجة عن إجراء بعض العمليات الحسابية، وأنت تريد معرفة قيمة هذا المتغير، إذا قمت بكتابة إسم المتغير وهو **x** بداخل علامات التنصيص، فإن الحرف الإنجليزي **x** هو الذي سيطبعه، وليس القيمة العددية التي يحتويها المتغير، لذلك يتم إستخدام هذه الدالة لإتمام هذه العملية.

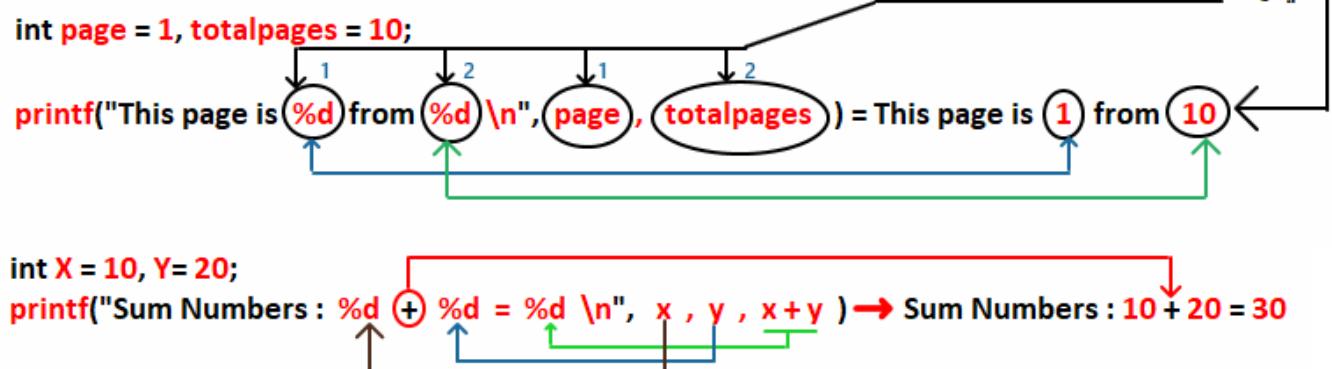
في هذا المثال نفترض أنك تريد طباعة الجملة التالية **Result = x**، بحيث يتم إستبدال الحرف **x** بقيمة المتغير، ولكن يتم تنفيذ هذه العملية يجب على الدالة معرفة أن الحرف **x** هو متغير وليس حرف، لذلك يتم إستخدام رموز خاصة ليتم إستبدالها من قبل الدالة بقيمة المتغير، وهو ما يعرف بحاجز للمكان **Placeholder** - يأخذ هذا الشكل - ( " )

يجب أيضا تحديد نوع المتغير الذي نريد طباعة قيمته، لكي تتمكن الدالة من ترجمة المتغير طبقا لنوعه إلى القيمة الصحيحة، لذلك في هذا المثال نريد أن يتم طباعة الجملة **Result =**  ، متبوعة بقيمة المتغير العددي **x** ، لذلك سوف نقوم بكتابة الأمر على الشكل التالي:

حيث قمنا بكتابة الجملة المراد طباعتها بين علامات تنصيص مزدوجة (**double quote**) " " ، وقمنا بحجز مكان بداخل النص **Placeholder** ، ليتم طباعة قيمة المتغير بداخله عن طريق كتابة الحرف **%** متبعا بالحرف **d** ، فالرمز **%d** بالنسبة للدالة **printf()** يعتبر ذو معنى خاص (حيث يتم إستبداله بقيمة المتغير) الذي تم تحديد نوعه عن طريق الحرف **d** ، ثم قمنا بتمرير المتغير **x** كأول **argument** (محددات أو مخصوصات **Specifiers**) للدالة (printf("Message %d : %.1n", id, content[id]);) كمثال :

## طباعة الأرقام الصحيحة integers

يقوم الـ **Compiler** بتبدل أول **%d** موجود وبين ما كان موقعها يبدلها بأول **Integers** وتكون **%d** للرقم الـ **Parameters** ويكون عدد **%d** بعد الـ **Parameters**



`printf("Message %d : %.1n", id, content[id]);`

`std::cout << "Message " << id << ":" << content[id] << "." << std::endl;`

لاحظ كيف ان الدالة **printf** أصبحت اقصر بكثير في الكتابة وأن النص يكون أكثر وضوحاً عند استعمالها.

```
25 int main()
26 {
27     for(int i = 1; i < 1000001; i *=10)
28     {
29         printf("%d i value\n", i);
30     }
31 }
```

```

1 i value
10 i value
100 i value
1000 i value
10000 i value
100000 i value
1000000 i value

```

```

8 //-----\n9 #include <iostream>
10
11 using namespace std;
12
13 void printnumber()
14 {
15     for(int i = 1; i < 1000001; i *=10)
16         printf("%*d i Value\n", 7, i);
17 }
18
19 void printnumber1()
20 {
21     for(int i = 1; i < 1000001; i *=10)
22         printf("%-7d i Value\n", i);
23 }
24
25 int main()
26 {
27     for(int i = 1; i < 1000001; i *=10)
28     {
29         printf("%d i value\n", i);
30     }
31
32     printf("-----\n");
33     printnumber();
34     printf("-----\n");
35     printnumber1();
36
37     return 0;
38 }

```

## مواصفات العرض - تنسيق الأرقام (Width Specification)

يعني يضع Digits مكان ال 0\* بعد الرقم في ال Parameters

("..... %0\*d", 2 or 3 or 4)

int page = 1, totalpages = 10;

printf("The page number = % 0\*d ", 2, page) = 01

printf("The page number = % 0\*d ", 3, page) = 001

في حالة ما إذا أردت طباعة مجموعة من الأرقام (أو الحروف) مع إمكانية التحكم في الحيز أو العرض **width** الذي يتم طباعة المتغير بداخله، أو بمعنى آخر إذا أردت أن يتم طباعة الأرقام أياً كان عدد رموزها **بالمحاذاة** مع باقي الأرقام، بحيث يتم طباعة الجملة **i value** أيضاً في نفس المكان من كل سطر، فكل ما عليك هو استخدام رمز خاص لتحديد الحد الأدنى الذي يتم حجزه لطباعة المتغير.

وبالتالي سوف نقوم بتعديل دالة الطباعة في المثال السابق لتصبح كالتالي:

```
printf("%*d i value\n", 7, i);
```

في هذا المثال قمنا بإضافة علامة النجمة \* والمعروفة بـ **Asterisk** بعد الرمز % وقبل الحرف الخاص بنوع البيانات **d**، ثم قمنا بتمرير الرقم 7 ليتم إستبدال علامة النجمة \* به ثم المتغير **i**.

عند تنفيذ الكود السابق سوف تقوم الدالة بحجز سبعة أماكن لحروف ليتم طباعة قيمة المتغير بهم، وإذا كان عدد أحرف المتغير **أكبر من 7**، فسوف يتم طباعة الحروف أو الأرقام المكونة لقيمة المتغير كاملة، وإن كان طول الأحرف أقل من 7 فسوف يتم إضافة مسافات " " space حتى يصبح طول المتغير 7 أحرف على الأقل.

ويمكن كتابة الكود السابق بالطريقة المختصرة (وهي الأكثر استخداماً)، حيث يتم كتابة الرقم الممثل لعرض الحقل مباشرة قبل الحرف **d** ، كالتالي:

```
printf("%7d i value\n", i);
```

بعد تنفيذ الكود السابق سوف تظهر النتيجة بالشكل التالي:

```
1 i Value
10 i Value
100 i Value
1000 i Value
10000 i Value
100000 i Value
1000000 i Value
-----
1 i Value
10 i Value
100 i Value
1000 i Value
10000 i Value
100000 i Value
1000000 i Value
```

وهنا لاحظ أن **الأرقام** تم طباعتها **بالمحاذاة إلى اليمين**، وهو **الوضع الافتراضي للطباعة**، فإذا ما أردت تغيير ذلك ليتم طباعة **الأرقام بالمحاذاة من اليسار إلى اليمين** كما في الجزء الثاني من الشكل السابق، نقوم بإستخدام **علامة الطرح** الرمز - قبل **علامة النجمة**، ليصبح الكود; `printf("%-7d i value\n", i);` أو `printf("%-*d i value\n", 7, i);` وهو رمز خاص من الرموز المسماه بـ **الأعلام أو Flags**، وهي مجموعة من الرموز الخاصة التي تستخدمن للتحكم في تنسيق الطباعة، والجدول التالي يوضح مجموعة الرموز التي يمكن أن يتم إستخدامها كـ **أعلام**، وشرح لإستخداماتهم.

جدول حقل الأعلام Flags Field

الرمز	الاستخدام
-	تستخدم علامة الطرح أو <b>Minus</b> . تغيير المحذاة الافتراضية لتصبح من اليسار إلى اليمين، كما تعرفنا في المثال السابق.
+	علامة الجمع أو <b>Plus</b> . تستخدم مع الأرقام الصحيحة أو الكسرية ليتم طباعة الإشارة الموجبة أو السالبة قبل الرقم، بالنسبة للأرقام الموجبة يتم طباعتها كما هي بدون إشارة، ولكن عند إضافة الرمز + يتم إضافة خانة الإشارة قبل طباعة الرقم، إذا كان الرقم سالب سوف يتم طباعة الإشارة السالبة كما هي.
Space	علامة المسافة أو <b>Space</b> . تستخدم مع الأرقام الموجبة ليتم طباعة مسافة قبل الرقم ، وعادة ما تستخدم لتنسيق طباعة الأرقام حيث تترك خانة قبل الرقم الموجب ليكون الرقم بمحاذاة أي رقم سالب في السطر السابق أو اللاحق.
0	رمز صفر أو <b>Zero</b> . يستخدم مع الرمز الخاص بالتحكم في العرض، حيث يعتبر الوضع الافتراضي لتنسيق عرض الطباعة هو إضافة مسافات فارغة قبل قيمة المتغير، فإذا كان المتغير عددي وأردت طباعة أصفار على يسار الرقم، يمكنك استخدام الرمز 0 للتنفيذ هذه المهمة.
#	رمز الهاش أو الشباك أو <b>Hash</b> ، ويستخدم مع مجموعة من الحروف <code>\#</code> حيث يقوم بالحفظ على طباعة الأصفار من اليمين في الأرقام العشرية، والوضع الافتراضي هو إزالة هذه الأصفار.
#	الحروف <code>f</code> و <code>g</code> و <code>E</code> و <code>e</code> و <code>G</code> ، يتم دائمًا طباعة العلامة العشرية.
#	الحروف <code>x</code> و <code>X</code> . يتم طباعة الرموز 0 و 0x و 0X على التوالي، قبل طباعة قيمة الرقم، فالوضع الافتراضي لهذه الحروف هو طباعتها كما هي، وجرى العرف على التمييز بين الأنظمة العددية عن طريق كتابة رمز معبر عن النظام العددي قبل الرقم.

الدالة `printf` تستخدم طريقة استدعاء الدوال العادية، لذلك قد يكون استعمالها اسهل بكثير من استعمال الدالة `std::cout` التي تستعمل `syntax` مختلفاً عن الدالة `printf` ، حيث يتم استعمال المعامل `<>` ، لذلك قد يكون استعمال الدالة `printf` اقصر في الكتابة ، لكن هذا الأمر لن يكون ملاحظاً ، حيث انه ليس فرق كبير ، لكن الفرق يظهر بشكل اكبر عندما تحاول ان تقوم بطباعة اكثر من قيمة في مرة واحدة ، حيث ان الدالة `printf` تسمح لك بتنسيق النص قبل طباعته (استخدام بعض القيم مثل `d` و `s` وتمرير متغيرات او قيم اخرى لتحل مكانها ) ، أيضاً سوف يصبح الامر اكثر تعقيداً من هذا بكثير عندما تحاول ان تقوم بطباعة متغيرات بقيم مختلفة مثل الأرقام بنظام `hex` او `octa` - مثال :

```
#include <iomanip>

printf("0x%05x\n", 0x3e3);
std::cout << "0x" << std::hex << std::setfill('0') << std::setw(5) << 0x3e3 << endl;
```

```
#include <iostream>
#include <cstdio>

int main()
{
    int a = 1, b = 10;
    printf("The irst number is: %d and The second number is %d \n", 10, 20);

    printf("Frist number '%d' + Second number '%d' = %d \n", a, b, a + b);

    //Print string and int variable
    int Page = 1, TotalPage = 10;
    printf("The page number = %d \n", Page);
    printf("You are in page %d of %d \n", Page, TotalPage);

    //Width specification
    printf("The page number = %0*d \n", 2, Page);
    printf("The page number = %0*d \n", 3, Page);
    printf("The page number = %0*d \n", 4, Page);
    printf("The page number = %0*d \n", 5, Page);

    for (int i = 1; i <= b; i++)
    {
        printf("the numbers is: %0*d \n", 2, i);
    }
}
```



# Lesson #24 - Float Format - printf

يوجد مجموعة محددة من الحروف التي يمكن استخدامها مع الرمز % للتحكم في طريقة طباعة قيم المتغيرات، حيث يمثل كل حرف نوع محدد من البيانات، والجدول التالي يوضح الحروف المحددة للاستخدام مع الرمز % ونوع البيانات الخاص بكل حرف.

الصيغة	الاستخدام	الرمز
%%	يستخدم هذا الرمز ليتم طباعة الرمز المئوي كما هو، حيث أن هذا الرمز يعتبر رمز خاص بالنسبة للدالة، فلا يمكن كتابته كما هو إلا إذا قمت بكتابته مرتين.	%
%d, %i	يستخدم الحرف d أو الحرف i لطباعة الأعداد الصحيحة والتي تمثل في نوع البيانات int، كلاهما يقومان بطباعة الأعداد الصحيحة في الدالة printf دون أي اختلاف، ولكن يختلفان في المهمة إذا تم استخدامهم في الدالة scanf، وهي الدالة التي تستخدم في عملية إدخال البيانات المنسقة، حيث عند استخدام حرف d مع الدالة scanf يتم قراءة الرقم الصحيح المدخل بالتمثيل الثنائي إذا كان مسبوقاً بالرقم 0، أو بالتمثيل السادس عشر Hexadecimal إذا تم إدخال الرقم مسبوقاً بـ 0x وهي طريقة كتابة البيانات بالنظام السادس عشر.	d, i
%u	يستخدم الحرف u لطباعة الأعداد الصحيحة التي لا تحتوي على إشارة، والتي تمثل في نوع البيانات unsigned int، فكما تعرف أن الأعداد يتم تمثيلها باستخدام نظام المتمم الثنائي (Two's Complement)، بما يعني أن الرقم السالب يتم حفظه في الذاكرة على هيئة المتمم الثنائي لنفس الرقم الموجب، وبالتالي يمكننا استخدام الحرف u، لطباعة الرقم كما تم تمثيله في الذاكرة.	u
%c	يستخدم الحرف c لطباعة البيانات من النوع char والتي تمثل حرف واحد أو رمز خاص من مجموعة الحروف ASCII.	c
%s	يستخدم الحرف s لطباعة نص يحتوي على أكثر من حرف، يعرف هذا النوع من البيانات في معظم لغات البيانات في لغة البرمجة string، ولكن لغة البرمجة C لا يوجد بها هذا النوع من البيانات، ولكي يتم كتابة نص يحتوي على أكثر من حرف في لغة البرمجة C يتم استخدام مصفوفة Array من النوع char على أن يكون آخر حرف في هذا النص هو الرمز \0 أو كما يعرف بـ NULL. وهذا النوع من البيانات يسمى Null Terminated String.	s
%f, %F	يستخدم الحرف f أو F لطباعة الأعداد الطبيعية والتي تحتوي على أرقام عشرية، ويعرف هذا النوع من البيانات بـ float أو double، وهو الأكثر استخداماً، والفرق بين حرف f أو F والحرف الكبير يتمثل في طباعة الرقم ما لاهيا Infinity والرقم NaN والمعرف بـ Not a Number، حيث الحرف الصغير يقوم بطباعة الكلمات السابقة بحروف صغيرة Small أما الحرف الكبير F فيقوم بطباعة الكلمات بحروف كبيرة Capital.	f, F
%e, %E	يستخدم الحرف e أو E لطباعة الأعداد العشرية من النوع float أو double بالطريقة العلمية المعروفة بـ Scientific Notation، وهي عبارة عن كتابة الأعداد العشرية مرفوعة لأس يمثل عدد الخانات العشرية، وهي الطريقة المتبعة لتمثيل البيانات العشرية و المعروفة بـ IEEE-32 و IEEE-64.	e, E
%g, %G	يستخدم الحرف g أو G لطباعة الأعداد العشرية من النوع float أو double، حيث يتم طباعة الأعداد بالطريقة العادي fixed-point إذا كان يمكن كتابة العدد بهذه الطريقة، فإذا كان العدد أكبر من أن يتم طباعته بهذه الطريقة يتم طباعة العدد باستخدام طريقة scientific notation، والفرق بين الحرف الصغير والكبير يتمثل في طباعة رمز الأس في العدد إما بحرف e صغير أو حرف E كبير في حالة ما تم طباعة العدد بالطريقة الثانية.	g, G
%x, %X	يستخدم الحرف x أو X لطباعة الأعداد الصحيحة التي لا تحتوي على إشارة أي من النوع unsigned int أو hexadecimal، بطريقة التمثيل العددي السادس عشر.	x, X
%a, %A	يستخدم الحرف a أو A لطباعة الأعداد العشرية من النوع float أو double بطريقة تمثيل الأعداد بالنظام السادس عشر، والفرق بين الحرف الكبير والصغير يتمثل في طباعة الرمز 0x السابق للعدد إما بحرف كبير أو صغير.	a, A
%o	يستخدم الحرف o لطباعة الأعداد الصحيحة بدون إشارة من النوع unsigned int بطريقة تمثيل الأعداد بالنظام الثنائي Octal.	o
%n	يستخدم الحرف n لتسجيل عدد الأحرف التي تم طباعتها بنجاح عن طريق حفظها داخل Pointer يتم تمريره للدالة، ولا يتم طباعة أي نتيجة عن هذه العملية، وتعد هذه العملية من الثغرات الأمنية في الدالة printf لذلك لا ينصح باستخدامها في البرامج.	n

**الصيغة %f , %F :** يستخدم الحرف f أو F لطباعة الأعداد الطبيعية والتي تحتوي على أرقام عشرية، ويعرف هذا النوع من البيانات بـ float أو double وهو الأكثر استخداماً، والفرق بين حرف f الصغير والحرف الكبير يتمثل في طباعة الرقم ما لاهيا Infinity والرقم NaN والمعرف بـ Not a Number، حيث الحرف الصغير يقوم بطباعة الكلمات السابقة بحروف صغيرة Small أما الحرف الكبير F فيقوم بطباعة الكلمات بحروف كبيرة Capital .

double PI = 3,1415926535898;

\* كم خانة تظهر بعد النقطة العشرية

```
printf("This number is : %.1f \n", 1, PI);
printf("This number is : %.2f \n", 2, PI);
printf("This number is : %.3f \n", 3, PI);
printf("This number is : %.4f \n", 4, PI);
```

```
This number is : 3.1
This number is : 3.14
This number is : 3.143
This number is : 3.1426
```

printf("This number is : %.5f \n", PI);

This number is : 3.14259

float X = 7.0, Y = 9.0;

printf("\nThe Div num : %.3f / %.3f = %.3f", X, Y, X / Y); The Div num : 7.000 / 9.000 = 0.778

```

#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
    double PI = 3.1415926535898;

    //Precision Specification   مواصفات الدقة
    printf("Precision Specification of %.1f \n", 1, PI);
    printf("Precision Specification of %.2f \n", 2, PI);
    printf("Precision Specification of %.3f \n", 3, PI);
    printf("Precision Specification of %.4f \n", 4, PI);
    printf("Precision Specification of %.5f \n", 5, PI);

    float Num1 = 6.2, Num2 = 4.4;
    printf("\nThe float division is: %.3f / %.3f = %.3f \n\n", Num1, Num2, Num1 / Num2);

    double D = 34.32;
    printf("The value number is: %.3f \n", D);
    printf("The value number is: %.4f \n\n", D);

    // يسمى هذا فرومات للذى سيظهر على الشاشة اما القيم فهو ثابتة لا تختلف في الكود
    // %.1f or %.2f = used with float and double

    system("pause");
}

```

```

Precision Specification of 3.1
Precision Specification of 3.14
Precision Specification of 3.142
Precision Specification of 3.1416
Precision Specification of 3.14159

The float division is: 6.200 / 4.400 = 1.409

The value number is: 34.320
The value number is: 34.3200

```



## Lesson #25 - String and Char Format - printf

- الدالة **printf** لا تتعامل مع **نوع البيانات من نوع String**
- لكن تتعامل مع الـ **Character Arrays**
- الـ **Array Of Characters** عبارة عن **String**
- مع تطورات الـ **C++** ظهرت الـ **String Object** - والأخير ليس **Supported** مع **Printf**
- ولكن **Supported** مع **Character Array**
- **الصيغة %c** : يستخدم الحرف **c** لطباعة البيانات من النوع **char** والتي تمثل حرف واحد أو رمز خاص من مجموعة **الحروف ASCII**.
- **الصيغة %s** : يستخدم الحرف **s** لطباعة نص يحتوي على أكثر من حرف، يعرف هذا النوع من البيانات في معظم لغات البرمجة **b**، ولكن الدالة **printf** لا تتعامل مع هذا النوع من البيانات، ولكي يتم كتابة نص يحتوي على أكثر من حرف في لغة البرمجة **C++** يتم استخدام مصفوفة **Array** من النوع **char**.

```
char C = 'A';
char str[] = "Ahmad ElSayed";
```

```
printf("Character c = %c and inline char = %c\n", C, 'B');
// Character c = A and inline char = B
```

```
printf("String str = %s and inline string = %s\n", str, "AHMED");
// String str = Ahmad ElSayed and inline string = AHMED
```

في هذا المثال إستخدمنا الرمز **%c** لطباعة حرف واحد من النوع **char** ، حيث طبعنا أولاً قيمة المتغير **c** ثم قمنا بتمرير الحرف **B** للدالة ليتم طباعته بدليلاً للتكرار الثاني للرمز **%c** ، وسوف تلاحظ أننا قمنا بتمرير الحرف **B** بين علامة تنسيص مفردة '**'** **Single-Quote** وهو الطريقة المتبعة مع الرموز من النوع **char** في لغة البرمجة **C** .

وفي الدالة الثانية قمنا بطباعة سلسلة من الحروف أو ما يعرف بـ **string** أو **char Array** ، بإستخدام الرمز **%s** ، وقمنا أيضاً بتمرير قيمة نصية ثابته وفي حالة السلسلة يتم كتابة النص بين علامة التنسيص المزدوجة "**"** **Double-Quote** " .

**الصيغة %c\*** او **%3c** تستخدم الـ \* او 3 لاضافة **مسافة فارغة واحدة** قبل قيمة المتغير

```
char c = 'S';
printf("Setting the width of c : %*c \n", 1, c);
Setting the width of c : S
```

```
char Lava = 'A';
printf("The 3 number is: %3c \n", Lava);
The 3 number is: A
```

```

#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
    char Name[] = "Ahmed AbdelRahim";
    char SchoolName[] = "Programming Advices";

    printf("Dear, %s, How are you? \n", Name);
    printf("Welcome to %s school! \n\n", SchoolName);

    char c = 'S';
    //Precision Specification  **c
    printf("Setting the width of c :%*c \n", 1, c);
    printf("Setting the width of c :%*c \n", 2, c);
    printf("Setting the width of c :%*c \n", 3, c);
    printf("Setting the width of c :%*c \n", 4, c);
    printf("Setting the width of c :%*c \n", 5, c);

    system("pause");

    char NameA[] = "Ahmed ElSayed";
    char OfficeName[] = "HORUS";

    printf("\nYour name is: %s \n", NameA);
    printf("You are work in: %s \n\n", OfficeName);

    char Lava = 'A';

    printf("The 1 number is: %*c \n", 1, Lava);
    printf("The 2 number is: %*c \n", 2, Lava);
    printf("The 3 number is: %*c \n", 3, Lava);
    printf("The 4 number is: %*c \n", 4, Lava);
    printf("The 5 number is: %*c \n\n", 5, Lava);
}

```

Dear, Ahmed AbdelRahim, How are you?  
Welcome to Programming Advices school!

Setting the width of c :S  
Setting the width of c : S  
Setting the width of c : S  
Setting the width of c : S  
Setting the width of c : S

Your name is: Ahmed ElSayed  
You are work in: HORUS

The 1 number is: A  
The 2 number is: A  
The 3 number is: A  
The 4 number is: A  
The 5 number is: A



# Lesson #26 - Set Width (SetW) Manipulator

## (Set Width) - "تعيين العرض" (setw)

تُستخدم طريقة **setw()** من مكتبة **ios** في لغة C++ لتعيين عرض حقل مكتبة **iomanip** بناءً على العرض المحدد كمعامل لهذه الطريقة. حيث تشير **setw()** إلى "تعيين العرض" (set width) وتعمل مع كل من التدفقات (streams) الداخلية والخارجية.

تتوارد هذه الطريقة في مكتبة **iomanip** : يجب تضمين هذه المكتبة في الكود باستخدام السطر التالي:  
**#include <iomanip>**

Syntax: **std::setw(int n);**

Parameters: **n:** It is the integer argument corresponding to which the field width is to be set.

الشرح التفصيلي للعبارة فهو كالتالي:

المعامل الصحيح (**integer argument**): هو متغير عدد صحيح يتم تمريره إلى دالة أو إجراء.

عرض الحقل (**field width**): هو عدد الأحرف التي يجب أن تشغلها قيمة معينة عند عرضها.

المعامل الصحيح الذي يتم تمريره إلى الدالة أو الإجراء هو الذي يحدد العرض الذي يجب تعينه.

تأخذ وسيطًا (**parameter**) يحدد العرض المطلوب: يتم تمرير العرض المطلوب كعدد صحيح إلى طريقة **setw()**.

تعمل مع كل من التدفقات الداخلية والخارجية: يمكن استخدامها لضبط عرض الحقول عند قراءة البيانات من ملفات الإدخال، وكذلك عند كتابة البيانات إلى ملفات الإخراج أو إلى وحدة التحكم (**console**).

تؤثر على العرض التالي فقط: يستمر تأثير **setw()** على العرض التالي فقط. فإذا أردت ضبط العرض للعديد من الحقول، يجب استدعاء **setw()** قبل كل حقل.

```
8
9 #include <iostream>
10 #include <iomanip>
11
12 using namespace std;
13
14 int main()
15 {
16     int num = 123;
17     std::cout << "\n" << std::setw(10) << num << std::endl;
18
19     return 0;
20 }
```

## Output

123

في هذا المثال:

تم ضبط العرض إلى 10 باستخدام **setw(10)**.

يتم تمرير المعامل الصحيح 10 إلى الدالة (**std::setw()**). هذا يعني أن الدالة ستضبط العرض إلى 10 أحرف. عند طباعة الرقم 123، سيتم إضافة 7 مسافات بادئة حتى يصل العرض الإجمالي إلى 10 أحرف.

```

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout << "======" << endl;
    cout << " | " << setw(5) << "M" << setw(5) << " | " << setw(15) << "Name" << setw(15) << " | " << endl;
    cout << "======" << endl;

    cout << " | " << setw(5) << "1" << setw(5) << " | " << setw(15) << "Ahmad" << setw(15) << " | " << endl;
    cout << " | " << setw(5) << "2" << setw(5) << " | " << setw(15) << "ElSayed" << setw(15) << " | " << endl;

    cout << " | " << setw(9) << "1" << " | " << setw(29) << "Ahmad" << " | " << endl;
    cout << " | " << setw(9) << "2" << " | " << setw(29) << "ElSayed" << " | " << endl;

    cout << " | " << left << setw(9) << "1" << " | " << setw(29) << "Ahmad" << " | " << endl;
    cout << " | " << left << setw(9) << "2" << " | " << setw(29) << "ElSayed" << " | " << endl;

    return 0;
}

```

=====	
1	Ahmad
2	ElSayed
1	Ahmad
2	ElSayed
1	Ahmad
2	ElSayed
=====	

## SetW Using Right alignment

```
#include <iostream>
#include <iomanip> // Setw يجب تضمين هذه المكتبة عند استعمال
using namespace std;

void SetW()
{
    cout << endl;

    cout << " | " << "====" << " | " << "===== " << " | " << endl;
    cout << " | " << setw(3) << "M" << setw(3) << " | " << setw(9) << "Name" << setw(7) << " | " << endl;
    cout << " | " << "====" << " | " << "===== " << " | " << endl;
    cout << " | " << setw(5) << 1 << " | " << setw(15) << "Ahmed" << " | " << endl;
    cout << " | " << setw(5) << 2 << " | " << setw(15) << "ElSayed" << " | " << endl;
    cout << " | " << setw(5) << 3 << " | " << setw(15) << "AbdelRahim" << " | " << endl;
    cout << " | " << setw(5) << 4 << " | " << setw(15) << "Office" << " | " << endl;
    cout << " | " << setw(5) << 10 << " | " << setw(15) << "of" << " | " << endl;
    cout << " | " << setw(5) << 100 << " | " << setw(15) << "Horussssss" << " | " << endl;
    cout << " | " << "====" << " | " << "===== " << " | " << endl << endl;
}

int main()
{
    system("color f0");
    SetW();
    system("pause");
}
```

====	=====
M	Name
====	=====
1	Ahmed
2	ElSayed
3	AbdelRahim
4	Office
10	of
100	Horussssss
====	=====

## SetW Using left alignment

```
#include <iostream>
#include <iomanip> // SetW
using namespace std;

void SetW()
{
    cout << endl;

    cout << "====" << "|" << "======" << "|" << endl;
    cout << "|" << setw(3) << "M" << setw(3) << "|" << setw(9) << "Name" << setw(7) << "|" << endl;
    cout << "====" << "|" << "======" << "|" << endl;
    cout << "|" << left << setw(5) << 1 << "|" << setw(15) << "Ahmed" << "|" << endl;
    cout << "|" << left << setw(5) << 2 << "|" << setw(15) << "ElSayed" << "|" << endl;
    cout << "|" << left << setw(5) << 3 << "|" << setw(15) << "AbdelRahim" << "|" << endl;
    cout << "|" << left << setw(5) << 4 << "|" << setw(15) << "Office" << "|" << endl;
    cout << "|" << left << setw(5) << 10 << "|" << setw(15) << "of" << "|" << endl;
    cout << "|" << left << setw(5) << 100 << "|" << setw(15) << "Horussssss" << "|" << endl;
    cout << "====" << "|" << "======" << "|" << endl << endl;
}

int main()
{
    system("color f0");
    SetW();
    system("pause");
    return 0;
}
```

M	Name
1	Ahmed
2	ElSayed
3	AbdelRahim
4	Office
10	of
100	Horusssss



# Lesson #27 - Two Dimensional Arrays

## المصفوفات الثنائية Two Dimensional Array

المصفوفة ثنائية الأبعاد في لغة C++ هي مجموعة من العناصر مرتبة في صفوف وأعمدة. يمكن تصورها كجدول أو شبكة، حيث يتم الوصول إلى كل عنصر باستخدام مؤشرين (فهرسين): واحد (فهرس) للصف والآخر (فهرس) للعمود. كما هو الحال في المصفوفة أحادية البعد، تتراوح مؤشرات (فهارس) المصفوفة ثنائية الأبعاد أيضاً من 0 إلى  $n-1$  لكل من الصفوف والأعمدة.

**Syntax of 2D array :** `data_Type array_name[Rows][Cols];`

**Rows :** Number of Rows.

**Cols :** Number of Columns.

إليك توضيح بالصور:

```
#include <iostream>
using namespace std;

int main()
{
    //int x[Rows][Cols];
    int x[3][4] = {
        {1,2,3,4},
        {5,6,7,8},
        {9,10,11,12}
    };

    return 0;
}
```

	Col 0	Col 1	Col 2	Col 3
Row 0	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 1	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 2	x[2][0]	x[2][1]	x[2][2]	x[2][3]

	Col 0	Col 1	Col 2	Col 3
Row 0	→ 1	→ 2	→ 3	→ 4
Row 1	→ 5	→ 6	→ 7	→ 8
Row 2	→ 9	→ 10	→ 11	→ 12

## الشرح:

**العناصر (Elements):** تتكون المصفوفة ثنائية الأبعاد من مجموعة من العناصر من نفس النوع، مثل الأرقام الصحيحة أو الأحرف أو القيم العائمة.

**صفوف وأعمدة:** يتم ترتيب هذه العناصر في شكل جدول له صفوف وأعمدة.

**الصفوف (Rows):** هي المجموعات الأفقية من العناصر.

**الأعمدة (Columns):** هي المجموعات العمودية من العناصر.

**فهرسان:** للوصول إلى عنصر معين في المصفوفة، تحتاج إلى تحديد فهرس الصف وفهرس العمود الذي يقع فيه العنصر.

**المؤشر (الفهرس) الأول (First index):** يحدد رقم الصف، ويبدأ من **0**.

**المؤشر (الفهرس) الثاني (Second index):** يحدد رقم العمود، ويبدأ أيضاً من **0**.

**بدء الفهارس من 0:** تبدأ فهارس الصفوف والأعمدة من **0** وتنتهي عند **n-1**، حيث **n** هو عدد الصفوف أو الأعمدة في المصفوفة.

**التصور كجدول:** يمكن تصور المصفوفة ثنائية الأبعاد كجدول أو شبكة، حيث يمثل كل صف سطراً في الجدول، ويمثل كل عمود عموداً في الجدول.

## تهيئة المصفوفات ثنائية الأبعاد في لغة C++

توجد طرق مختلفة لتهيئة مصفوفة ثنائية الأبعاد، منها ما يلي:

### 1. استخدام قائمة التهيئة (Initializer List):

تسمح هذه الطريقة بتعيين **قيم أولية للعناصر** عند إعلان المصفوفة.

يتم ذلك باستخدام **قوسين معقوقين {}**، حيث يتم وضع القيم داخلها مفصولة بفواصل.

تكتب القيم لكل صف داخل أقواس معقوقة خاصة بها.

مثال:

```
int myArray[3][4] = {  
    {1, 2, 3, 4} // قيم الصف الأول  
    {5, 6, 7, 8} // قيم الصف الثاني  
    {9, 10, 11, 12} // قيم الصف الثالث  
};
```

### 2. استخدام الحلقات (Loops) وهذه المفضلة:

يمكن استخدام الحلقات لتعيين **قيم العناصر بشكل فردي** بعد إعلان المصفوفة.

غالباً ما تُستخدم حلقتين متداخلتين (nested loops)، عادةً من نوع **for**، للمرور عبر جميع عناصر (كل صف وعمود) في المصفوفة سواء لإدخال قيم فيها أو للحصول على قيمها.

تُستخدم **الحلقة الخارجية** للصفوف (تنقلك من صف أو سطر لآخر)، **والحلقة الداخلية** للأعمدة (تمر على كل العناصر الموجودة في الصف أو السطر).

مثال:

```
int myArray[3][4];  
for (int i = 0; i < 3; i++)  
{  
    for (int j = 0; j < 4; j++)  
    {  
        myArray[i][j] = i * 4 + j + 1; // تعيين قيمة حسابية  
    }  
}
```

## حجم المصفوفة متعددة الأبعاد

حجم المصفوفة يساوي حجم نوع البيانات مضروباً في إجمالي عدد العناصر التي يمكن تخزينها في المصفوفة. يمكننا حساب إجمالي عدد العناصر في المصفوفة عن طريق ضرب حجم كل بعد من أبعاد المصفوفة متعددة الأبعاد.

إليك شرح أكثر تفصيلاً:

**حجم نوع البيانات:** يختلف حجم كل نوع بيانات في الذاكرة. على سبيل المثال، يشغل عدد صحيح (int) عادةً 4 بait، بينما يشغل حرف (char) بait واحد.

**إجمالي عدد العناصر:** يعتمد إجمالي عدد العناصر التي يمكن تخزينها في المصفوفة على حجمها ونوع البيانات التي تخزنها.

على سبيل المثال: في مصفوفة ثنائية الأبعاد ذات 5 صفوف و 3 أعمدة، يكون إجمالي عدد العناصر هو  $5 * 3 = 15$ .

مثال:

افتراض أن لدينا مصفوفة ثنائية الأبعاد من النوع int، بأبعاد 3 × 4:

int arr[3][4];

حجم كل عنصر (int) هو 4 بait.

اجمالي عدد العناصر في المصفوفة هو (3 صف × 4 اعمدة) = 12 عنصراً.

حجم المصفوفة الإجمالي هو 4 بait × 12 عنصراً = 48 بait.

بشكل عام يمكن حساب حجم مصفوفة متعددة الأبعاد باستخدام هذه الصيغة:

حجم المصفوفة = حجم نوع البيانات × (عدد العناصر في البعد الأول) × (عدد العناصر في البعد الثاني) × ... × (عدد العناصر في البعد الأخير)

```

#include <iostream>
#include <cstdio>
#include<iomanip>

using namespace std;

void ArrayExample()
{
    //int x[Rows][Cols]

    int x[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };

    for (int i = 0; i < 3; i++)           // to rows = الصفوف
    {
        cout << "\n\t";
        for (int j = 0; j < 4; j++)      // to columns = اعمدة
        {
            cout << setw(3) << x[i][j] << " ";
        }
        cout << endl;
    }
}

int main()
{
    system("color f0");
    ArrayExample();
    cout << endl;
    system("pause");
}

```

1	2	3	4
5	6	7	8
9	10	11	12

**Exam: Use two dimensional arrays to Write a program to store the multiplication table results 10 x 10 and print them on the screen**

**Answer 1 (Me):**

```
#include <iostream>
#include <cstdio>
#include <iomanip>

using namespace std;

void ReadAndPrintTwoDimeArr()
{
    int TwoArr[10][10];
    cout << "\n";
    for (int i = 0; i < 10; i++)
    {
        cout << "\t";
        for (int j = 0; j < 10; j++)
        {
            TwoArr[i][j] = (i + 1) * (j + 1);
            printf("%02d ", TwoArr[i][j]);
        }
        cout << endl;
    }
}

int main()
{
    system("color f0");
    ReadAndPrintTwoDimeArr();
    cout << endl;
    system("pause");
}
```

```
01 02 03 04 05 06 07 08 09 10
02 04 06 08 10 12 14 16 18 20
03 06 09 12 15 18 21 24 27 30
04 08 12 16 20 24 28 32 36 40
05 10 15 20 25 30 35 40 45 50
06 12 18 24 30 36 42 48 54 60
07 14 21 28 35 42 49 56 63 70
08 16 24 32 40 48 56 64 72 80
09 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

### Answer 3 (AbouHadhod):

```
#include <iostream>
#include <cstdio>
#include <iomanip>

using namespace std;

int main()
{
    //int x[Rows][Cols]
    system("color f0");
    int x[10][10];

    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            x[i][j] = (i + 1) * (j +1);
        }
    }

    for (int i = 0; i < 10; i++)
    {
        cout << "\n\t";
        for (int j = 0; j < 10; j++)
        {
            printf("%02d ", x[i][j]);
        }
        cout << "\n";
    }
    cout << endl;
    system("pause");
}
```

01	02	03	04	05	06	07	08	09	10
02	04	06	08	10	12	14	16	18	20
03	06	09	12	15	18	21	24	27	30
04	08	12	16	20	24	28	32	36	40
05	10	15	20	25	30	35	40	45	50
06	12	18	24	30	36	42	48	54	60
07	14	21	28	35	42	49	56	63	70
08	16	24	32	40	48	56	64	72	80
09	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100



# Lesson #28 - Introduction , Declaration and Initialization

## أهم الدوال في حاوية Vector

الدالة	استخدامها
<code>push_back(val)</code>	إضافة عنصر جديد في نهاية <code>(vector)</code> ، يؤدي هذا إلى زيادة حجم الحاوية بمقدار واحد. وإذا تجاوز عدد العناصر في <code>(vector)</code> عن السعة الحالية، فإنه يؤدي إلى توسيع حجم التخزين تلقائياً.
<code>pop_back()</code>	يزيل العنصر الأخير في <code>(vector)</code> ، مما يقلل حجم الحاوية بمقدار واحد .
<code>begin()</code>	يُرجع مؤشر <code>(Pointer)</code> ، يشير إلى موقع أول عنصر في <code>(vector)</code> ، و تستطيع من خلاله التنقل عبر محتويات <code>(vector)</code> .
<code>end()</code>	يُرجع مؤشر يشير إلى موقع الذي يلي العنصر الأخير في <code>(vector)</code> ، وبالتالي لا يشير إلى أي عنصر في <code>(vector)</code> .
<code>rbegin()</code>	يُرجع مؤشر <code>(Pointer)</code> ، يشير إلى موقع آخر عنصر في <code>(vector)</code> ، ويعني <code>(reverse begin)</code> أي بداية عكسية.
<code>rend()</code>	يُرجع مؤشر يشير إلى موقع الذي يسبق العنصر الأول في <code>(vector)</code> ، وبالتالي لا يشير إلى أي عنصر في <code>(vector)</code> ، ويعني <code>(reverse end)</code> أي نهاية عكسية.
<code>size()</code>	إرجاع عدد العناصر الفعلية في <code>(vector)</code>
<code>resize(n, val)</code>	يغير حجم الحاوية إلى الحجم <code>(n)</code> ، فإذا كان <code>(n)</code> أصغر من حجم الحاوية الحالي، يتم نقل المحتوى إلى أول <code>(n)</code> من العناصر، وإزالة العناصر المتبقية، أما إذا كان <code>(n)</code> أكبر من حجم الحاوية الحالي، فسيتم توسيع المحتوى إلى حجم <code>(n)</code>
<code>empty()</code>	تحتبر ما إذا كانت الحاوية فارغة أم لا ، فإذا كانت الحاوية فارغة (أي ما إذا كان حجمها صفر)، تُرجع <code>(true)</code> غير ذلك <code>(false)</code>
<code>max_size()</code>	يُرجع الحد الأقصى لعدد العناصر التي يمكن أن يحويها <code>(vector)</code>
<code>swap(x)</code>	يتبادل محتوى <code>(vector)</code> الذي تم الاستدعاء به بمحتويات <code>(x)</code> ، وهي <code>(vector)</code> أخرى.
<code>erase(position)</code>	يزيل عنصر من <code>(vector)</code> أو، سلسلة عناصر.
<code>erase(first, last)</code>	إذا أردته أن تزيل عنصر فإِنَّك تضع موقع العنصر المراد حذفه، وأردت أن تزيل سلسلة محددة من العناصر فإِنَّك تضع موقع العنصر الأول المراد حذفه، وموقع العنصر الأخير
<code>clear()</code>	مسح جميع العناصر داخل <code>(vector)</code> .

## دوال للوصول لعنصر محدد في حاوية Vector

الدالة	استخدامها
<code>[n]</code>	كما المصفوفة، فإنها ترجع محتوى المصفوفة في الموقع <code>(n)</code> في <code>(vector)</code>
<code>front()</code>	يُرجع أول عنصر في <code>(vector)</code>
<code>back()</code>	يُرجع آخر عنصر في <code>(vector)</code>

💡 Vectors are used to store elements of similar data types. However, unlike arrays, the size of a vector can grow dynamically.

That is, we can change the size of the vector during the execution of a program as per our requirements.

💡 تُستخدم المتجهات (**vectors**) لإنشاء كائن يمثل حاوية لتخزين العناصر التي نضيفها فيها بشكل متسلسل وراء بعضها البعض مع إعطاء كل عنصر منهم رقم **Index**. وتكون هذه العناصر من **نفس نوع البيانات** ( - **integer - string** - **double**). ولكن على عكس المصفوفات (الازم تحدد حجمها قبل ما تستخدمنا)، يمكن أن يتغير حجم الـ **vector** بشكل ديناميكي أثناء **تنفيذ البرنامج** في الـ **run time** ، وذلك حسب متطلباتنا. (الحجم بيكبر ويصغر حسب الذاكرة من غير ما احجز اماكن زيادة).

💡 لاستخدام الـ **vector** يجب تضمين الملف `#include <vector>` لأنه موجود فيه.

💡 إليك شرح مبسط :

- **تخزين عناصر من نفس النوع**: مثل المصفوفات، تُستخدم المتجهات لتخزين مجموعة من العناصر التي تنتمي إلى نفس النوع، مثل: **Integer, String, Double**.

- **الحجم الديناميكي**: بينما يكون حجم المصفوفة ثابتاً بعد إنشائها، يمكن **زيادة أو تقليل حجم الـ Vector** أثناء تشغيل البرنامج (Run Time).

- **إدارة الذاكرة تلقائياً**: تتعامل الـ **Vectors** مع تخصيص الذاكرة وإلغاء تخصيصها **تلقائياً**، لذا لا داعي للقلق بشأن إدارة الذاكرة يدوياً كما هو الحال مع المصفوفات.

💡 الـ **Vector** لا يأخذ الـ **Size** مثل الـ **Array** ... لكن يأخذ الـ **Size** من **initial values** (القيم الأولية)

💡 & لطباعة القيم نستخدم الـ **ranged loop** ومن المهم استخدام الـ **Ranged Loop** (&) مع الـ **Reference** (&) حتى لا يقوم بنسخ العناصر للذاكرة وهذا يأخذ وقت في تشغيل الكود وأيضاً يحجز مساحات كثيرة لا حاجة لها في الـ **Memory** ، حيث ان الـ **Reference** يأشر على موقع المتغير في الذاكرة ، فهكذا سيكون البرنامج أسرع وأخف.

💡 **المصفوفة الديناميكية**: هي نوع من بنية البيانات التي يمكن تغيير حجمها أثناء تشغيل البرنامج، على عكس المصفوفات التقليدية التي يكون حجمها ثابتاً بعد إنشائها.

💡 الـ **Vectors** كمصفوفات ديناميكية: تمتلك الـ **Vectors** في C++ هذه الخاصية، إذ يمكنها زيادة حجمها تلقائياً عند إضافة عناصر جديدة، أو تقليل حجمها عند حذف عناصر. وهذا يجعلها أكثر مرونة من المصفوفات التقليدية في التعامل مع مجموعات البيانات المتغيرة.

💡 **كيفية تغيير الحجم**: عندما يتم إضافة عنصر جديد إلى الـ **Vector**، فإنه يقوم بفحص ما إذا كانت هناك مساحة كافية في ذاكرته الحالية. إذا لم تكن هناك مساحة كافية، فإنه يقوم بإنشاء مصفوفة داخلية جديدة بحجم أكبر، ويقوم بنسخ جميع العناصر من المصفوفة القديمة إلى المصفوفة الجديدة، ثم يقوم بحذف المصفوفة القديمة. **هذه العملية تحدث بشكل تلقائي وسريع في معظم الحالات.**

💡 **الفوائد**:

- **المرونة**: تسمح الـ **Vectors** بإضافة وحذف عناصر بسهولة أثناء تشغيل البرنامج، مما يسهل التعامل مع مجموعات البيانات المتغيرة.

- **سهولة الاستخدام**: توفر الـ **Vectors** العديد من الدوال والعمليات التي تجعل من السهل التعامل معها، مثل إضافة عناصر جديدة، وحذف عناصر، والوصول إلى عناصر معينة، وتعديلها، وترتيبها، والبحث فيها.

- **الأداء الجيد**: في معظم الحالات، يكون أداء الـ **Vectors** جيداً جداً، حتى عند التعامل مع مجموعات بيانات كبيرة.

بشكل عام، تعتبر الـ **Vectors** من أهم هيكلات البيانات في لغة C++ ، وغالباً ما يتم استخدامها في العديد من التطبيقات المختلفة بسبب مرونتها وسهولة استخدامها وأدائها الجيد.

## ملخص الاختلافات الرئيسية بين الـ **Arrays** والـ **Vectors**

الـ <b>vector</b>	الـ <b>Array</b>	الميزة
لا	نعم	حجم ثابت
نعم	لا	فئة قابلية
تلقائية	يدوية	إدارة الذاكرة
أكثـر	أقل	سهولة الاستخدام
قد يكون أبطأ قليلاً في بعض الحالات	قد يكون أسرع في بعض الحالات	الأداء

### ؟لمحات:

- تشغـل الـ **vectors** مساحة أكبر مقارنة بالمصفوفات لأنـ الـ **vectors** تحتاج إلى مساحة إضافـية لتخـزين معلومات حول حجمـها الحالي وحجمـها الأقصـى. أما المصفـوفات فلا تحتاج إلى هذه المعلومات، لأنـ حجمـها ثابت بعد إنشـائـها.
- عمـليـات إعادة تـخصـيص الـ ذاـكرة: عندـا يتم إضافـة عـناـصـر جـديـدة إلى الـ **vector**، قد يتـطلـب ذلك إعادة تـخصـيص مسـاحة ذـاـكرة أـكـبر، مما يتـضـمن نـسـخـ البيانات المـوجـودـة إلى المـوقـعـ الجـديـد. هـذـه العمـليـة تستـغرـق وقتـاً إضافـياً.
- الوصول غير المباشر: في بعضـ الحالـات، قد لا تخـزن عـناـصـر الـ **vector** بشـكل متـجاـور في الـ ذـاـكرة، مما يـعـني أنـ الوصول إلى عـناـصـر معـين قد يتـطلـب الـ انتـقال عبر مؤـشـرات إضافـية، وهذا يستـغرـق وقتـاً أـطـول منـ الوصول المباشر إلى عـناـصـر المـصـفـوفـة.
- المـصـفـوفـات ثـابـتـة بـطـبيـعـتها، لـذـا يستـغرـق الوصول إلى أيـ عـناـصـر في المـصـفـوفـة باـسـتـخدـام عـاملـ الفـهـرـسة وقتـاً ثـابـتاً.
- عندما يتم إضافـة عـناـصـر جـديـدة إلى الـ **Vector**، فإـنه يـقـوم تـلـقـائـياً بـزيـادة حـجمـه لـاستـيعـاب العـناـصـر الجـديـدة. وعـنـدـما يتم حـذـف عـناـصـر، فإـنه يـقـوم تـلـقـائـياً بـتـقـليـص حـجمـه لـتحـريـز الـ ذـاـكرة غـير المستـخدمـة.
- ذـاـكرة كـوـمةـ البرـنـامـج (**Heap Memory**): هي منـطـقة منـ الـ ذـاـكرة مـخـصـصة لـتخـزينـ البياناتـ التيـ يتمـ إـنشـاؤـهاـ أـثنـاءـ تشـغـيلـ البرـنـامـجـ. عـلـى عـكـسـ الـ ذـاـكرةـ المـكـدـسـة (**Stack Memory**)، والـتيـ تـسـتـخدـمـ لـتخـزينـ المـتـغـيرـاتـ المـحلـيةـ والـدوـالـ، فـإـنـ ذـاـكرةـ الـ كـوـمةـ يـمـكـنـ أـنـ تـمـوـ وـتـنـقلـصـ حـسبـ الحاجـةـ أـثنـاءـ تشـغـيلـ البرـنـامـجـ.
- بـسـبـبـ هـذـهـ الخـصـائـصـ، تـعـتـبـرـ الـ **Vectors** خـيـارـاً مـثـالـياًـ عـنـدـماـ تـحـتـاجـ إـلـىـ الـ عـمـلـ معـ مـجـمـوعـةـ منـ الـ بـيـانـاتـ الـيـقـظـةـ قدـ يـتـغـيرـ حـجمـهاـ أـثنـاءـ تشـغـيلـ البرـنـامـجـ.
- عـنـدـ إـنشـاءـ مـصـفـوفـةـ، يـجـبـ تحـديـدـ حـجمـهاـ مـسـبـقاًـ. لا يـمـكـنـ تـغـيـيرـ حـجمـ المـصـفـوفـةـ بـعـدـ إـنشـائـهاـ. أماـ الـ **Vectors**ـ،ـ فـيـمـكـنـ تـغـيـيرـ حـجمـهاـ أـثنـاءـ تشـغـيلـ البرـنـامـجـ.
- عـنـدـماـ يـتـمـ إـضافـةـ عـناـصـرـ جـديـدةـ إلىـ الـ **Vector**ـ،ـ فـقـدـ يـكـونـ منـ الـ ضـرـوريـ تـخـصـيصـ مـسـاحةـ ذـاـكرةـ جـديـدةـ أـكـبرـ لـاستـيعـابـ عـناـصـرـ الـ جـديـدةـ. تـقـومـ الـ **Vectors**ـ بـإـجـراءـ إـعادـةـ تـخـصـيصـ الـ ذـاـكرةـ تـلـقـائـياـ فيـ هـذـهـ الـ حـالـةـ،ـ دونـ الحاجـةـ إـلـىـ تـدـخلـ الـ مـسـتـخـدـمـ.

## استخدامات الـ **Vectors** !

تتميز الـ **vectors** بفاءة عالية من حيث استخدام الذاكرة والأداء.

- تخزين مجموعات من البيانات التي قد يتغير حجمها أثناء تشغيل البرنامج.
- إنشاء هيكل بيانات أكثر تعقيداً، مثل القوائم المرتبطة والمكبسات والأشجار.
- التعامل مع البيانات التي يتم إدخالها من قبل المستخدم أو قراءتها من الملفات.
- إذا لم يكن حجم البيانات معروفاً قبل البدء، فلن يطلب منك الـ **Vector** تعين الحد الأقصى لحجم الحاوية.

بعض الأمثلة على استخدام الـ **vectors** لتخزين مجموعات من البيانات التي قد يتغير حجمها أثناء تشغيل البرنامج:

- تخزين قائمة المهام: يمكن استخدام الـ **vectors** لتخزين قائمة المهام التي يجب تنفيذها، حيث يمكن إضافة مهام جديدة إلى القائمة أو حذف مهام موجودة من القائمة دون الحاجة إلى إعادة تخصيص الذاكرة.
- تخزين سجل الأحداث: يمكن استخدام الـ **vectors** لتخزين سجل الأحداث التي تحدث في النظام، حيث يمكن إضافة أحداث جديدة إلى السجل أو حذف أحداث موجودة من السجل دون الحاجة إلى إعادة تخصيص الذاكرة.
- تخزين نتائج الحساب: يمكن استخدام الـ **vectors** لتخزين نتائج الحساب التي يتم الحصول عليها في برنامج، حيث يمكن إضافة نتائج جديدة إلى الـ **vector** أو حذف نتائج موجودة من الـ **vector** دون الحاجة إلى إعادة تخصيص الذاكرة.

## إعلان **Vector Declaration** !

بعد تضمين الملف الرأس (`#include <vector>`) ، يمكننا إعلان الـ **vector** في C++ بالطريقة التالية:

```
#include <vector>           // تضمين الملف الرأس اللازم
vector<int> myVector;      // من نوع int إعلان
vector<string> myStringVector; // من نوع string إعلان
vector<double> myDoubleVector; // من نوع double إعلان
```

شرح الإعلان:

`#include<vector>` هذا السطر يضمن الملف الرأس **vector** الذي يحتوي على تعريف الكائن **vector** ووظائفه.

- `vector<type> name;` هذه الصيغة العامة لإعلان **vector**.
- `type` : يحدد نوع البيانات التي سيخزنها الـ **vector**، مثل `int`، `string`، أو `double`، أو أي نوع آخر.
  - `name` : يحدد اسمًا للـ **vector**، والذي يمكن استخدامه للإشارة إليه لاحقًا في البرنامج.

بهذا تكون قد أنشأك **vector** فارغاً جاهزاً للاستخدام. يمكنك بعد ذلك إضافة العناصر إليه وتعديلها حسب احتياجاتك أثناء تشغيل البرنامج.

## تهيئة **Vector Initialization** !

هناك طرق مختلفة لتهيئة الـ **vector** في C++ ، وفيما يلي بعض منها:

1. تهيئة الـ **vector** باستخدام قائمة التهيئة (**initializer list**). تسمح هذه الطريقة بتعيين قيمة أولية للعناصر أثناء إعلان الـ **vector**. مثال:

```
#include <vector>

int main()
{
    تهيئة vector بقيم محددة
    vector <int> myVector = {1,2,3,4,5};
    تهيئة vector فارغة
    vector <int> myVector2;
    تهيئة vector بقيمة واحدة
    vector <int> myVector3 = {10};

    return 0;
}
```

2. تهيئة الـ **vector** باستخدام حلقات (**loops**)  
نُستخدم الحلقات لتعيين قيم للعناصر بعد إعلان الـ **vector**  
مثال:

```
#include <vector>
int main()
{
    تهيئة vector بقيم متزايدة
    vector <int> myVector;
    for (int i = 0; i < 10; i++)
    {
        myVector.push_back(i);
    }

    تهيئة vector بقيم عشوائية
    vector <int> myVector2;
    for (int i = 0; i < 10; i++)
    {
        myVector2.push_back(rand() % 100);
    }
    return 0;
}
```

```
#include <iostream>
#include <vector>

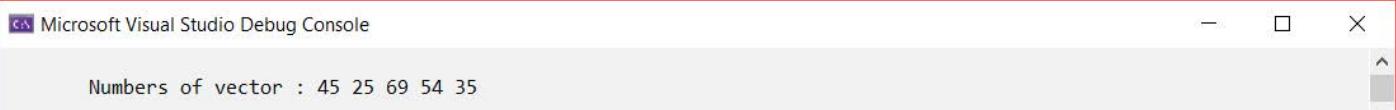
using namespace std;

int main()
{
    system("color f0");

    vector<int> MyFristVector = { 45, 25, 69, 54, 35 };

    cout << "\n\tNumbers of vector : ";

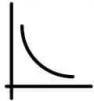
    for (int& Num : MyFristVector) // مهم جدا الريفرنس &
    {
        cout << Num << " ";
    }
    cout << endl;
}
```





## C++ vector vs array

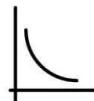
### C++ vector      array



Size of the vectors is not fixed (it can be resized) and it can grow and shrink on insertion and deletion. They are allocated on heap memory.

حجم الـ **vectors** غير ثابت (يمكن تغييره)، حيث يمكن أن ينمو أو يتقلص عند إضافة أو حذف العناصر. يتم تحديدهم في ذاكرة كومة البرنامج (Heap Memory).

### array



Size of the array is fixed unlike vectors.

حجم المصفوفة ثابت بخلاف الـ **vectors**.

### C++ vector



Reallocation of memory in case of Vectors is done implicitly.

إعادة تخصيص الذاكرة في حالة الـ **vectors** يتم ص�نياً.

### array



Reallocation of memory in case of Array is not done implicitly.

إعادة تخصيص الذاكرة في حالة المصفوفات لا يتم صصنياً.

### C++ vector



In programming, vectors can be copied or assigned directly.

في البرمجة، يمكن نسخ الـ **vectors** أو تعيينها مباشرةً.

### array



In programming, arrays can never be copied or assigned directly.

في البرمجة، لا يمكن نسخ المصفوفات أو تعيينها مباشرةً.

### C++ vector



It follows the non-index based structure as the elements are stored dynamically.

تنبع هيكلًا غير قائم على الفهرس لأن العناصر يتم تخزينها ديناميكياً.



It follows the index-based structure with the elements stored in the contiguous memory having the first element at the lowest address and the last element at the highest address.

تنبع هيكلًا قائماً على الفهرس حيث تُخزن العناصر في ذاكرة متغيرة، بحيث يكون العنصر الأول في أقل عنوان والعنصر الأخير في أعلى عنوان.

### C++ vector



Vector is a template class in C++ which will be shipped from the C++ library if needed to use the vector functions.

الـ **vector** هو فئة قابلة (template class) في لغة C++ يتم استدعاؤها من مكتبة C++ عند الحاجة إلى استخدام وظائف الـ **.vector**.

### array



Array is not a template class but is a lower level data structure which can be used anytime.

المصفوفة ليست فئة قابلية، بل هي بنية بيانات ذات مستوى أدنى يمكن استخدامها في أي وقت.

### C++ vector



Vectors in C++ can be considered as a dynamic array whose size can be changed with the insertion and deletion of elements.

يمكن اعتبار الـ **vectors** في لغة C++ بمثابة مصفوفات ديناميكية يمكن تغيير حجمها مع إضافة وحذف العناصر.

### array



Array in C++ can be considered as a static array whose size is fixed once initialized.

يمكن اعتبار المصفوفة في لغة C++ بمثابة مصفوفة ثابتة الجم يتم تحديد حجمها عند التهيئة.

### C++ vector



When it comes to memory management, vectors occupy more memory in comparison to Arrays.

من حيث إدارة الذاكرة، تشغل الـ **vectors** مساحة أكبر مقارنة بالمصفوفات.

### array



Arrays are memory efficient and occupy less memory than vectors.

المصفوفات أكثر كفاءة في استخدام الذاكرة وتحتل مساحة أقل من الـ **vectors**.

### C++ vector



As the Vectors follow the dynamic structure, so it takes more time to access the elements in Vectors.

نظرًا لأن الـ **vectors** تتبع بنية ديناميكية، فإن الوصول إلى العناصر فيها يستغرق وقتًا أطول مقارنة بالمصفوفات التقليدية.

### array



Arrays are static in nature so it takes constant time to access any element in the array using the index operator.

المصفوفات ثابتة بطبعتها، لذا يستغرق الوصول إلى أي عنصر في المصفوفة باستخدام عامل الفهرسة وقتًا ثابتًا.



# Lesson #29 - Add elements

- الـ **Stack** هو نوع من انواع **Data Structures**
- الـ **Vector** يستخدم **Stack** من نوع **Data Structures**
- الـ **Stack** يمتاز بخصائص **Push** دخول البيانات و **Pop** خروج البيانات

؟ دائمًا استخدم **Vector** مع الـ **By Reference (&)** !

؟ العمليات على الـ **Vectors**: تدعم الـ **Vectors** العديد من العمليات الشائعة على المصفوفات، مثل:

- الوصول إلى العناصر: باستخدام الـ **index** الخاص بكل عنصر، مثل **[0]** للوصول إلى العنصر الأول.
- إضافة عناصر: باستخدام الدالة **push\_back()** لإضافة عنصر في نهاية الـ **Vector**.
- حذف عناصر: باستخدام الدالة **pop\_back()** لحذف العنصر الأخير من الـ **Vector**، أو باستخدام الدالة **erase()** لحذف عناصر معينة من أي مكان في الـ **Vector**.
- الحصول على حجم الـ **Vector**: باستخدام الدالة **(size)** لمعرفة عدد العناصر الموجودة فيه.
- التحقق من كون الـ **Vector** فارغ: باستخدام الدالة **empty()** لمعرفة ما إذا كان الـ **Vector** فارغاً أم لا.

بفضل هذه الخصائص، تعتبر الـ **Vectors** من أكثر بنى البيانات استخداماً في لغة **C++** ، خاصةً عند الحاجة إلى التعامل مع مجموعات من البيانات التي قد يتغير حجمها أثناء تشغيل البرنامج.

؟ تستخدم دالة **(push\_back())** لإضافة عناصر جديدة إلى نهاية الـ **vector**. يتم إدراج القيمة الجديدة في نهاية الـ **vector**، بعد آخر عنصر حالي ، ويزداد حجم الـ **vector** بمقدار عنصر واحد.

؟ بناء الجملة: **vectorname.push\_back(value)**

المتغيرات (**Parameters**):

القيمة (**Value**): هي القيمة التي تريد إضافتها إلى نهاية الـ **vector**. ويمكن أن تكون أي نوع من البيانات يدعمه الـ **vector** (مثل الأرقام، النصوص، الكائنات، إلخ).

النتيجة (**Result**):

تضيف دالة **(push\_back())** القيمة المذكورة كمعلمة إلى نهاية الـ **vector** المسمى اسم الـ **vector**. يزداد حجم الـ **vector** تلقائياً بمقدار عنصر واحد لاستيعاب القيمة الجديدة.

؟ شرح مفصل:

عندما تقوم باستدعاء : **push\_back()**

- يتم إنشاء نسخة جديدة من العنصر المراد إضافته.
- يتم إدراج النسخة الجديدة في نهاية الـ **vector**، بعد العنصر الأخير الحالي.
- يتم زيادة حجم الـ **vector** بمقدار عنصر واحد.

```

#include <iostream>
#include <vector>
using namespace std;
int main()
{
    system("color f0");
    vector <int> VNumbers = { 10,20,30,40,50 };
    cout << "\nWith Ranged Loop For\n";
    cout << "-----\n";
    //Ranged Loop
    for (int& i : VNumbers)
    {
        cout << i << " ";
    }
    cout << "===== \n";
    cout << "\nWith Normal Loop For\n";
    cout << "-----\n";
    vector <char> vMyName = { 'A','h','m','e','d' };
    for (int i = 0; i < vMyName.size(); i++)
    {
        cout << vMyName[i];
    }
    cout << "===== \n";
    cout << "\nWith Ranged Loop For\n";
    cout << "-----\n";
    for (char& C : vMyName)
    {
        cout << C;
    }
    cout << "===== \n\n";
    vector <int> Array2;           // size is Zero
    Array2.push_back(110);         // .push_back()
    نبع الغاصل بـ()
    Array2.push_back(120);
    Array2.push_back(130);
    for (int& j : Array2)
    {
        cout << j << " ";
    }
    cout << endl;
}

```

Microsoft Visual Studio Debug Console

```

With Ranged Loop For
-----
10 20 30 40 50
=====
With Normal Loop For
-----
Ahmed
=====
With Ranged Loop For
-----
Ahmed
=====
110 120 130

```

```
#include <iostream>
#include <vector>

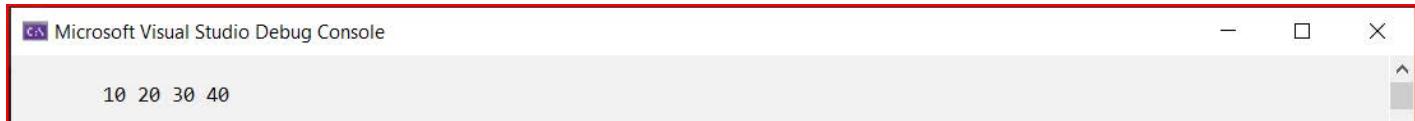
using namespace std;

void vNumbers()
{
    vector <int> vNumber;

    vNumber.push_back(10);
    vNumber.push_back(20);
    vNumber.push_back(30);
    vNumber.push_back(40);
    cout << "\n\t";
    for (int& i : vNumber)
    {
        cout << i << " ";
    }
    cout << endl;
}

// Exam

int main()
{
    vNumbers();
}
```



A screenshot of the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console displays the following text:  
10 20 30 40

## Exams

Write a Program to ask user to enter as many numbers as s/he wants, each time a number entered add it to your vector, and ask the user if s/he wants to add more numbers until s/he says No, then print all vector elements on the screen.

```
#include <iostream>
#include <vector>
using namespace std;

void ReadNumbers(vector <int>& vNumber) // مهم الرفرنس
{
    int Num;
    char ReadMore = 'Y';
    do
    {
        cout << "Enter a number: ";
        cin >> Num;
        vNumber.push_back(Num);

        cout << "\nDo you want to add more numbers (Y/N): ";
        cin >> ReadMore;
    } while (ReadMore == 'Y' || ReadMore == 'y');
}

void PrintVectorNumber(vector <int>& vNumber) // مهم الرفرنس
{
    ReadNumbers(vNumber);

    cout << "\n\nThe Vectors Numbers is: ";

    for (int& i : vNumber)
    {
        cout << i << " ";
    }
    cout << endl;
}

int main()
{
    vector<int>vNumber;

    PrintVectorNumber(vNumber);
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The console output is as follows:

```
Microsoft Visual Studio Debug Console
Enter a number: 15
Do you want to add more numbers (Y/N): y
Enter a number: 20
Do you want to add more numbers (Y/N): y
Enter a number: 25
Do you want to add more numbers (Y/N): y
Enter a number: 30
Do you want to add more numbers (Y/N): y
Enter a number: 35
Do you want to add more numbers (Y/N): n
The Vectors Numbers is: 15 20 25 30 35
```



# Lesson #30 - Vector of Structure

يمكن استخدام متوجه الهياكل (**Vector of structs**) عندما تحتاج إلى تتبع السجلات في هيكل بيانات لأشخاص مختلفين بخصائص مختلفة. على سبيل المثال، تخيل أنك تطور نظاماً لإدارة مكتبة. قد ترغب في تخزين معلومات حول كل عضو، بما في ذلك اسمه ورقم عضويته وحالته النشطة والكتب المستعارة حالياً.

في هذه الحالة، يمكنك تعريف **هيكل "عضو"** (**Member**) يحتوي على حقول لجميع المعلومات ذات الصلة:

```
struct Member{  
    string name;  
    int id;  
    bool active;  
    vector<Book> borrowedBooks;  
};
```

```
int main()  
{
```

ثم يمكنك استخدام متوجه من الهياكل لتخزين معلومات حول جميع الأعضاء في المكتبة:

```
vector<Member> members;  
Member newMember;           // اضافة عضو جديد  
newMember.name = "John Doe";  
newMember.id = 1234;  
newMember.active = true;  
members.push_back(newMember);  
Member firstMember = members[0]; // الوصول الى معلومات عضو معين  
cout << "First Name : " << firstMember.name << endl;
```

الوصول الى معلومات الاعضاء //

```
for(Member &Persons : members)  
{  
    cout << "First Name : " << Persons.name << endl;  
    cout << "Id          :" << Persons.id << endl;  
    cout << "Active      :" << Persons.active << endl;  
}
```

كما ترى، فإن استخدام متوجه الهياكل يوفر طريقة منظمة وفعالة لتخزين المعلومات حول مجموعة من الكائنات ذات الخصائص المختلفة.

```

#include <iostream>
#include <vector>
using namespace std;

struct stEmployee
{
    string FirstName;
    string LastName;
    float Salary;
};

int main()
{
    vector<stEmployee> vEmployees;
    stEmployee tempEmployee;
    tempEmployee.FirstName = "Ahmed";
    tempEmployee.LastName = "AbdelRahim";
    tempEmployee.Salary = 5000;
    vEmployees.push_back(tempEmployee);
    tempEmployee.FirstName = "Ibrahim";
    tempEmployee.LastName = "ElSayed";
    tempEmployee.Salary = 4000;
    vEmployees.push_back(tempEmployee);
    tempEmployee.FirstName = "Mohamed";
    tempEmployee.LastName = "AbdelRahim";
    tempEmployee.Salary = 3500;
    vEmployees.push_back(tempEmployee);
    tempEmployee.FirstName = "Islam";
    tempEmployee.LastName = "AbdelRahim";
    tempEmployee.Salary = 2500;
    vEmployees.push_back(tempEmployee);
    cout << "\n\tThe data of employees: " << endl;
    for (stEmployee& Employee : vEmployees)
    {
        cout << "\n\tFirst Name: " << Employee.FirstName << endl;
        cout << "\tLast Name : " << Employee.LastName << endl;
        cout << "\tSalary      : " << Employee.Salary << endl;
    }
    cout << endl;
}

```

Microsoft Visual Studio Debug Console

```

The data of employees:

First Name: Ahmed
Last Name : AbdelRahim
Salary     : 5000

First Name: Ibrahim
Last Name : ElSayed
Salary     : 4000

First Name: Mohamed
Last Name : AbdelRahim
Salary     : 3500

First Name: Islam
Last Name : AbdelRahim
Salary     : 2500

```

## Exam

Write a Program to ask user to enter as many Employees as s/he wants, each time an Employee entered add it to your vector and ask the user if s/he wants to add more Employees until s/he says No, then print all vector elements on the screen.

```
#include <iostream>
#include <vector>
using namespace std;

struct stOffice {
    string OfficeName;
    string Address;
};

struct stEmployee {
    string firstname;
    string lastname;
    int salary;
    stOffice Office;
};

void ReadEmployeesData(vector<stEmployee>& VEmployee)
{
    stEmployee tempEmployee;
    char MoreAgain = 'Y';
    int i = 1;

    while (MoreAgain == 'Y' || MoreAgain == 'y')
    {
        cout << "\nEmployee [" << i << "]: \n";
        cout << "Enter First Name : ";
        cin >> tempEmployee.firstname;
        cout << "Enter Last Name : ";
        cin >> tempEmployee.lastname;
        cout << "Enter Salary : ";
        cin >> tempEmployee.salary;
        cout << "Enter Name of office: ";
        cin >> tempEmployee.Office.OfficeName;
        cout << "Enter Address of office: ";
        cin >> tempEmployee.Office.Address;
        VEmployee.push_back(tempEmployee);
        cout << "Do you want to add more Employees ? (Y/N) : ";
        cin >> MoreAgain;
        i++;
    }
}
```

```

void PrintData(vector<stEmployee>& VEmployee)
{
    cout << "\n-----\n";
    cout << "The Data of Employees : \n";
    int i = 1;

    for (stEmployee& Employees : VEmployee)
    {
        cout << "Employee [" << i << "] is: \n";
        cout << "First Name : " << Employees.firstname << endl;
        cout << "Last Name : " << Employees.lastname << endl;
        cout << "Salary : " << Employees.salary << endl;
        cout << "Name of office : " << Employees.Office.OfficeName << endl;
        cout << "Address of office: " << Employees.Office.Address << endl;
        i++;
    }
    cout << endl;
}

int main()
{
    vector<stEmployee> VEmployee;

    ReadEmployeesData(VEmployee);
    PrintData(VEmployee);

    system("pause");
}

```

```

Employee [1]:
Enter First Name : Ahmad
Enter Last Name : AbdelRahim
Enter Salary : 3000
Enter Name of office: HORUS
Enter Address of office: Egypt
Do you want to add more Employees ? (Y/N) : y

```

```

Employee [2]:
Enter First Name : Mahmoud
Enter Last Name : Ahmad
Enter Salary : 4000
Enter Name of office: HORUS
Enter Address of office: Egypt
Do you want to add more Employees ? (Y/N) : n
-----
```

```

The Data of Employees :
Employee [1] is:
First Name : Ahmad
Last Name : AbdelRahim
Salary : 3000
Name of office : HORUS
Address of office: Egypt

```

```

Employee [2] is:
First Name : Mahmoud
Last Name : Ahmad
Salary : 4000
Name of office : HORUS
Address of office: Egypt

```



# Lesson #31 - Remove elements

الدالة **pop\_back()** : تُستخدم لإزالة أو حذف عناصر من نهاية المتجه (vector). يتم حذف القيمة من نهاية المتجه، ويقل حجم المتجه بمقدار 1.

؟ شرح مفصل:

- دالة **pop\_back()** هي طريقة شائعة لإزالة العنصر الأخير من المتجه.

عند استدعاء **:pop\_back()**

- يتم حذف العنصر الأخير من المتجه.
- يتم تقليل حجم المتجه تلقائياً بمقدار 1.
- يتم استدعاء المدمر (destructor) للعنصر المحذوف، مما يؤدي إلى تحرير أي موارد مرتبطة به.

؟ ملاحظات مهمة:

- لا ترجع دالة **pop\_back()** قيمة. إنها تزيل العنصر من المتجه فقط.
- إذا كان المتجه فارغاً، فإن استدعاء **pop\_back()** يؤدي إلى سلوك غير محدد. يجب التحقق من أن المتجه ليس فارغاً قبل استدعاء **pop\_back()**.
- لاتحاول الوصول إلى عناصر تمت إزالتها بالفعل لتجنب حدوث سلوك غير محدد.

؟ استخدامات شائعة لدالة **:pop\_back()**

- إزالة العناصر من نهاية قائمة انتظار (queue).
- التراجع عن آخر عملية إضافة إلى المتجه.
- تحرير الذاكرة المستخدمة من قبل العناصر غير الضرورية في المتجه.

؟ الدالة **size()**

تُستخدم الدالة **vector::size()** لإعادة حجم الـ **Vector** أو عدد العناصر الموجودة حالياً في الـ **Vector**.  
تُستخدم مع اسم الـ **Vector** ، مثل **(numbers.size())**.

؟ كيفية استخدامها:

استدعاء الدالة **size()** على الـ **Vector** :

```
#include <vector>
#include <iostream>
int main()
{
    vector<int> numbers = {1,2,3,4,5};
    size_t numElements = numbers.size();           // numElements سيكون 5
    cout << "Numbers of elements in vector : " << numElements << endl;
}
```

## • ملاحظات:

- تُرجع الدالة `size()` قيمة من النوع `unsigned int` `size_t` (وهو نوع بيانات غير سالب يستخدم لتمثيل أحجام الكائنا `empty()` الدالة).
- لا تُستخدم الدالة `empty()` للتحقق أو معرفة ما إذا كان ال `Vector` يحتوي على أي عناصر أم لا.
- القيمة `empty()` الدالة.
- من الـ `empty()` كيفية استخدامها:

```
#include <vector>
int main()
{
    vector<int> myVector; // المتوجه فارغ في البداية

    if (myVector.empty()) // true
    {
        cout << "Vector is empty" << endl;
    }
    else // false
    {
        cout << "Vector is not empty" << endl;
    }

    for (size_t i = 0; i < myVector.size(); ++i)
    {
        cout << myVector[i] << " ";
    }
}
```

## • أهمية

- الوصول إلى العناصر باستخدام الفهارس (بشرط أن يكون الفهرس أقل من `(size)`).
- تحديد مقدار الذاكرة المستخدمة بواسطة المتوجه.
- تجنب الوصول إلى عناصر خارج نطاق المتوجه، مما قد يؤدي إلى أخطاء أو سلوك غير محدد.
- تحسين أداء الكود عن طريق تجنب العمليات غير الضرورية على عناصر المتوجه الفارغة.

## • الدالة `empty()`

تُستخدم الدالة `empty()` للتحقق أو معرفة ما إذا كان ال `Vector` يحتوي على أي عناصر أم لا.

## • كيفية استخدامها:

```
#include <vector>
int main()
{
    vector<int> myVector; // المتوجه فارغ في البداية

    if (myVector.empty()) // true
    {
        cout << "Vector is empty" << endl;
    }
    else // false
    {
        cout << "Vector is not empty" << endl;
    }
}
```

## ؟ ملاحظات مهمة:

- لا تحتاج إلى تمرير أي معلومات إضافية (معلمات) (Parameters) عند استدعائها.
- تُرجع الدالة **empty()** قيمة منطقية (boolean) من نوع **bool** (صواب أو خطأ):
  - تُرجع القيمة **true** (صحيح) : إذا كان المتجه فارغاً (لا يحتوي على أي عناصر).
  - تُرجع القيمة **false** (خطأ) : إذا كان المتجه يحتوي على عناصر.

## ؟ تُعد هذه الدالة مفيدة في العديد من الحالات، مثل:

- التحقق من خلو المتجه قبل إجراء عمليات عليه. **معنى آخر** : تجنب حدوث أخطاء عند محاولة الوصول إلى عناصر من متجه فارغ.
- التحكم في تدفق البرامج بناءً على حالة المتجه.
- تحسين أداء الخوارزميات من خلال تجنب العمليات غير الضرورية على المتجهات الفارغة.
- تجنب الأخطاء الناتجة عن محاولة الوصول إلى عناصر في متجه فارغ.

## ؟ الدالة **clear()**:

تُستخدم الدالة **clear()** لإزالة جميع عناصر حاوية الـ **Vector** ، مما يجعل حجمها **0** - **معنى آخر** : طريقة سريعة وفعالة لتفريغ الـ **Vector** من جميع محتوياته.

## ؟ كيفية استخدامها:

```
#include <vector>
int main()
{
    vector<int> numbers = {1,2,3,4,5};
    numbers.clear()           // إزالة جميع العناصر من المتجه
    if (numbers.empty())
    {
        cout << "Vector is empty" << endl;
    }
}
```

## ؟ ملاحظات مهمة:

- لا تحتاج إلى تمرير أي معلومات إضافية (معلمات) (Parameters) عند استدعائها.
- لا تمحى الدالة **clear()** الذاكرة المخصصة للـ **Vector** نفسه، بل تزيل فقط العناصر الموجودة فيه.  
بعد استدعاء **clear()**:
  - يصبح حجم الـ **Vector** يساوي **0**.
  - تصبح جميع عناصر الـ **Vector** غير صالحة، ولا يمكن الوصول إليها.
  - لا يتم استدعاء مدمر عناصر الـ **Vector** ، مما قد يؤدي إلى تسرب للذاكرة إذا كانت العناصر تحتوي على موارد تتطلب تحريراً صريحاً.

## ؟ استخداماتها الشائعة:

- إعادة تعين الـ **Vector** لإعادة استخدامه.
- تحرير الذاكرة المستخدمة من قبل عناصر الـ **Vector** (في بعض الحالات).
- تهيئة الـ **Vector** لحالة فارغة.

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    system("color f0");

    vector <int> vNumbers;
    // تعيينة العناصر
    vNumbers.push_back(10);
    vNumbers.push_back(20);
    vNumbers.push_back(30);
    vNumbers.push_back(40);
    vNumbers.push_back(50);

    cout << "\n\tStack Size: " << vNumbers.size() << endl; // عرض حجم الفيكتور

    // مسح العناصر
    vNumbers.pop_back();
    vNumbers.pop_back();
    vNumbers.pop_back();
    vNumbers.pop_back();
    vNumbers.pop_back();

    cout << "\n\tStack Size: " << vNumbers.size() << endl;

    vNumbers.clear();

    for (int& Number : vNumbers)
    {
        cout << Number << endl;
    }

    return 0;
}

```

The screenshot shows the Microsoft Visual Studio Debug Console window. It displays two lines of text output: "Stack Size: 5" followed by "Stack Size: 0". The console has standard window controls (minimize, maximize, close) at the top right.

```

Microsoft Visual Studio Debug Console
Stack Size: 5
Stack Size: 0

```



# Lesson #32 - Vector Functions

## الدالة front()

تُستخدم هذه الدالة لجلب العنصر الأول من حاوية ال **Vector** - بمعنى اخر - تتيح الدالة **front()** إمكانية الوصول إلى العنصر الأول في ال **Vector** بشكل مباشر وفعال.

Syntax : **vectorname.front()**

المعلمات (**Parameters**) : لا تحتاج إلى تمرير أي قيمة كمعلمة (**Parameter**).  
قيمة الإرجاع (**Returns**) : تُرجع مرجعاً مباشراً إلى العنصر الأول في حاوية ال **Vector**.

## كيفية استخدامها:

```
#include <vector>
int main()
{
    vector<int> numbers = {10,20,30,40};
    int firstElement = numbers.front();           // firstElement
    cout << "The first element in vector" << firstElement << endl;
}
```

## ملاحظات مهمة:

- تُرجع الدالة **front()** مرجعاً إلى العنصر الأول في ال **Vector** ، مما يعني أنه يمكن استخدامه لتعديل القيمة مباشرةً إذا لزم الأمر.
- يجب توخي الحذر عند استخدام (**front()** مع **Vector** فارغ) حيث يؤدي استدعاؤها على **Vector** فارغ إلى سلوك غير محدد (**undefined behavior**) مما قد يتسبب في حدوث أخطاء أو أخطال في البرنامج.

## استخداماتها الشائعة:

- الحصول على قيمة العنصر الأول في ال **Vector** لتفحصه أو استخدامه في عمليات أخرى.
- تعديل قيمة العنصر الأول بشكل مباشر.
- التحقق من قيمة العنصر الأول قبل إجراء عمليات على ال **Vector**.

## ؟ الدالة back()

تُستخدم هذه الدالة لجلب العنصر **الأخير** من حاوية المتّجه - بمعنى آخر - توفر الدالة **back()** طريقة سهلة للوصول إلى **العنصر الأخير** في المتّجه دون الحاجة إلى معرفة حجمه مسبقاً.

Syntax : **vectorname.back()**

المعلمات (**Parameters**) : لا تحتاج إلى تمرير أي قيمة كمعلمة (**Parameter**).  
قيمة الإرجاع (**Returns**) : تُرجع مرجعاً مباشراً إلى العنصر الأخير في حاوية ال **Vector**.

### ؟ كيفية استخدامها:

```
#include <vector>
int main()
{
    vector<int> numbers = {1,2,3,4,5};
    int lastElement = numbers.back();           // lastElement 5 سيكون
    cout << "The last element in vector" << lastElement << endl;
}
```

### ؟ ملاحظات مهمة:

- تُرجع الدالة **back()** مرجعاً إلى العنصر الأخير في ال **vector** ، مما يعني أنه يمكن استخدامه لتعديل القيمة مباشرةً.
- يجب توخي الحذر عند استخدام **back()** مع **Vector** فارغ ، حيث يؤدي استدعاؤها على **Vector** فارغ إلى سلوك غير محدد (**undefined behavior**) مما قد يتسبب في حدوث أخطاء أو أخطال في البرنامج.

### ؟ استخدامات شائعة:

- الحصول على العنصر الأخير في ال **vector** لتفحصه أو استخدامه في عمليات أخرى.
- تعديل قيمة العنصر الأخير بشكل مباشر.
- التحقق من صحة العنصر الأخير قبل إجراء عمليات على ال **vector**.

## ؟ الدالة capacity()

- تُستخدم الدالة capacity() لمعرفة مقدار الذاكرة التي تم تخصيصها بالفعل لـ **Vector** ، مما يوفر معلومات حول مدى إمكانية نمو ال **Vector** قبل الحاجة إلى إعادة تخصيص الذاكرة. حيث أنها ترجع حجم مساحة التخزين المخصصة حالياً لـ **Vector** ، معتبراً عنه بعدد العناصر.
- لا تتساوى هذه السعة بالضرورة مع حجم ال **Vector** . إذ يمكن أن تكون مساوية له أو أكبر منه، حيث تسمح المساحة الإضافية إمكانية استيعاب الزيادة دون الحاجة إلى إعادة تخصيص الذاكرة عند كل إضافة عنصر جديد.
- لا تمثل السعة حداً لحجم ال **Vector** . فعندما تنفذ هذه السعة وهناك حاجة إلى زيادة المساحة، يقوم الحاوية بتوسيعتها تلقائياً (من خلال إعادة تخصيص مساحة التخزين).
- يتم تحديد الحد النظري لحجم ال **Vector** من خلال العضو **.max\_size()**.

Syntax : `vector_name.capacity()`

المعلمات (Parameters): لا تحتاج إلى تمرير أي قيمة كمعلمة (Parameter).  
قيمة الإرجاع (Returns): تُرجع الدالة حجم مساحة التخزين المخصصة حالياً للمتجه، معتبراً عنه بعدد العناصر.

## ؟ كيفية استخدامها:

```
#include <vector>
using namespace std;

int main()
{
    vector<int>v;
    for (int i = 0; i < 100; i++)          // inserts elements
    {
        v.push_back(i * 10);
    }
    cout << "The size of vector is " << v.size();           output : The size of vector is 100
    cout << "\nThe maximum capacity is " << v.capacity();   The maximum capacity is 128
    return 0;
}
```

## ؟ استخدامات شائعة:

- مراقبة استخدام الذاكرة من قبل ال **Vector** .
- تحسين أداء عمليات الإدراج المتكررة.
- تجنب عمليات إعادة التخصيص غير الضرورية.
- التحقق من سعة ال **Vector** قبل إضافة عناصر جديدة.

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vNumbers;

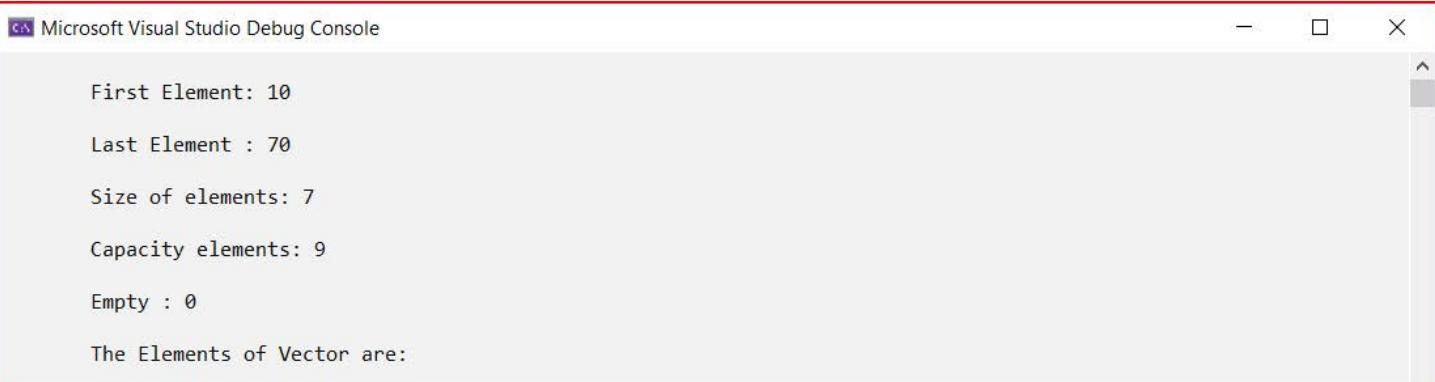
    vNumbers.push_back(10);
    vNumbers.push_back(20);
    vNumbers.push_back(30);
    vNumbers.push_back(40);
    vNumbers.push_back(50);
    vNumbers.push_back(60);
    vNumbers.push_back(70);

    cout << "\n\tFirst Element: " << vNumbers.front() << endl;
    cout << "\n\tLast Element : " << vNumbers.back() << endl;
    cout << "\n\tSize of elements: " << vNumbers.size() << endl;
    cout << "\n\tCapacity elements: " << vNumbers.capacity() << endl;
    cout << "\n\tEmpty : " << vNumbers.empty() << endl;

    vNumbers.clear();

    cout << "\n\tThe Elements of Vector are: " << endl;
    for (int &Number : vNumbers)
    {
        cout << "\t" << Number << endl;
    }
}

```



The screenshot shows the Microsoft Visual Studio Debug Console window. The console displays the following output from the program:

```

First Element: 10
Last Element : 70
Size of elements: 7
Capacity elements: 9
Empty : 0
The Elements of Vector are:

```



# Lesson #33 - Call ByRef/ByVal

## Important Review

The **actual parameters** also known as **arguments** are the parameters that are passed into the function when we make a function call whereas **formal parameters** are the parameters that we see in the function or method definition i.e. the parameters received by the function.

المعاملات الفعلية (**Actual Parameters**)، المعروفة أيضاً باسم **البُرُومات** (**Arguments**)، هي القيم التي يتم تمريرها إلى الدالة عند استدعائهما. أما المعاملات الشكلية أو الرسمية (**Formal Parameters**) فهي المتغيرات التي تظهر في تعريف الدالة، أي القيم التي تستقبلها الدالة عند استدعائهما.

Here's a clear explanation of actual and formal parameters in function calls:

فيما يلي شرح واضح للمعاملات الفعلية والرسمية في استدعاء الدوال:

في عالم البرمجة، عند استخدام الدوال نقوم بإمدادها ببيانات تحتاجها لتعمل بشكل صحيح. لتوضيح كيفية انتقال هذه البيانات، نستخدم مصطلح "المعاملات الفعلية" و"المعاملات الرسمية".

**Actual Parameters (Arguments):** المعاملات الفعلية (البُرُومات)

These are the values that you provide when calling a function.

هي القيم التي نمررها فعلياً إلى الدالة عند استدعائهما.

They represent the actual data that you want the function to work with.

تعتبر هي "المدخلات" التي نمنحها للدالة لتعمل عليها.

They are listed within the parentheses during the function call.

يتم كتابة المعاملات الفعلية داخل الأقواس عند استدعاء الدالة.

**Example:**

```
int multiply(int x, int y) { ... } // x and y are formal parameters
int main() {
    int result = multiply(5, 3);    // 5 and 3 are actual parameters (arguments) }
```

**Formal Parameters:** المعاملات الشكلية

These are the placeholders defined within the function's header.

هي المتغيرات التي نضعها داخل تعريف الدالة لاستقبال المعاملات الفعلية.

They act as variables that receive the values passed as actual parameters.

تعمل كمتغيرات تتلقى القيم التي تم تمريرها كمعاملات فعلية.

They are used within the function's body to access and manipulate the data.

تستخدم داخل جسم الدالة للوصول إلى البيانات ومعالجتها.

**Example (continuing from above):**

```
int multiply(int x, int y) { // x and y are formal parameters
    int product = x * y;
    return product; }
```

Key Points:

**Actual parameters represent the input values you supply when calling the function.**

المعاملات الفعلية تمثل البيانات (القيم) التي ندخلها عند استدعاء الدالة.

**Formal parameters are like "receivers" that capture those values inside the function.**

المعاملات الشكلية بمثابة "حاويات" تستقبل هذه القيم داخل الدالة.

**The number and types of actual parameters must match the formal parameters in the function's definition.**

يجب أن يتطابق عدد وأنواع المعاملات الفعلية مع المعاملات الشكلية في تعريف الدالة.

**Imagine a function as a machine:** يمكننا تشبیه الدالة بالآلة تعمل بشكل مشابه:

**Actual parameters are like the raw materials you feed into the machine.**

المعاملات الفعلية تشبه المواد الخام التي ندخلها في الآلة.

**Formal parameters are like the designated slots within the machine that hold those materials for processing.**

المعاملات الشكلية تشبه الفتحات المخصصة داخل الآلة لاستيعاب هذه المواد ومعالجتها.

## Call By Value & Call By Reference !

في لغة C++, هناك طريقتان لتمرير المعاملات (Arguments) إلى الدوال (Functions) :

- تمرير بالقيمة (Call By Value)
- تمرير بالمرجع (Call By Reference).

### ١- تمرير بالقيمة (Call By Value)

- هو الطريقة الافتراضية لتمرير المعاملات (Arguments) إلى الدوال (Functions) إلى الدوال (Arguments) إلى الدوال (Functions).
- يتم إنشاء نسخة من قيمة المعامل (Argument) الأصلي وإرسالها إلى الدالة (Function).
- لا تؤثر أي تغييرات تتم على النسخة داخل الدالة (Function) على المعامل (Argument) الأصلي.

#### مثال توضيحي ١ :

```
void add_one(int x)
{
    x++;
}

int num = 5;
add_one(num);
cout << num << endl;      // 5 يطبع
```

في هذا المثال، يتم تمرير المتغير num بالقيمة إلى دالة add\_one() تقوم الدالة بزيادة قيمة x بمقدار 1، ولكن لا تؤثر على قيمة المتغير الأصلي num.

في أسلوب تمرير بالقيمة، يتم تمرير المعاملات إلى الدوال عن طريق نسخ قيمة المعامل الفعلي - مما يضمنبقاء القيم الأصلية دون تغيير. إذ يتم نسخ القيمة إلى المعامل الشكلي - مما يعني إنشاء نسختين من القيمة:

**النسخة الأصلية**: وهي المتغير الأصلي الذي تم تمريره إلى الدالة من خارجها.

**نسخة الدالة**: وهي النسخة التي يتم إنشائها داخل الدالة لاستقبال المعامل، وتكون مستقلة عن النسخة الأصلية.

\* أي تغييرات يتم إجراؤها على المعامل داخل الدالة **تؤثر فقط على نسخة الدالة**، ولا تؤثر على النسخة الأصلية خارج الدالة مما يضمنبقاء القيم الأصلية دون تغيير.

## ؟**شرح خطوة بخطوة:**

**استدعاء الدالة**: عندما يتم استدعاء دالة باستخدام طريقة تمرير بالقيمة، يقوم المترجم بإنشاء نسخة من قيمة المعامل الفعلي وإرسالها إلى الدالة.

**إنشاء نسخة الدالة**: يتم تخزين هذه النسخة في متغير خاص داخل الدالة، يسمى المعامل الشكلي.

**التعامل مع النسخة**: جميع العمليات التي تحدث داخل الدالة تتم على هذه النسخة، وليس على النسخة الأصلية.

**انتهاء الدالة**: عدم التأثير على الأصل، أي تعديل أو تغيير يتم على النسخة داخل الدالة تظل محصورة داخلها ولا تتعكس على المتغير الأصلي خارج الدالة.

### **مثال توضيحي 2 :**

```
void add_one(int x)      // هو المعامل الشكلي (نسخة الدالة) x
{
    x++;                 // زيادة قيمة x داخل الدالة ( يتم تعديل النسخة فقط )
}
int main()
{
    int num = 5;          // هو المعامل الفعلي (النسخة الأصلية) num
    add_one(num);         // تمرير num بالقيمة ( يتم إنشاء نسخة من num داخل الدالة )
    cout << num << endl;   // سيظل يطبع 5 ، لأن النسخة الأصلية لم تتغير
}
```

في هذا المثال، على الرغم من زيادة قيمة x داخل الدالة ( add\_one() ) إلى 6 ، إلا أن قيمة num في main تبقى 5 لأن التعديل تم على نسخة الدالة فقط.

## ؟**مميزات تمرير بالقيمة:**

• حماية البيانات الأصلية: يضمن عدم تعديل القيم الأصلية للمتغيرات عن طريق الخطأ داخل الدالة.

• سهولة الاستخدام: هو الأسلوب الافتراضي في C++ ، ولا يتطلب استخدام أي علامات خاصة.

• مناسب للقيم البسيطة: مثل الأرقام والمتغيرات الأساسية.

## ؟**حالات استخدام تمرير بالقيمة:**

• عندما لا تزيد تعديل المعاملات الأصلية داخل الدالة.

• عندما تزيد حماية البيانات الأصلية من التعديلات غير المقصودة.

• عندما تكون المعاملات صغيرة الحجم، بحيث لا يؤثر نسخها على الأداء بشكل كبير.

## ؟**ملاحظات مهمة:**

• يُستخدم تمرير بالقيمة عادةً عندما لا ترغب في تعديل المعاملات الأصلية داخل الدالة، أو عندما تتعامل مع بيانات بسيطة لا تتطلب مشاركة الذاكرة.

• عند تمرير المتغيرات الكبيرة (مثل المصفوفات) بالقيمة، قد يؤدي ذلك إلى استهلاك ذاكرة أكثر، لأنه يتم إنشاء نسخة كاملة من المتغير.

## ٢- تمرير بالمرجع (Call By Reference)

In the **call-by-reference** method, the **Memory Address (Reference)** of the actual parameter is passed to the function, allowing direct access and modification of the original values. The actual and the formal parameters **point to the same memory address**. Any changes made to the parameters within the function are directly reflected in the original values outside the function.

في طريقة تمرير بالمرجع (Call By Reference)، يتم تمرير عنوان الذاكرة (Memory Address) (المرجع Reference) للمعامل الفعلي إلى الدالة، مما يسمح بالوصول المباشر وتعديل القيم الأصلية. يشير المعامل الفعلي و المعامل الشكلي إلى نفس عنوان الذاكرة. أي تغييرات يتم إجراؤها على المعاملات داخل الدالة تعكس مباشرة على القيم الأصلية خارج الدالة.

### شرح مبسط:

- **تمرير مرجع، وليس نسخة:** في هذه الطريقة، لا يتم نسخ قيمة المعامل الفعلي، بل يتم تمرير مرجع (أو عنوان) إلى موقعه في الذاكرة.
- **وصول مباشر:** داخل الدالة، يمكن الوصول إلى المعامل الأصلي مباشرة عبر المرجع، مما يسمح بإجراء تغييرات عليه.
- **تأثير على الأصل:** أي تعديلات تتم على المعامل داخل الدالة تعكس على المتغير الأصلي نفسه خارج الدالة، لأنهم يشيرون إلى نفس المكان في الذاكرة.
- يتم تحقيق ذلك عادةً عن طريق تحديد كلمة رئيسية مرجعية بشكل صريح، مثل & .

### مثال توضيحي 1:

```
void add_one(int& x)
{
    x++;
    cout << x << endl;
}

int main()
{
    int num = 5;
    add_one(num);
    cout << num << endl;           // يطبع 6
}
```

في هذا المثال، يتم تمرير المتغير num بالمرجع إلى دالة add\_one(). تقوم الدالة بزيادة قيمة x بمقدار 1، مما يؤدي إلى زيادة قيمة المتغير الأصلي num إلى 6.

### ؟ اختيار الطريقة المناسبة:

يعتمد اختيار call by ref أو call by value على الاحتياجات المحددة لدالتك وكيفية رغبتك في التعامل مع المعاملات. بشكل عام، يفضل call by value للبساطة وحماية البيانات، بينما يكون call by ref مفيداً عند تعديل المعاملات أو مشاركة هيكل البيانات الكبيرة بكفاءة.

### ؟ بعض الأمثلة على استخدام call by value :

- تمرير ثوابت إلى الدوال.
- تمرير المتغيرات التي لا تحتاج إلى تعديل داخل الدالة.
- حماية الدالة من تعديل المعاملات عن طريق الخطأ.

### ؟ بعض الأمثلة على استخدام call by ref :

- تمرير المتغيرات التي تحتاج إلى تعديل داخل الدالة.
- تنفيذ المتتبعين (tracers) أو المتغيرات المساعدة (helper variables) داخل الدوال.
- مشاركة هيكل البيانات الكبيرة بين الدوال بكفاءة.

## Difference between the Call by Value and Call by Reference in C++

Feature	Call by Value	Call by Reference
Value Passed	In this method, the value of the variable is passed to the function.	In call by reference memory address of the variable is passed.
Scope of Changes	In this method, the original value remains unchanged even when we make changes in the function.	In this method, the changes made are reflected in the original variable.
Performance	It may require extra memory and time to copy so less efficient.	It is more memory and time efficient as compared to "call by value".
Memory Location	The memory addresses of the actual and formal parameters are different.	The actual and the formal parameters point at the same memory address.
Applications	Mainly used to pass values for small data or when we do not want to change original values.	It is used when we want to modify the original value or save resources.

### الفرق بين تمرير بالقيمة وتمرير بالمرجع في C++

الصفة	تمرير بالقيمة	تمرير بالمرجع
القيمة المُمُرّرة	يتم تمرير قيمة المتغير إلى الدالة.	يتم تمرير عنوان الذاكرة للمتغير إلى الدالة.
نطاق التغييرات	تبقي القيمة الأصلية للمتغير دون تغيير حتى عند إجراء تغييرات داخل الدالة.	تحعكس التغييرات التي تتم داخل الدالة على المتغير الأصلي.
الأداء	قد يتطلب ذاكرة ووقتاً إضافياً لنسخ القيمة، لذلك يكون أقل كفاءة في بعض الحالات.	أكثر كفاءة من حيث الذاكرة والوقت مقارنة بتمرير بالقيمة.
موقع الذاكرة	عنواناً الذاكرة للمعامل الفعلي والمعامل الشكلي مختلفان.	يشير كلا المعاملين الفعلي والشكلي إلى نفس عنوان الذاكرة.
التطبيقات	يستخدم بشكل أساسى لتمرير القيم الصغيرة أو عندما لا نريد تغيير القيم الأصلية.	يستخدم عندما نريد تعديل القيمة الأصلية أو توفير الموارد.

فيما يلي جدول يوضح الاختلافات الرئيسية بين **:call by ref** و **call by value**

<b>call by reference</b>	<b>call by value</b>	<b>الميزة</b>
مرجع مباشر للمعامل	نسخة من المعامل	<b>الوصول</b>
يتم تعديل المعامل الأصلي	يظل المعامل الأصلي دون تغيير	<b>التغيير</b>
قد يكون أبطأ قليلاً بسبب النفقات العامة لمعالجة المراجع	بشكل عام أسرع	<b>الأداء</b>
تعديل المعاملات، تنفيذ المتبعين، مشاركة هيكل البيانات الكبيرة بكفاءة	بيانات بسيطة، تمرين الثوابت، حماية الدالة من تعديل المعاملات	<b>حالات الاستخدام</b>

## Without Reference

### Program

```
#include <iostream>
using namespace std;

void Function1(int x)
{
    x++;
}

int main()
{
    int a = 10;

    Function1(a);
    cout << a << endl;

    return 0;
}
```

### Memory

Name: a	10
Address:	000000469851FC54
Name: x	11
Address:	000000469851FD31

## With Reference

### Program

```
#include <iostream>
using namespace std;

void Function1(int &x)
{
    x++;
}

int main()
{
    int a = 10;

    Function1(a);
    cout << a << endl;

    return 0;
}
```

### Memory

Name: a , x

11

Address:

000000469851FC54

```

#include <iostream>
using namespace std;

void Number1(int x)
{
    x++;

    cout << "\n===== This is By Value =====" << endl;
    cout << "\nThis is show the value of parameter x: " << x << endl;
    cout << "This is show the reference of parameter x: " << &x << endl << endl;
}

void Number2(int& y)
{
    y++;

    cout << "===== This is By Reference & =====" << endl;
    cout << "\nThis is show the value of parameter y: " << y << endl;
    cout << "This is show the reference of parameter y: " << &y << endl << endl;
}

int main()
{
    int a = 10;
    int b = 20;

    Number1(a);

    cout << "This is show the value of parameter a: " << a << endl;
    cout << "This is show the reference of parameter a: " << &a << endl;
    cout << "\n-----\n\n";

    Number2(b);

    cout << "This is show the value of parameter b: " << b << endl;
    cout << "This is show the reference of parameter b: " << &b << endl << endl;

    // system("pause");
}

```

```

Microsoft Visual Studio Debug Console
===== This is By Value =====
This is show the value of parameter x: 11
This is show the reference of parameter x: 000000692F0FF770

This is show the value of parameter a: 10
This is show the reference of parameter a: 000000692F0FF794

-----
===== This is By Reference & =====
This is show the value of parameter y: 21
This is show the reference of parameter y: 000000692F0FF7B4

This is show the value of parameter b: 21
This is show the reference of parameter b: 000000692F0FF7B4

```



# Lesson #34 - Creating References

## Creating References:

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting ‘&’ in the declaration.

عندما يتم إعلان متغير ك(**reference**) ، فإنه يصبح اسمًا بديلاً لمتغير موجود بالفعل. يمكن إعلان متغير ك(**reference**) عن طريق وضع الرمز "&" في الإعلان.

ببساطة، يمكنك التفكير في المتغير المرجعى كأنه "نفس" المتغير الأصلي، ولكن باسم جديد. أي تغييرات تتم على المتغير المرجعى ستنتعكـس على المتغير الأصلي والعكس صحيح.

**ملخص:** إعلان متغير كمرجع يمنحك وسيلة إضافية للوصول إلى متغير موجود بالفعل دون إنشاء متغير جديد.

Also, we can define a reference variable as a type of variable that can act as a reference to another variable. ‘&’ is used for signifying the address of a variable or any memory. Variables associated with reference variables can be accessed either by its name or by the reference variable associated with it.

بالإضافة إلى ما شرحناه، يمكننا أيضًا تعريف متغير مرجعي كنوع خاص من المتغيرات يلعب دور الوسيط للوصول إلى متغير آخر موجود بالفعل. يُستخدم رمز & للإشارة إلى عنوان ذاكرة المتغير أو أي مكان آخر في الذاكرة. ويمكن الوصول إلى المتغيرات المرتبطة بالمتغيرات المرجعية إما باستخدام اسمها الأصلي أو باستخدام اسم المتغير المرجعي نفسه.

معنى آخر:

- عند تعريف متغير مرجعي، فإننا نحدد نوعاً خاصاً له مثل `&int` أو `&double`، وهذا يعني أنه يمكنه فقط الإشارة إلى متغيرات من هذا النوع.
- **المتغير المرجعي:** هو نوع خاص من المتغيرات التي تحمل مرجعًا (عنوانًا) ، لا يحتوي على قيمة خاصة به، ولكنه بمثابة "جسر" يصلنا إلى قيمة متغير آخر موجود. هذا يعني أن أي تعديل يتم إجراؤه على المتغير المرجعي سينعكس على المتغير الأصلي الذي يشير إليه.
- رمز & يُستخدم كـ "**معرف**" لـ **مكان القيمة في الذاكرة**، وليس القيمة نفسها.
- يمكنك الوصول إلى القيمة التي يشير إليها المتغير المرجعي باستخدام اسم المتغير الأصلي أو اسم المتغير المرجعي.

تخيل أن لديك متغيراً اسمه "num" ويحتوي على القيمة 5.  
إذا أعلنت متغيراً جديداً وسميته "ref" باستخدام "& ref" ، فإن "ref" يصبح اسمًا بديلاً لـ "num". أي تعديل يتم إجراؤه على "ref" داخل الدالة أو أي سطر من الكود، سيعكس نفس التغيير على "num" مباشرة. بمعنى آخر: "ref" هو مثل اختصار لمتغير موجود بالفعل "num" ، وليس متغيراً منفصلاً يحتوي على قيمته الخاصة.

### مثال توضيحي 1 :

```
int num = 5;
int& ref = num;           الآن 'ref' هو مرجع ل'num'

cout << "num: " << num << endl    // سيعطي 5
cout << "ref: " << ref << endl    // سيعطي 5 أيضاً

ref++;                     // زيادة قيمة ref

cout << "num: " << num << endl    // لان num , ref يشيران الى نفس المكان في الذاكرة - سيعطي 6
```

### مثال توضيحي 2 :

```
int main()
{
    int x = 10;
    int& ref = x;           // ref is a reference to x.

    ref = 20;                // Value of x is now changed to 20
    cout << "x = " << x << '\n';

    x = 30;                  // Value of x is now changed to 30
    cout << "ref = " << ref << '\n';

    return 0;
}
```

---

## حالات استخدام المتغيرات المرجعية:

- تمرير المتغيرات كمرجع إلى الدوال (يسمح بتعديل القيم الأصلية)
- إنشاء أسماء بديلة للمتغيرات لتسهيل الاستخدام
- مشاركة المعلومات بين أجزاء مختلفة من البرنامج
- إنشاء هياكل بيانات معقدة حيث يمكن استخدام المراجع لتأشير إلى نفس المتغير في عدة أماكن.
- توفير الذاكرة والوقت عند التعامل مع قيم كبيرة.

## مميزات استخدام المتغيرات المرجعية كوسطاء:

- تجنب نسخ القيمة:** بدلاً من نسخ قيمة المتغير الأصلي، يشير المتغير المرجعي إلى نفس القيمة، مما يوفر الذاكرة والوقت.
- تعديل القيم الأصلية مباشرة:** أي تغيير يتم إجراؤه على متغير مرجعي سينعكس على المتغير الأصلي، مما يسهل التلاعب بالقيمة الأصلية.
- سهولة الوصول:** يوفر طريقة مختصرة للوصول إلى قيمة متغير آخر دون الحاجة لمعرفة عنوانه الدقيق في الذاكرة.
- تجنب التكرار:** يمكنك استخدام اسم المتغير المرجعي كاختصار لتجنب كتابة الاسم الأصلي للمتغير بشكل متكرر، مما يحسن قابلية قراءة الكود.
- كفاءة الأداء:** في بعض الحالات، يمكن أن يوفر استخدام المتغيرات المرجعية كوسطاء كفاءة في الذاكرة والوقت، خاصة عند التعامل مع هياكل بيانات معقدة أو قيم كبيرة.

## يمكن الوصول إلى المتغيرات المرتبطة بالمتغيرات المرجعية بطريقتين:

- باستخدام اسم المتغير الأصلي.
- باستخدام اسم المتغير المرجعي.

## ملاحظات:

- يجب أن يشير المتغير المرجعي دائمًا إلى متغير موجود بالفعل وليس لإنشاء متغيرات جديدة. لا يمكن للمتغيرات المرجعية أن تكون "معلقة" دون إشارة.
- لا يمكن تغيير مرجع المتغير المرجعي بعد تحديده، أي لا يمكنك "إعادة توجيهه" إلى متغير آخر.

### Program

```
#include <iostream>
using namespace std;

int main()
{
    int a = 10;
    int & x = a;

    cout << &a << endl;
    cout << &x << endl;

    cout << a << endl;
    cout << x << endl;

    return 0;
}
```

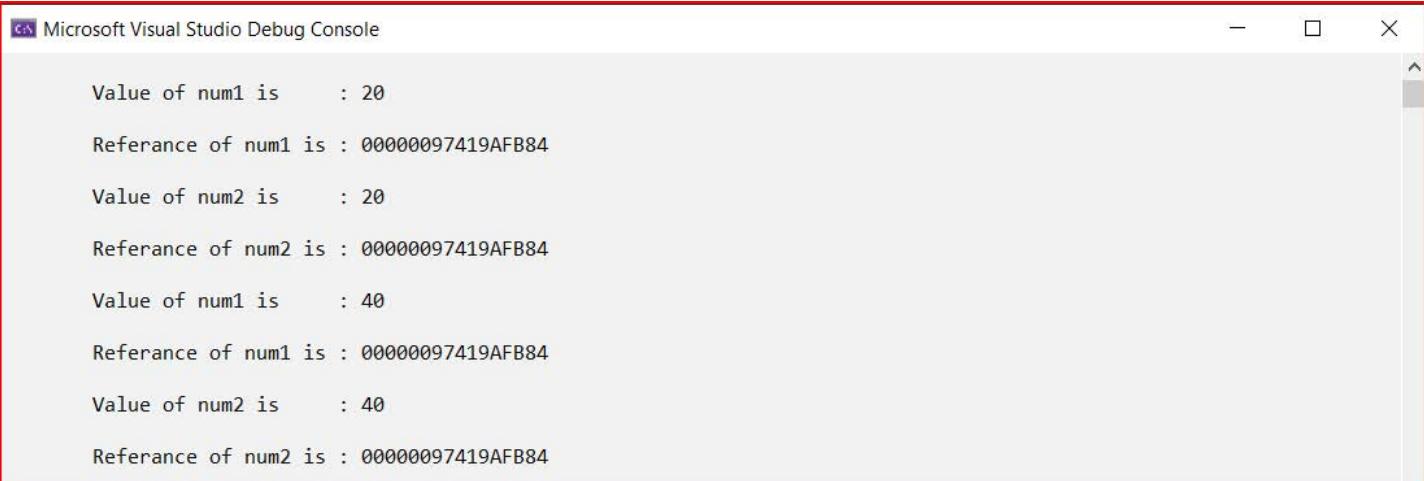
### Memory

Name: a , x  
10  
Address:  
000000469851FC54

```
int a = 3  
int &x = a
```

اسم دلع للفيربول ، x يلتصق به a

```
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
    int num1 = 20;  
  
    int& num2 = num1;  
  
    cout << "Value of num1 is : " << num1 << endl;  
    cout << "Reference of num1 is : " << &num1 << endl;  
  
    cout << "\nValue of num2 is : " << num2 << endl;  
    cout << "Reference of num2 is : " << &num2 << endl;  
  
    num2 = 40;  
  
    cout << "\nValue of num1 is : " << num1 << endl;  
    cout << "Reference of num1 is : " << &num1 << endl;  
  
    cout << "\nValue of num2 is : " << num2 << endl;  
    cout << "Reference of num2 is : " << &num2 << endl;  
  
    return 0;  
}
```



```
Microsoft Visual Studio Debug Console  
  
Value of num1 is : 20  
Reference of num1 is : 00000097419AFB84  
  
Value of num2 is : 20  
Reference of num2 is : 00000097419AFB84  
  
Value of num1 is : 40  
Reference of num1 is : 00000097419AFB84  
  
Value of num2 is : 40  
Reference of num2 is : 00000097419AFB84
```

## تطبيقات المتغيرات المرجعية (References) في C++

تقديم المتغيرات المرجعية في C++ مزايا عديدة، ومن أهم تطبيقاتها:

### 1. تعديل قيم المعاملات داخل الدوال:

إذا استقبلت دالة مرجعاً لمتغير، فيمكنها تعديل قيمة المتغير الأصلي نفسه.  
لا ينشئ تمرير المتغير بالمرجع نسخة مستقلة عنه، بل يوفر وصولاً مباشراً إلى المتغير الأصلي.  
أي تغييرات تتم على المرجع داخل الدالة ستنعكس على المتغير الأصلي خارجها.

مثال توضيحي: تبديل قيم متغيرين باستخدام المراجعات:

```
معاملات مرجعية y و x // x = y, y = x
```

```
void swap(int& x, int& y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}
```

```
int main() {  
    int a = 5, b = 10;  
    cout << "Before Swap : " << a = " << a << ", b = " << b << endl;  
    swap(a, b); // تمرير a و b بالمرجع  
    cout << "After Swap : " << a = " << a << ", b = " << b << endl;  
}
```

الشرح:

الدالة `swap` تستقبل معاملين مرجعيين `x` و `y`.  
داخل الدالة، يتم تبديل القيم باستخدام متغير مؤقت `temp`.  
عند استدعاء الدالة (`swap(a, b)` في `main`)، يتم تمرير المتغيرين `a` و `b` بالمرجع، مما يسمح للدالة بالوصول المباشر إلى قيمهما الأصلية.  
بعد التبديل داخل الدالة، تعكس القيم الجديدة للمتغيرين `x` و `y` على المتغيرين الأصليين `a` و `b`.  
نتيجة لذلك، سيطّبع البرنامج أن قيم `a` و `b` قد تبدلت بالفعل.

## 2. توفير الذاكرة والوقت بتجنب نسخ هياكل بيانات كبيرة:

تخيل دالة تحتاج إلى استقبال كائن كبير الحجم. إذا مررتها هذا الكائن دون استخدام مرجع، فإن نسخة جديدة تماماً منه ستُنشأ داخل الدالة، وهذا بدوره يُهدى وقت معالج المعالج (CPU) والذاكرة. لحل هذه المشكلة، يمكننا استخدام المراجع لتجنب هذا النسخ غير الضروري.

### إليك التفاصيل:

عند تمرير كائن كبير بالاسم (دون مرجع)، فإن الدالة لا تتلقى الكائن الأصلي نفسه، بل نسخة جديدة كاملة عنه. إنشاء هذه النسخة يتطلب وقتاً طويلاً من المعالج ومساحة إضافية في الذاكرة.

باستخدام المراجع، نمرر فقط عنوان الكائن الأصلي للدالة، مما يعني أنها تعمل مباشرة على هذا الكائن ولا تحتاج إلى عمل نسخة. وبالتالي، نوفر وقت المعالج والمساحة في الذاكرة.

### مثال توضيحي:

```
class LargeData{  
    كود معقد يمثل بيانات كبيرة ... //  
};  
  
void processData(const LargeData& data) {  
    معالجة البيانات مباشرة (بدون نسخ) ... //  
}  
  
int main() {  
    LargeData bigData;           // إنشاء كائن كبير  
    processData(bigData);        // تمرير الكائن بالمرجع  
}
```

### في هذا المثال:

دالة `processData` تتلقى كائن `LargeData` باستخدام مرجع ثابت (`&const LargeData`)، مما يضمن عدم تعديله داخل الدالة.

عندما تمرر `bigData` للدالة، فإنها تتلقى فقط عنوان الكائن وليس نسخة منه، وبالتالي تعالج البيانات الأصلية مباشرةً ويوفر استهلاك الذاكرة ومعالجة النسخ.

باختصار، استخدام المراجع لتدمير الهياكل الكبيرة يُقلل بشكل كبير من استهلاك الذاكرة ووقت المعالج، ويحسن كفاءة برامجك بشكل ملحوظ.

### 3. تعديل عناصر قائمة "ForEach" بشكل مباشر:

عند استخدام حلقة "ForEach" للتنقل عبر قائمة من العناصر، فإن المتغيرات المرجعية تتيح تعديل العناصر مباشرة داخل الحلقة. بدونه، ستنشأ نسخة من كل عنصر داخل الحلقة، وهو أمر غير ضروري إذاً كنا نريد فقط تعديل العناصر الأصلية.

مثال توضيحي 1:

```
vector<int> numbers {1,2,3};  
for (int& number : numbers) {  
    number *= 2; // مضاعفة قيمة كل عنصر في القائمة الأصلية  
}  
//
```

مثال توضيحي 2:

```
int main()  
{  
    vector<int> vect{ 10, 20, 30, 40 };  
    // We can modify elements if we use reference  
  
    for (int& x : vect) {  
        x = x + 5;  
    }  
  
    // Printing elements  
    for (int x : vect) {  
        cout << x << " ";  
    }  
    cout << '\n';  
    return 0;  
}
```

### 4. منع نسخ عناصر قائمة "ForEach" لتوفير الذاكرة:

في بعض الحالات، قد لا نحتاج إلى تعديل العناصر ضمن حلقة "ForEach"، بل نحتاج فقط إلى الوصول إلى قيمها. عند استخدام متغير مرجعي، نتجنب إنشاء نسخة عن كل عنصر، مما يوفر الذاكرة.

#### 4. منع نسخ عناصر قائمة "ForEach" لتوفير الذاكرة:

في بعض الحالات، قد لا تحتاج إلى تعديل العناصر ضمن حلقة "ForEach"، بل تحتاج فقط إلى الوصول إلى قيمها. عند استخدام متغير مرجعي، نتجنب إنشاء نسخة عن كل عنصر، مما يوفر الذاكرة.

مثال توضيحي 1:

```
vector<string> names{"Alice", "Bob", "Charlie"};
for (const string& name : names) {
    cout << name << " ";
}
```

طباعة أسماء القائمة بدون نسخ العناصر //

مثال توضيحي 2:

```
int main()
{
    // Declaring vector
    vector<string> vect
    { "geeksforgeeks practice",
      "geeksforgeeks write",
      "geeksforgeeks ide"
    };
    // We avoid copy of the whole string
    // object by using reference.
    for (const auto& x : vect)
    {
        cout << x << '\n';
    }
    return 0;
}
```

هذه مجرد نماذج قليلة لتطبيقات المتغيرات المرجعية في C++. استخدامها الصحيح يمكن أن يحسن أداء وكفاءة أ��وادق ويوفر لك إمكانيات أكبر للتعامل مع البيانات بشكل مرن.



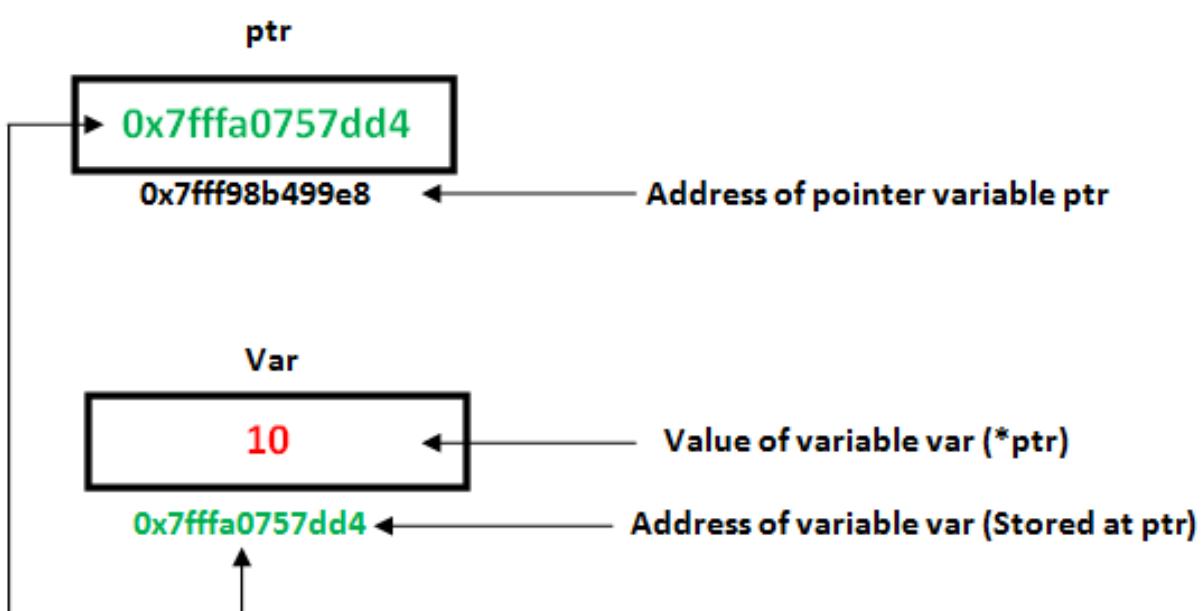
# Lesson #35 - What is Pointer

## الـ :Pointers !

- المؤشرات تعطي الوصول الكامل (full control) للذاكرة والتحكم الكامل فيها .(Full control)
- لغات البرمجة الأخرى لا تدعم المؤشرات لكنها موجود فيها بالـ **Background** ، ولا تسمح كـ **Developer** باستخدام الـ **Pointer** ، لكن لغتي البرمجة **C++** و **C** يسمحان باستخدامها.
- تعتبر المؤشرات **Pointers** من أقوى الميزات التي تمتاز بها لغة **C++** عن باقي اللغات المشهورة مثل جافا، فيجوال بيسك ...
- لفهم عمل المؤشرات واستخدامها لابد أولاً أن يكون لديك معرفة عن ما هو العنوان **Address** في ذاكرة الحاسوب.
- تُقسم ذاكرة الحاسوب إلى أجزاء من **bytes** وكل بait لديه العنوان الخاص به على سبيل المثال في ذاكرة حجمها **KB 1** يوجد **bytes 1024** وكل بait يعطي عنوان (أي أن مجال العناوين من 0 - 1023).
- فالمؤشرات (**Pointers**) إذاً هي متغيرات (**Variables**) تخزن عناوين (**Address**) متغيرات أخرى في الذاكرة بالـ **primitive** النظام السنتعشري، يمكن للمؤشرات أن تؤثر على متغير من نوع **hexadecimal** **object** أو مصفوفة **array** أو غرض **array**.

## تذكر :

- تشير المؤشرات إلى **عناوين** في الذاكرة، وليس **القيم** مباشرةً.
- يجب التعامل مع المؤشرات **بحذر** لتجنب الأخطاء البرمجية المحتملة.
- يعتبر استخدام المؤشرات من المفاهيم الأساسية في لغات البرمجة مثل **C** و **C++**.



```
int *pc, c;  
c = 5;
```



&c = 0x7fff5fbff80c  
c = 5

```
pc = &c;
```



pc = 0x7fff5fbff80c  
\*pc = 5

```
c = 11;
```



pc = 0x7fff5fbff80c  
\*pc = 11

```
*pc = 2;
```



&c = 0x7fff5fbff80c  
c = 2

Name: a



Address:  
000000469851FC54

هذه العملية اسمها **Referencing** لـ **Pointer**

(يعني خلية البوينتر يؤشر على الفاريبول)

Name: p



Address:  
000000FD98533A54

## ؟ كيف تستخدم المؤشر؟

### 1. تعريف متغير المؤشر:

- حدد نوع البيانات الذي سيشير إليه المؤشر (مثل int أو string).
- استخدم رمز النجمة \* - Asterisk قبل اسم المتغير للإشارة إلى أنه مؤشر.

مثال:

int\* ptr; (تعريف متغير ptr كمؤشر يشير إلى متغيرات من نوع int)

### 2. تهيئة (تعيين) عنوان متغير للمؤشر:

- تعيين عنوان متغير موجود باستخدام المشغل الأحادي & :
- اسند هذا العنوان لمتغير المؤشر باستخدام علامة المساواة = .

مثال:

int x = 10; (تعريف متغير x بقيمة 10)  
int\* ptr = &x; (تعيين عنوان x للمؤشر)

```
#include <iostream>
using namespace std;

void WhatIsPointer1()
{
    int a = 10;

    cout << "\n\t The Value of a is : " << a << endl;
    cout << "\n\t The Address of a is : " << &a << endl;

    int* p;
    p = &a;

    // a هنا قمنا بطباعة القيمة الموجودة في المؤشر p والتي هي عنوان المتغير
    cout << "\n\t The value in p, it's address of a: (use p): " << p << endl << endl;

    // هنا قمنا بطباعة عنوان المؤشر p في الذاكرة
    cout << "\n\tThe address of P in memory : (use &p): " << &p << endl << endl;
}

int main()
{
    WhatIsPointer1();

    cout << endl;

    return 0;
}
```



```
Microsoft Visual Studio Debug Console
The Value of a is : 10
The Address of a is : 000002B9F6FF934
The value in p, it's address of a: (use p): 000002B9F6FF934
The address of P in memory : (use &p): 000002B9F6FF958
```

- يمكن تغيير اتجاه المؤشر لأي شيء من نفس النوع (مثل int أو string) في أي مكان بالبرنامج.
- مثال:** عندما متغير a من نوع int يكون اتجاه المؤشر (ptr = &a) ، وإذا كان هناك متغير آخر b من نوع int فيمكن أن نجعل المؤشر يؤشر له كالتالي: (ptr = &b)

```
#include <iostream>
using namespace std;

void WhatIsPointer1()
{
    int a = 10;

    cout << "\n\t The Value of a is : " << a << endl;
    cout << "\n\t The Address of a is : " << &a << endl;

    int* p;
    p = &a;

    // a هنا قمنا بطباعة القيمة الموجودة في المؤشر p والتي هي عنوان المتغير
    cout << "\n\t The value in p, it's address of a: (use p): " << p << endl << endl;

    // p هنا قمنا بطباعة عنوان المؤشر p في الذاكرة
    cout << "\n\t The address of P in memory : (use &p) : " << &p << endl << endl;

    int b = 30;

    p = &b;           // تغيير اتجاه المؤشر إلى متغير آخر

    cout << "\n\t The Value of b is : " << b << endl;
    cout << "\n\t The Address of b is : " << &b << endl;

    // b هنا قمنا بطباعة القيمة الموجودة في المؤشر p والتي هي عنوان المتغير الجديد
    cout << "\n\t The value in p, it's address of a: (use p): " << p << endl << endl;

}

int main()
{
    WhatIsPointer1();

    cout << endl;

    return 0;
}
```

```
The Value of a is : 10
The Address of a is : 0000000F66CFF8D4
The value in p, it's address of a: (use p): 0000000F66CFF8D4
The address of P in memory : (use &p) : 0000000F66CFF8F8
The Value of b is : 30
The Address of b is : 0000000F66CFF914
The value in p, it's address of a: (use p): 0000000F66CFF914
```



# Lesson #36 - Dereferencing Pointer

## الوصول إلى قيمة المتغير الذي يشير إليه المؤشر : Dereferencing Pointer

أمر أساسى لاستخدام المؤشرات في لغات البرمجة مثل C و C++.  
يسمح لك Dereferencing Pointer بالوصول إلى القيمة المخزنة في العنوان الذي يشير إليه المؤشر، والتي يمكن أن تكون مفيدة للعديد من المهام، مثل:

- معالجة البيانات
- إنشاء هيكل بيانات معقدة
- تخصيص الذاكرة ديناميكياً

يُستخدم رمز (\*) "نطلق عليه مجازاً مفتاح" للوصول إلى القيمة المخزنة في العنوان الذي يشير إليه المؤشر.  
على سبيل المثال: إذا كان لدينا مؤشر يسمى ptr يشير إلى متغير x من نوع int، فإن ptr \* سيعيد قيمة x.

```
int x = 10;
int* ptr = &x;                                // ptr points to x
int y = *ptr;                                 // Dereferences ptr to get the value of x
printf("%d\n", y);      // Prints 10        // Prints the value of x
```

في هذا المثال: يشير المؤشر ptr إلى المتغير x.  
باستخدام (\*) يمكننا الوصول إلى قيمة x المخزنة في العنوان الذي يشير إليه المؤشر.

إلى Dereferencing ضروري للوصول إلى البيانات المخزنة في الذاكرة. بدونه لن تتمكن البرامج من الوصول إلى البيانات المخزنة في الذاكرة.

؟ فيما يلي بعض الملاحظات حول استخدام (\*) Asterisk للوصول إلى قيمة المتغير الذي يشير إليه المؤشر:

- يجب أن يكون المؤشر معرفاً ومؤقتاً بالقيمة قبل استخدامه للوصول إلى قيمة المتغير.
- يجب أن يكون نوع المؤشر متوافقاً مع نوع المتغير الذي يشير إليه.
- يجب أن يكون العنوان الذي يشير إليه المؤشر صالحًا.

```

#include <iostream>
using namespace std;

void HowToMakePointer()
{
    // "Arabic" وقيمة language
    string language = "Arabic";
    // هنا قمنا بتعريف مؤشر لعنوان المتغير language في الذاكرة
    string* ptr = &language;
    // language هنا قمنا بطباعة قيمة المتغير
    cout << "language = " << language << endl;
    // language هنا قمنا بطباعة عنوان المتغير في الذاكرة
    cout << "Address of language in memory = " << &language << endl;
    // ptr هنا قمنا بطباعة عنوان المؤشر في الذاكرة
    cout << "Address of ptr in memory      = " << &ptr << endl;
    // ptr والتي هي عنوان المتغير language هنا قمنا بطباعة القيمة الموجودة في المؤشر
    cout << "Value of ptr in memory      = " << ptr << endl;
    // ptr هنا قمنا بطباعة القيمة التي يشير إليها عنوان المؤشر ptr في الذاكرة و التي هي قيمة المتغير language
    cout << "Value that ptr point to     = " << *ptr << endl;
    // language هنا قمنا بتغيير قيمة المتغير من خلال المؤشر
    *ptr = "English";
    // ptr هنا قمنا بطباعة القيمة الجديدة التي يشير إليها عنوان المؤشر ptr في الذاكرة و التي هي قيمة المتغير
    cout << "Value that ptr point to     = " << *ptr << endl;
    cout << endl;
    // ptr هنا قمنا بإنشاء متغير جديد من نفس النوع
    string Subject = "C++";
    // ptr هنا قمنا بتغيير اتجاه المؤشر لهذا المتغير الجديد
    ptr = &Subject;
    cout << "Subject = " << Subject << endl;
    cout << "Address of Subject in memory = " << &Subject << endl;
    cout << "Value of ptr in memory      = " << ptr << endl;
    cout << "Value that ptr point to     = " << *ptr << endl;
}

int main()
{
    HowToMakePointer();
    cout << endl;
    return 0;
}

```

Microsoft Visual Studio Debug Console

```

language = Arabic
Address of language in memory = 00000025729BFB98
Address of ptr in memory      = 00000025729BFBD8
Value of ptr in memory        = 00000025729BFB98
Value that ptr point to       = Arabic
Value that ptr point to       = English

Subject = C++
Address of Subject in memory = 00000025729BFBF8
Value of ptr in memory        = 00000025729BFBF8
Value that ptr point to       = C++

```



## Lesson #37 - Common Mistakes with Pointers

الاخطاء التي يمكن ان تحدث عند كتابة كود البوينتر

### ⚠ نصيحة امنية عند استخدام Dereferencing Pointer

يجب توخي الحذر عند استخدام **Pointer** حيث يمكن أن يؤدي إلى أخطاء خطيرة إذا لم يتم استخدامه بشكل صحيح على سبيل المثال، إذا كان المؤشر يشير إلى عنوان غير صالح، فقد يؤدي ذلك إلى حدوث عطل في البرنامج ، امثلة:

int var, \*varPoint;

**Wrong!** : varPoint is an address but var is not

varPoint = var;

الخطأ: تخزن **Pointer** في **Value**

**Correct!** : varPoint is an address and so is &var

الصح: تخزن **Address** في **Pointer**

varPoint = &var;

**Wrong!** : & : var is an address

الخطأ: لا تأخذ **Address of Variable** **Value of pointer**

\* : varPoint is the value stored in &var

\*varPoint = &var;

**Correct!** : both \*varPoint and var are values

الصح: يأخذ **Value of pointer** **Value of Variable**

\*varPoint = var;

```
1  #include <iostream>
2
3  using namespace std;
4
5
6  int main()
7  {
8      int x, * p;
9
10
11     p = x;      //Wrong : p is an address but x is value
12     p = &x;      //Correct: p is an address and &x is address too
13
14     *p = &x;    //Wrong: *p is the value stored in x but &x is an address
15     *p = x;     //Correct: *p is value and x too
16
17 }
```



# Lesson #38 - Pointers vs References

## References VS Pointers

أخذنا قبل كده الـ **pointer** وقتنا انو بيشيل عنوان درج تانى.. زى كده

```
int number = 3;
int *pointer = &number;
```

وممكن اختيار القيمة اللي جوه الدرج اللي اسمه **number** عن طريق الـ **pointer** ده ... كده يعني

```
*pointer = 4;
cout << number;
```

\* معاناها هنا القيمة اللي عند العنوان اللي **pointer** شايله خليه بـ ؟  
ده بقه الـ ... **pointer** نفكير الـ **Reference** كده  
قلتنا ان الـ **pointer** أكثرك بتدي اسم تانى للدرج ... مثال:

```
int number = 3;
int &newName = number;
```

كده انا ممكن انادى على الدرج اللي اسمه **number** بالاسم الجديد اللي هو الـ **newname** ... زى كده  
**newName = 4;**  
cout << number << endl;

فاحنا بنقول خلى الـ **newName** بـ ؟ والـ **newName** ده أصلا اسم تانى للدرج اللي اسمه **number** بـ ؟  
فكتنا بنقول خلى الـ **number**

حلو... طيب ايه الفرق بقية ما بين الـ **Pointers** و الـ **References** ؟

أول حاجة:

احنا ممكن نجز **pointer** من غير منملأه بأى عنوان أو من غير مشاور على حاجة زى كده  
**int \*pointer;**

بس مينفعش تعمل كده في الـ **References** يعني مينفعش تقول كده

**int &newName;**

الـ **code** ده مش هيستغل و هيقولك لازم تقوله الـ **newName** ده اسم تانى للدرج ايه ؟؟  
فلازم تعمل كده مثلا

```
int number;
int &newname = number;
```

و طبعا في الـ **declaration** بتاع الـ **pointer** بنسخدم \* زى كده  
**int \*pointer;**

لكن في الـ **Reference** بنسخدم & زى كده

**int &newname = number;**

ثانية حاجة:

**cout << \*pointer;**

لما تجيب القيمة اللي بيشاور عليها الـ **pointer** بنسخدم الـ \*

**cout << newName;**

لكن في الـ **Reference** بنادي عليه عادي جدا

ثالث حاجة:

وانت بتعامل مع الـ **pointer** ممكن يكون مش بيشاور على حاجة مثلا ..

لكن الـ **Reference** هو دايما بيبيقى اسم تانى لـ **object** موجود فعلا (valid Object)

## Program

```

int a = 10;
int& x = a;

cout << &a << endl;
cout << &x << endl;

cout << a << endl;
cout << x << endl;

int* p = &a; البيونتر

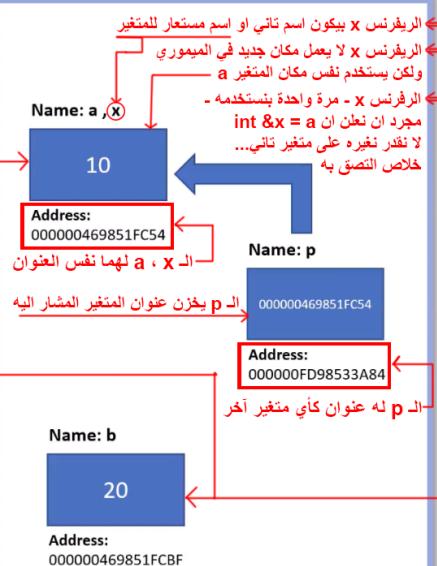
cout << p << endl;
cout << *p << endl;

int b = 20;
p = &b; البيونتر

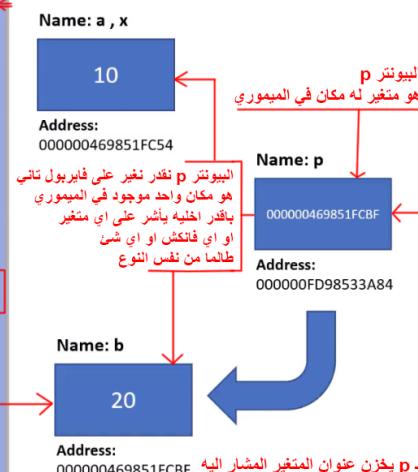
cout << p << endl;
cout << *p << endl;

```

## Memory With Reference



## Memory With Pointer



```

#include <iostream>

using namespace std;

int main()
{
    int a = 10;
    int& x = a;

    cout << "Reference of a : " << &a << endl;
    cout << "Reference of x : " << &x << endl;
    cout << "\n-----\n";
    cout << "Value of a : " << a << endl;
    cout << "Value of x : " << x << endl;
    cout << "\n-----\n";

    int* p;
    p = &a;

    cout << "This is value in pointer p - Reference of a : " << p << endl;
    cout << "This is Value of reference which the pointer to point to in p : " << *p << endl;
    cout << "\n-----\n";
    int b = 20;
    p = &b;

    cout << "This is value in pointer p - Reference of b : " << p << endl;
    cout << "This is Value of reference which the pointer to point to in p : " << *p << endl;
    cout << "\n-----\n";

    return 0;
}

```

```

Reference of a : 00000826B18F4D4
Reference of x : 00000826B18F4D4
-----
Value of a : 10
Value of x : 10
-----
This is value in pointer p - Reference of a : 00000826B18F4D4
This is Value of reference which the pointer to point to in p : 10
-----
This is value in pointer p - Reference of b : 00000826B18F534
This is Value of reference which the pointer to point to in p : 20
-----
```

## جدول يوضح الفرق بين المراجعات (Pointers) و المؤشرات (References) في C++

الخصية	المراجعات (References)	المؤشرات (Pointers)
إعادة التعيين	لا يمكن إعادة تعيين المتغير (يشير دائمًا إلى المتغير الأصلي)	يمكن إعادة تعيينه ليشير إلى عنوان متغير آخر
عنوان الذاكرة	تشارك المراجعات نفس عنوان المتغير الأصلي	تمتلك المؤشرات عناوين ذاكرة خاصة بها
طريقة العمل	تشير المراجعات إلى متغير آخر بشكل مباشر	تخزن المؤشرات عنوان المتغير المشار إليه
قيمة فارغة (Null)	لا يمكن أن تأخذ المراجعات قيمة فارغة (null) للمؤشرات	يمكن تعين قيمة فارغة (null) للمؤشرات
تمرير المتغيرات للدوال	يتم تمرير المراجعات بالقيمة (نسخة من المتغير)	يتم تمرير المؤشرات بالمرجع (عنوان المتغير)
أمان الذاكرة	أكثر تعقيدًا وخطيراً لسوء الاستخدام	أكثر أماناً بسبب عدم السماح بإعادة التعين
أداء	قد يكون أسرع في بعض الحالات	عادةً يكون أبطأ قليلاً بسبب عملية الإلغاء التلقائي

### الفرق الرئيسي:

- المؤشرات (Pointers) متغيرات مستقلة لها عناوين ذاكرة خاصة بها.
- المراجعات (References) ليست متغيرات مستقلة، بل هي مجرد أسماء مستعارة لمتغيرات موجودة.

### استخدامات المؤشرات والمراجعات:

- تستخدم المؤشرات غالباً للتعامل مع الذاكرة الديناميكية، وتمرير المتغيرات بالإشارة إلى الدوال، وإنشاء هيكل بيانات معقدة.
- تستخدم المراجعات عندما تريدها تمرير متغير بالمرجع إلى دالة (تعديل المتغير الأصلي من داخل الدالة) وذلك لتجنب نسخ المتغيرات الكبيرة عند تمريرها إلى الدوال، ولتسهيل كتابة الكود وقراءته في بعض الحالات مما يشكل أماناً أكثر من المؤشرات.

### ملخص:

**المؤشرات و المراجعات** أدوات قوية تسمح بالوصول والتلاعب بالبيانات في الذاكرة بشكل مرن. فهم يمتلكون خصائص وقيوداً مختلفة، لذا فإن اختيار الأداة المناسبة يعتمد على احتياجات البرنامج وظروف الاستخدام.



# Lesson #39 - Call by Reference:

## Using pointers

الاستدعاء بالمرجع في C++ باستخدام المؤشرات - **C++ Call by Reference: Using pointers**

الاستدعاء **بالقيمة (Pass by value)**: في هذه الطريقة، يتم نسخ قيمة المتغير الأصلي **a** ، **b** إلى المتغير المؤقت **n1** ، **n2**. أي أن **أي تغيير يتم على المتغير داخل الدالة لا ينعكس على المتغير الأصلي Swap (local variable)**. مثال توضيحي:

```
void Swap(int n1, int n2) {  
    int tamp;  
    tamp = n1;  
    n1 = n2;  
    n2 = tamp; }  
  
int main() {  
    int a = 1, b = 2;  
    cout << "Before Swapping" << endl;  
    cout << "Value of a = " << a << endl;  
    cout << "Value of b = " << b << endl;  
    Swap(a, b);  
    cout << "After Swapping" << endl;  
    cout << "Value of a = " << a << endl;  
    cout << "Value of b = " << b << endl;  
}
```

الاستدعاء **بالمرجع (Pass by reference)**: في هذه الطريقة، يتم **تمرير عنوان المتغير الأصلي a** ، **b** إلى المتغير المؤقت **n1** ، **n2** في الدالة **Swap**. أي أن **أي تغيير يتم على المتغير داخل الدالة ينعكس على المتغير الأصلي**. مثال توضيحي:

```
void Swap(int &n1, int &n2) {  
    int tamp;  
    tamp = n1;  
    n1 = n2;  
    n2 = tamp; }  
  
int main() {  
    int a = 1, b = 2;  
    cout << "Before Swapping" << endl;  
    cout << "Value of a = " << a << endl;  
    cout << "Value of b = " << b << endl;  
    Swap(a, b);  
    cout << "After Swapping" << endl;  
    cout << "Value of a = " << a << endl;  
    cout << "Value of b = " << b << endl;  
}
```

## استخدام المؤشرات للاستدعاء بالمرجع

يمكن استخدام المؤشرات للاستدعاء بالمرجع. في هذه الحالة، يتم تمرير المؤشر إلى المتغير الأصلي **a** ، **b** إلى المتغير المؤقت **n1** ، **n2** في الدالة **Swap**. ثم يمكن للدالة الوصول إلى المتغير الأصلي وتعديله من خلال المؤشر.

```
void Swap(int *n1, int *n2) {  
    int tamp;  
    tamp = *n1;  
    *n1 = *n2;  
    *n2 = tamp; }  
  
int main() {  
    int a = 1, b = 2;  
    cout << "Before Swapping" << endl;  
    cout << "Value of a = " << a << endl;  
    cout << "Value of b = " << b << endl;  
    Swap(&a , &b);  
    cout << "After Swapping" << endl;  
    cout << "Value of a = " << a << endl;  
    cout << "Value of b = " << b << endl;  
}
```

**&a** is address of **a** // **&b** is address of **b**

**swap(&a , &b);**

هنا، يتم تمرير **عنوان المتغير (بدلاً من قيمته)** إلى الدالة أثناء استدعائهما. وبما أن **العنوان** يتم تمريره بدلاً من **القيمة**، يجب استخدام الرمز (\*) للوصول إلى القيمة المخزنة في ذلك العنوان.  
**n1** و **n2**\* يمثلان القيمة المخزنة في العنوانين **n1** و **n2** على التوالي.  
بما أن **n1** و **n2** يحتويان على عناوين المتغيرين **a** و **b** ، فإن أي شيء يتم فعله مع **n1** و **n2**\* سيغير القيم الفعلية لـ **a** و **b**. لذلك، عندما نطبع قيم **a** و **b** في الدالة الرئيسية، تكون القيم قد تغيرت.

### شرح مفصل:

عند تمرير متغير دالة في C++, يتم نسخ قيمة المتغير إلى متغير محلي داخل الدالة. أي تغيير يتم على المتغير المحلي داخل الدالة لا يؤثر على المتغير الأصلي خارج الدالة.

تمرير المتغيرات بالمرجع باستخدام المؤشرات هو طريقة لتمرير **عنوان المتغير الأصلي** إلى الدالة. تسمح هذه الطريقة للدالة بتعديل **المتغير الأصلي** مباشرةً.

يتم الإشارة إلى **عنوان المتغير** باستخدام الرمز (&) أمام اسم المتغير. على سبيل المثال، **&a** تشير إلى عنوان المتغير **a**. داخل الدالة، يمكن الوصول إلى **القيمة المخزنة في العنوان** الذي تم تمريره باستخدام الرمز (\*).

على سبيل المثال: **n1**\* يعطي القيمة المخزنة على العنوان **n1**. أي تغيير يتم على القيمة التي يتم الوصول إليها باستخدام الرمز (\*) سيغير **القيمة المخزنة في العنوان الأصلي** للمتغير خارج الدالة.

في هذا المثال، تمرير **&a** و **&b** إلى الدالة **Swap** يسمح للدالة بتغيير القيم الفعلية لـ **a** و **b** من خلال تعديل القيم في العنوانين المشار إليهما.

### ملخص:

تمرير المتغيرات بالمرجع باستخدام المؤشرات هو أداة قوية في C++ وطريقة قوية تسمح للدوال بتعديل المتغيرات الأصلية مباشرةً. لكنها تتطلب فهماً جيداً للذاكرة والمؤشرات ذلك استخدامها بحكمة لتحسين أداء وموثونة برامحك، ومع ذلك، فإن استخدام المؤشرات بشكل غير صحيح يمكن أن يؤدي إلى أخطاء خطيرة، لذا يجب توخي الحذر عند استخدامها.

## نصائح إضافية:

- يعتبر استخدام المراجع (references) بدلاً من المؤشرات عادةً أسلوباً أسهل وأكثر أماناً لتمرير المتغيرات بالمرجع، لأنها تتعامل مع إلغاء التأشير تلقائياً.
- إذا كنت بحاجة إلى إمكانية أكبر في التحكم بعناوين الذاكرة، فإن المؤشرات تظل أداة قوية ويمكن استخدامها بشكل فعال مع توخي الحذر.

## فوائد تمرير المتغيرات بالمرجع:

- تعديل القيم الأصلية للمتغيرات داخل الدالة.
- مفید لتبادل قيم المتغيرات أو إجراء عمليات أخرى تؤثر على المتغيرات الأصلية.

## سلبيات تمرير المتغيرات بالمرجع:

- زيادة صعوبة فهم الكود مقارنة بالاستدعاء بالقيمة.
- يتطلب توخي الحذر لتجنب أخطاء الوصول إلى البيانات غير الصالحة.

## أهمية فهم هذا المفهوم:

فهم تمرير المتغيرات بالمرجع باستخدام المؤشرات أمر ضروري لكتابه برامح C++ فعالة واستخدام الذاكرة بشكل مناسب. يمنحك تحكمًا أكبر في البيانات ويسمح لك بإنشاء هيكل بيانات معقدة.

```
#include <iostream>

using namespace std;

void SwapWithReference(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void SwapWithPPointer(int* a, int* b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int x = 10, y = 20;

    cout << "Call By Reference: " << endl;
    cout << "-----\n";

    cout << "Befor Swap: " << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl << endl;

    SwapWithReference(x, y);

    cout << "After Swap: " << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl << endl;
```

```

cout << "\n=====\n";
int m = 30, n = 40;

cout << "Call By Pointer: " << endl;
cout << "-----\n";

cout << "Befor Swap: " << endl;
cout << "m = " << m << endl;
cout << "n = " << n << endl << endl;

SwapWithPPointer(&m, &n);

cout << "After Swap: " << endl;
cout << "m = " << m << endl;
cout << "n = " << n << endl << endl;

cout << "\n=====\n";

// البيانات متغير ثالث الاكس ، والواي
cout << "Call By Pointer: " << endl;
cout << "-----\n";
cout << "=====Pointer is point to x and y===== " << endl;

SwapWithPPointer(&x, &y);

cout << "After Swap: " << endl;
cout << "x = " << x << endl;
cout << "y = " << y << endl << endl;

system("pause");

return 0;
}

```

```

Call By Reference:
-----
Befor Swap:
x = 10
y = 20

After Swap:
x = 20
y = 10

=====
Call By Pointer:
-----
Befor Swap:
m = 30
n = 40

After Swap:
m = 40
n = 30

=====
Call By Pointer:
-----
=====Pointer is point to x and y=====
After Swap:
x = 10
x = 20

```



# Lesson #40 - Pointers and Arrays

## C++ - المؤشرات والمصفوفات في Pointers and Arrays !

في لغة البرمجة C++ ، المؤشرات هي متغيرات تخزن عناوين متغيرات أخرى. لا تقتصر مهمة المؤشرات على تخزين عنوان متغير واحد، بل يمكنها أيضًا تخزين عناوين عناصر مصفوفة (array) كاملة.

### إذن، كيف تعمل المؤشرات مع المصفوفات؟ !

تخزين عناوين خانات المصفوفة: يمكن للمؤشر أن يشير إلى أي خانة من خانات المصفوفة. على سبيل المثال، إذا كان لديك مصفوفة [5] int arr ، فإن كل خانة لها عنوان فريد. يمكن للمؤشر تخزين عنوان أي من هذه الخمس خانات.

الوصول إلى قيم خانات المصفوفة: باستخدام الرمز (\*) ، يمكنك الوصول إلى القيمة المخزنة في الخانة التي يشير إليها المؤشر. على سبيل المثال، إذا كان لديك مؤشر int\* ptr = &arr[2] ، فإن ptr \* ستعطي القيمة المخزنة في الخانة الثالثة من المصفوفة arr.

التحريك عبر خانات المصفوفة: يمكن استخدام العمليات الحسابية الأساسية على المؤشرات لتحريكها عبر خانات المصفوفة. على سبيل المثال، ptr++ سيزيد قيمة المؤشر بمقدار حجم نوع البيانات في المصفوفة (4 بait ل int) ، مما يجعله يشير إلى الخانة التالية.

تغير قيم العناصر: إذا كان المؤشر يشير إلى عنصر معين في المصفوفة، يمكنك تعديل قيمته مباشرةً. هذا يختلف عن تمرير المصفوفة بالكامل بالمرجع، حيث يتعامل مع نسخة مؤقتة من المصفوفة داخل الدالة.

دعم هياكل البيانات المعقدة: العديد من هياكل البيانات المتقدمة، مثل القوائم المرتبطة (Linked Lists) والأشجار (Trees)، تعتمد على استخدام المؤشرات لربط العناصر بعضها البعض.

### فائدة استخدام المؤشرات مع المصفوفات:

مرنة الوصول للعناصر: يمكنك تحريك المؤشر لتنبيع عناصر المصفوفة وتعديل قيمتها بسهولة.

كفاءة الذاكرة: مقارنة بتمرير المصفوفة بأكملها للدالة، يمكنك تمرير مؤشر يشير إلى العنصر المطلوب، مما يوفر مساحة في الذاكرة.

تعقيدات متقدمة: يمكن استخدام المؤشرات لإنشاء هياكل بيانات معقدة مثل القوائم المرتبطة.

إمكانية الوصول المباشر إلى أي خانة في المصفوفة: على عكس التكرارات، التي تتطلب معرفة مسبقة لمؤشر الخانة المطلوبة، يمكن للمؤشرات الوصول إلى أي خانة ديناميكياً.

مرنة أكبر في معالجة المصفوفات: يمكن استخدام المؤشرات لتنفيذ عمليات معقدة على المصفوفات، مثل البحث عن عنصر معين، أو فرز البيانات، أو عكس ترتيب العناصر.

إعلان المؤشر: إعلان متغير من نوع المؤشر (مثل int\* ptr) ليحتفظ بعنوان العنصر الأول في المصفوفة.

تعيين العنوان: استخدم اسم المصفوفة (مثل arr) مع الرمز (&) للحصول على عنوان العنصر الأول وتعيينه للمؤشر (مثل : ptr = &arr[0]).

الوصول إلى العناصر: للوصول إلى عنصر معين، استخدم المؤشر متبعاً بمؤشر الترقيم (index) بين قوسين مربعين (مثل : ptr[i]).

مثال:

```
int arr[] = {1,2,3,4,5};  
int* ptr = &arr[0];  
for (int i = 0; i < 5; i++) {  
    cout << ptr[i] << endl;  
}  
ptr[1]++;  
cout << ptr[1] << endl;
```

طباعة عناصر المتتالية باستخدام المؤشر

زيادة قيمة العنصر الثاني باستخدام المؤشر

طباعة قيمة العنصر الثاني بعد التعديل

في هذا المثال، نستخدم مؤشراً ptr للوصول إلى عناصر مصفوفة arr. نقوم بطباعة العناصر وتعديل قيمة العنصر الثاني، مما يوضح كيفية استخدام المؤشرات للتلاعب بالممتاليات.

## ؟ سلبيات استخدام المؤشرات:

**صعوبة الاستخدام:** التعامل مع المؤشرات يتطلب فهماً أكبر لمفاهيم الذاكرة والمخاطر المحتملة مثل الوصول إلى عناوين غير صالحة.

**أخطاء محتملة:** الاستخدام غير الصحيح للمؤشرات يمكن أن يؤدي إلى أخطاء في البرنامج يصعب اكتشافها.

## ؟ خاتمة:

المؤشرات والمصفوفة أدوات قوية في **C++** تسمح بالتحكم الدقيق في البيانات وتنفيذ خوارزميات متقدمة. لكن تذكر أن استخدامها يتطلب توخي الحذر والانتباه إلى ممارسات البرمجة الآمنة لتجنب الأخطاء والحفاظ على استقرار البرامج.

```
int *ptr;  
int arr[5];  
ptr = &arr[0];
```

Suppose we need to point to the **fourth element** of the **array** using the same pointer **ptr**.

Here, if **ptr** points to the **first element** in the above example then **ptr + 3** will point to the **fourth element**. For example.

لنفترض أننا بحاجة إلى الإشارة إلى **العنصر الرابع في المصفوفة** باستخدام نفس المؤشر **ptr**. هنا، إذا كان **ptr** يشير إلى **العنصر الأول** في المثال أعلاه، فإن **ptr + 3** سيشير إلى **العنصر الرابع**. على سبيل المثال:

```
int *ptr;  
int arr[5];  
ptr = arr;
```

**ptr + 1** is equivalent to **&arr[1]**;  
**ptr + 2** is equivalent to **&arr[2]**;  
**ptr + 3** is equivalent to **&arr[3]**;  
**ptr + 4** is equivalent to **&arr[4]**;

Similarly, we can access the elements using **the single pointer**. For example,

وبالمثل، يمكننا الوصول إلى العناصر باستخدام **مؤشر واحد**. على سبيل المثال:

```
// use dereference operator ( * )  
  
*ptr == arr[0];  
*(ptr + 1) is equivalent to arr[1];  
*(ptr + 2) is equivalent to arr[2];  
*(ptr + 3) is equivalent to arr[3];  
*(ptr + 4) is equivalent to arr[4];
```

Suppose if we have initialized **ptr = &arr[2]**; then

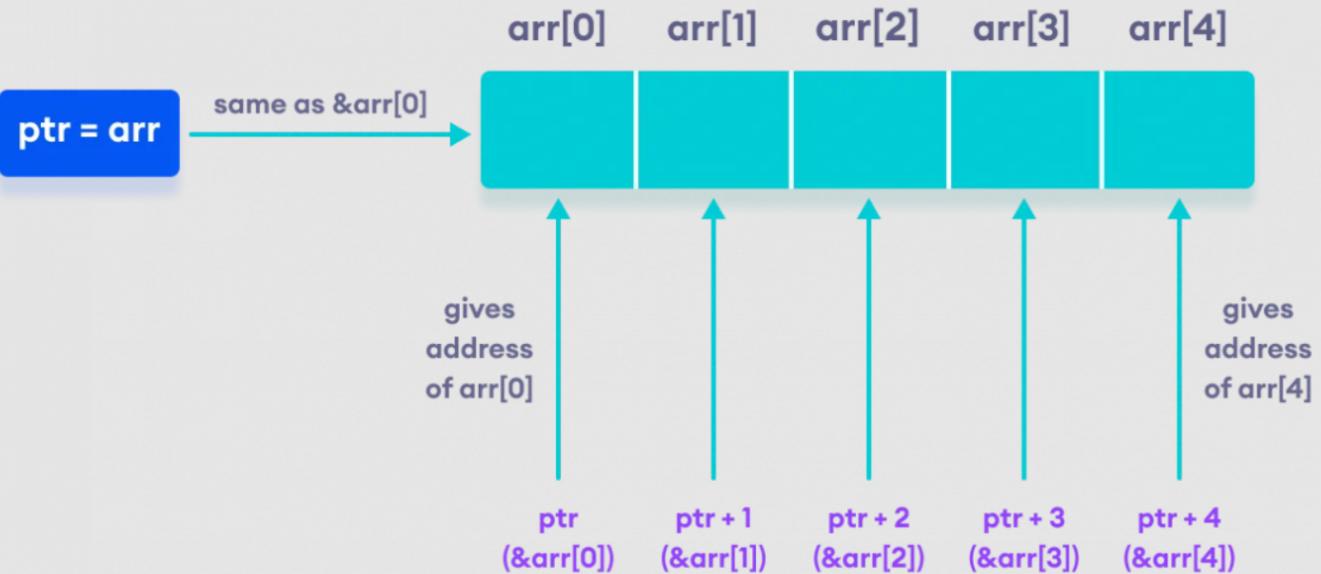
**ptr - 2** is equivalent to **&arr[0]**;  
**ptr - 1** is equivalent to **&arr[1]**;  
**ptr + 1** is equivalent to **&arr[3]**;  
**ptr + 2** is equivalent to **&arr[4]**;

**Note:** The address between **ptr** and **ptr + 1** differs by **4 bytes**. It is because **ptr** is a pointer to an **int** data. And, the size of **int** is **4 bytes** in a 64-bit operating system.

**ملاحظة:** يختلف العنوان بين **ptr** و **ptr + 1** بمقدار **4 بآيت**. ذلك لأن **ptr** هو مؤشر لبيانات **int**. ويبلغ حجم **int** **4 بآيت** في نظام التشغيل **64 بت**.

Similarly, if pointer **ptr** is pointing to **char** type data, then the address between **ptr** and **ptr + 1** is **1 byte**. It is because the size of a **character** is **1 byte**.

وبالمثل، إذا كان المؤشر **ptr** يشير إلى بيانات من نوع **char**، فإن العنوان بين **ptr** و **ptr + 1** هو **1 بآيت**. وذلك لأن حجم **الحرف** هو **1 بآيت**.



```

#include <iostream>
using namespace std;

int main()
{
    cout << endl;

    int arr[5] = { 1,2,3,4,5 };
    // declare pointer variable
    int* ptr;
    cout << "Displaying address using arrays: " << endl;
    // use for loop to print addresses of all array elements
    for (int i = 0; i < 5; i++)
    {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }
    cout << "\nDisplaying Values using arrays: " << endl;
    // use for loop to print all array elements
    for (int i = 0; i < 5; i++)
    {
        cout << "arr[" << i << "] = " << arr[i] << endl;
    }
    // ptr = &arr[0]
    ptr = arr;
    cout << "\nDisplaying address using pointers: " << endl;
    // use for loop to print addresses of all array elements
    // using pointer notation
    for (int i = 0; i < 5; i++)
    {
        cout << "ptr + " << i << " = " << ptr + i << endl;
    }
    cout << "\nDisplaying Values using pointers: " << endl;
    // use for loop to print all array elements
    // using pointer dereference operator (*).
    for (int i = 0; i < 5; i++)
    {
        cout << "ptr + " << i << " = " << *ptr + i << endl;
    }
    cout << endl;
    system("pause");
}

```

```
Displaying address using arrays:
```

```
&arr[0] = 00000096CDFCFB68  
&arr[1] = 00000096CDFCFB6C  
&arr[2] = 00000096CDFCFB70  
&arr[3] = 00000096CDFCFB74  
&arr[4] = 00000096CDFCFB78
```

```
Displaying Values using arrays:
```

```
arr[0] = 1  
arr[1] = 2  
arr[2] = 3  
arr[3] = 4  
arr[4] = 5
```

```
Displaying address using pointers:
```

```
ptr + 0 = 00000096CDFCFB68  
ptr + 1 = 00000096CDFCFB6C  
ptr + 2 = 00000096CDFCFB70  
ptr + 3 = 00000096CDFCFB74  
ptr + 4 = 00000096CDFCFB78
```

```
Displaying Values using pointers:
```

```
ptr + 0 = 1  
ptr + 1 = 2  
ptr + 2 = 3  
ptr + 3 = 4  
ptr + 4 = 5
```

```
#include <iostream>
using namespace std;

int main() {
    cout << endl;

    float arr[5];

    // Insert data using pointer notation
    cout << "Enter 5 numbers: " << endl;
    for (int i = 0; i < 5; ++i)
    {
        // store input number in arr[i]
        cin >> *(arr + i);
    }

    // Display data using pointer notation
    cout << "\nDisplaying data: " << endl;
    for (int i = 0; i < 5; ++i)
    {
        // display value of arr[i]
        cout << *(arr + i) << endl;
    }

    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Enter 5 numbers:
41
23
65
98
74

Displaying data:
41
23
65
98
74
```



# Lesson #41 - Pointers and Structure

## C++ - مؤشرات الهياكل في Structures Pointers

في لغة البرمجة C++, لا تقتصر إمكانية إنشاء المؤشرات على الأنواع الأساسية مثل **الأعداد الصحيحة (int)** و **الأعداد العشرية (float)**، بل يمكن أيضًا إنشاؤها لأنواع البيانات التي يُعرفها المستخدم، بما في ذلك **الهياكل (Structures)**.

يمكننا استخدام مؤشرات (Pointers) للوصول إلى هياكل البيانات. يُسمى المؤشر الذي يشير إلى عنوان كتلة الذاكرة التي تخزن هيكلًا باسم **مؤشر الهيكل (Structure Pointer)**. (Structure Pointer).

**مؤشر الهيكل (Structure Pointer)** : هو متغير يخزن **عنوان** كتلة الذاكرة التي تحتوي على **Structure**. بمعنى آخر، إنه يشير إلى **Structure** معين في الذاكرة.

مثال:

```
struct Person {
    int age;
    char name[20];
};
```

تعريف **Structure**

Person\* ptr;

تعريف مؤشر (Pointer) إلى هيكل (Structure) إلى هيكل (Structure) من نوع Person يشير إلى هيكل (Structure)

```
Person p1 = {25, "John"};
ptr = &p1;
```

إنشاء هيكل (Structure) وتخزينه في الذاكرة

جعل المؤشر (Pointer) ptr يشير إلى هيكل (Structure) p1

الوصول إلى عناصر الهيكل (Structure) باستخدام المؤشر (Pointer) باستخدامة (Structure) سببيط 25

```
cout << ptr -> age << endl;
cout << ptr -> name << endl;
```

سببيط 25

"John"

ملاحظات:

- تُستخدم علامة النجمة (\*) لإلغاء تأثير المؤشر (Pointer) والوصول إلى عناصر الهيكل (Structure).
- يمكن أيضًا تعريف قيمة مباشرة للمؤشر (Pointer) باستخدام عنوان هيكل (Structure) آخر، مثل: ptr = &p2؛ حيث p2 هيكل (Structure) آخر من نفس النوع.
- يمكن استخدام المؤشرات (Pointers) إلى الهياكل (Structures) لبناء هياكل (Pointers) بيانات معقدة، مثل القوائم المرتبطة.

### الفرق بين المتغيرات الهيكلية ومؤشرات الهياكل:

- المتغيرات الهيكلية هي نسخ مستقلة من الهيكل (Structure) في الذاكرة، بينما المؤشرات (Pointers) تشير إلى نفس كتلة الذاكرة التي يحتويها هيكل (Structure) واحد.
- تعديل عناصر الهيكل (Structure) باستخدام متغير هيكل (Structure) يؤثر فقط على ذلك المتغير، بينما تعديل عناصر الهيكل (Structure) باستخدام مؤشر (Pointer) سيؤثر على الهيكل (Structure) الذي يشير إليه.

### ميزايات استخدام مؤشرات الهياكل:

- **ديناميكية أو مرونة الوصول** : يمكن تغيير الهيكل (Structure) الذي يشير إليه المؤشر (Pointer) وقت التشغيل - **معنى آخر** - يتيح لك المؤشر (Pointer) الوصول إلى أي هيكل (Structure) في الذاكرة وتحويله بسهولة بين الهياكل المختلفة (Different Structures) - **معنى ثالث** : يمكنك بسهولة تحريك المؤشر للوصول إلى هياكل بيانات مختلفة أو تعديل محتواها، وهو ما يوفر مرونة أكبر مقارنة بتمرير الهيكل نفسه للدواو.
- **كفاءة الذاكرة**: في بعض الحالات، يمكن أن يكون تمرير مؤشر (Pointer) إلى هيكل (Structure) في الدوال (Functions) بدلاً من تمرير الهيكل (Structure) بالكامل أكثر كفاءة في استخدام الذاكرة، خاصةً إذا كان الهيكل كبير الحجم. - **معنى آخر** : يمكن للمؤشر (Pointer) أن يشير إلى هيكل (Structure) موجود بالفعل، مما يوفر مساحة في الذاكرة.
- **إنشاء هيكل بيانات معقدة**: يمكن استخدام مؤشرات الهياكل لإنشاء هيكل بيانات معقدة مثل القوائم المرتبطة والأشجار.

## سلبيات استخدام مؤشرات الهياكل:

- صعوبة الاستخدام: التعامل مع المؤشرات (Pointers) يتطلب فهماً أكبر لمفاهيم الذاكرة والمخاطر المحتملة مثل الوصول إلى عناوين غير صالحة.
- أخطاء محتملة: الاستخدام غير الصحيح للمؤشرات (Pointers) يمكن أن يؤدي إلى أخطاء في البرنامج يصعب اكتشافها.
- قد يكون استخدام الهياكل نفسها أفضل في بعض الحالات، اعتماداً على التطبيق وظروف الاستخدام.

### Example #1

```
#include <iostream>

using namespace std;

struct Person {
    int age;
    char name[20];
};

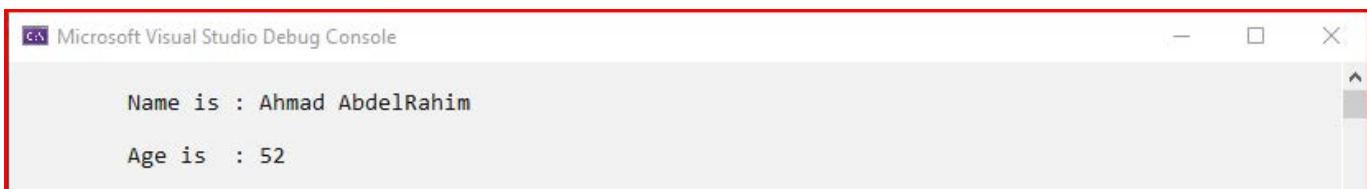
int main()
{
    system("color f0");
    Person* ptr;

    Person P1 = { 52, "Ahmad AbdelRahim" };

    ptr = &P1;

    cout << "\n\t Name is : " << ptr->name << endl;
    cout << "\n\t Age is : " << ptr->age << endl;

    return 0;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console displays two lines of text:  
Name is : Ahmad AbdelRahim  
Age is : 52

## Example #2

```
#include <iostream>
using namespace std;

struct Distance {
    int Minute;
    float Second;
};

int main()
{
    Distance* ptr, T;

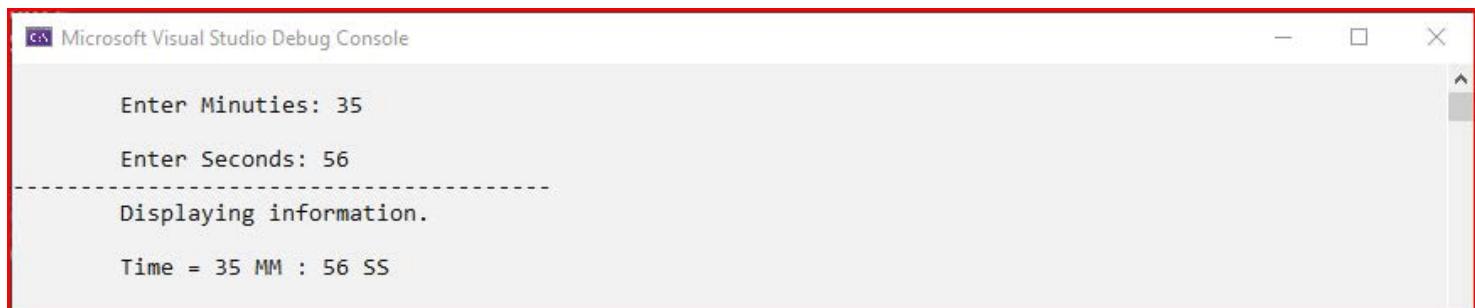
    ptr = &T;

    cout << "\n\tEnter Minuties: ";
    cin >> ptr->Minute;           // = (*ptr).Minute
    cout << "\n\tEnter Seconds: ";
    cin >> ptr->Second;          // = (*ptr).Second

    cout << "-----" << endl;
    cout << "\tDisplaying information." << endl;

    // (*ptr).feet      (*ptr).inch
    cout << "\n\tTime = " << ptr->Minute << " MM : "
        << ptr->Second << " SS\n" << endl;

    return 0;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. It displays the following text output:

```
Microsoft Visual Studio Debug Console

Enter Minuties: 35
Enter Seconds: 56
-----
Displaying information.

Time = 35 MM : 56 SS
```



# Lesson #42 - Pointer to Void

## Pointer to Void (المؤشر الفارغ)

في لغة C++, يُطلق على المؤشر الفارغ اسم (void pointer) وهو نوعاً خاصاً من المؤشرات يتم الإعلان عنه باستخدام الكلمة الأساسية (void) متبوعة بعلامة النجمة (\*). يختلف عن المؤشرات العادية في أنه لا يرتبط بنوع بيانات محدد، بل يمكنه الإشارة إلى بيانات من أي نوع، لذلك يُسمى أيضاً "المؤشر العام" (Generic Pointer).

تركيب المؤشر الفارغ في C++:

```
void* Ptr_name;
```

شرح التركيب:

الكلمة الأساسية void\* تشير إلى غياب نوع البيانات المحدد، متبوعة بعلامة النجمة (\*) والتي تشير إلى كونه مؤشر.

اسم المتغير الذي سيخزن عنوان الذاكرة الذي يشير إليه المؤشر الفارغ.

إليك شرحاً لأهم خصائص المؤشرات الفارغة ومميزاتها واستخداماتها:

### 1. عدم تحديد نوع البيانات:

لا يشير المؤشر الفارغ إلى نوع بيانات محدد عند تعريفه، سواء كان عدداً صحيحاً، أو حرفًا، أو بنية بيانات، أو حتى مؤشراً آخر. بل يمكنه الإشارة إلى بيانات من أي نوع، وعند إعلانه نستخدم (\*void). خلافاً للمؤشرات العادية التي ترتبط بنوع محدد من البيانات (مثلاً int\* أو char\*)، هذا يعني أنه يمكنه تخزين عنوان أي متغير، بغض النظر عن نوعه، مما يجعله مرنّاً للغاية.

### 2. لا يمكن طباعة القيمة المخزنة مباشرةً:

لا يمكن الوصول إلى القيمة المخزنة في العنوان الذي يشير إليه المؤشر الفارغ مباشرةً باستخدام المؤشر نفسه. يجب أولاً تحويله إلى نوع مؤشر محدد (مثلاً int\*, char\*, الخ) قبل الوصول إلى القيمة. يرجع ذلك إلى أن المترجم يحتاج إلى معرفة حجم نوع البيانات لحساب العنوان الصحيح للقيمة.

### استخدامات المؤشرات الفارغة:

- تمرير البيانات من نوع غير معروف للدواال: يمكن استخدام المؤشرات الفارغة لتمرير البيانات من نوع غير معروف إلى الدوال، مما يوفر مرونة في التعامل مع أنواع مختلفة من البيانات.
- برمجة وظائف عامة يمكنها العمل مع أنواع مختلفة من البيانات.
- التعامل مع تخصيص الذاكرة بشكل عشوائي (dynamic memory allocation) عندما لا يكون نوع البيانات معروفاً مسبقاً. حيث يتم استخدام المؤشرات الفارغة بشكل شائع في دوال مثل malloc و free لتخصيص الذاكرة وتحريرها، حيث لا تهتم هذه الدوال بنوع البيانات التي سيتم تخزينها في الذاكرة.
- ربط دوال مكتوبة بلغات مختلفة: يمكن استخدام المؤشرات الفارغة لربط دوال مكتوبة بلغات برمجة مختلفة، مثل C و C++, حيث قد تختلف أنظمة أنواع البيانات بين اللغات.

مثال على استخدام المؤشر الفارغ:

```
void* Ptr;  
int num = 10;  
Ptr = &num;
```

إعلان مؤشر فارغ

جعل المؤشر يشير إلى متغير من نوع int

قبل استخدام القيمة، يجب تحويل المؤشر إلى نوع int وهناك ثلاث طرق للتحويل:

الطريقة الأولى (C-style casting):

```
int* int_Ptr = (int*)Ptr;  
int Value = *int_Ptr;
```

يمكن استخدام القيمة بشكل آمن بهذه الطريقة

الطريقة الثانية (static\_cast operator) وهي الأفضل:

```
int* int_Ptr = static_cast<char*>(Ptr);  
int Value = *int_Ptr;
```

يمكن استخدام القيمة بشكل آمن بهذه الطريقة

الطريقة الثالثة (static\_cast operator):

```
cout << *(static_cast<char*>(Ptr));
```

الأمور الواجب مراعاتها عند استخدام المؤشرات الفارغة:

- التعامل بحذر: رغم مرونته، ينبغي التعامل مع المؤشرات الفارغة بحذر لأنها قد تؤدي إلى أخطاء إذا لم يتم التعامل معها بشكل صحيح بسبب عدم تحديد نوع البيانات التي تشير إليها.
- تحويل النوع قبل الاستخدام: قبل استخدام البيانات التي يشير إليها المؤشر الفارغ، يجب تحويله إلى نوع البيانات الصحيح باستخدام عملية تسمى "القولبة" (casting) لتجنب الأخطاء.
- تجنب استخدامها بشكل عام: يُفضل استخدام أنواع المؤشرات المحددة كلما أمكن ذلك لتحسين وضوح الكود وقابليته للقراءة.

```
#include <iostream>

using namespace std;

int main()
{
    void* Ptr;

    int num = 10;

    Ptr = &num;

    cout << endl;

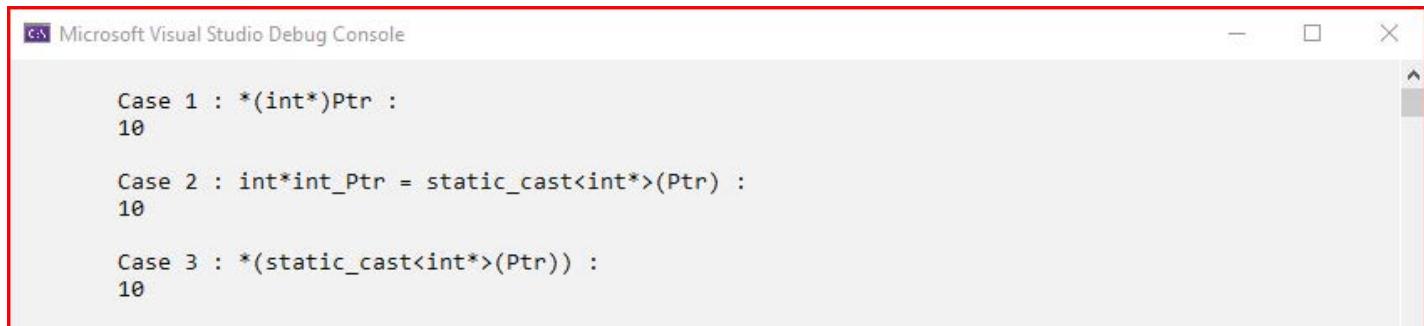
    cout << "\tCase 1 : *(int*)Ptr : " << endl;
    cout << "\t" << *(int*)Ptr << endl;

    int* int_Ptr = static_cast<int*>(Ptr);

    cout << "\n\tCase 2 : int*int_Ptr = static_cast<int*>(Ptr) : " << endl;
    cout << "\t" << *int_Ptr << endl;

    cout << "\n\tCase 3 : *(static_cast<int*>(Ptr)) : " << endl;
    cout << "\t" << *(static_cast<int*>(Ptr)) << endl;

    return 0;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. It displays three cases of pointer casting:

- Case 1: Direct cast of void\* to int\*. The output is 10.
- Case 2: Dynamic cast of void\* to int\*. The output is 10.
- Case 3: Dynamic cast of void\* through an intermediate variable. The output is 10.

```
#include <iostream>

using namespace std;

int main()
{
    void* Ptr;

    int num = 40;

    float fnum = 45.89;

    Ptr = &num;

    cout << "\n\tPrint address to variable num : " << Ptr << endl;

    cout << "\n\tPrint Value to variable num: \n" << *(static_cast<int*>(Ptr));

    cout << "\n\t-----\n";

    Ptr = &fnum;

    cout << "\n\tPrint address to variable fnum: " << Ptr << endl;

    cout << "\n\tPrint Value to variable fnum: \n" << *(static_cast<float*>(Ptr));

    return 0;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console displays the following text:

```
Print address to variable num : 0000001E4031FB74
Print Value to variable num: 40
-----
Print address to variable fnum: 0000001E4031FB94
Print Value to variable fnum: 45.89
```



# Lesson #43 - Memory Management:

## new and delete

### (Memory Management) إدارة الذاكرة

#### ما هي إدارة الذاكرة؟

هي عملية تنظيم واستخدام ذاكرة الوصول العشوائي (RAM) بشكل فعال من قبل نظام التشغيل والبرامج. تهدف إلى ضمان توفير ذاكرة كافية لتشغيل جميع البرامج والنظام بشكل سلس، مع تجنب مشاكل مثل تسربات الذاكرة والأداء البطيء بمراعبة استخدام الذاكرة وإعادة تخصيصها حسب الحاجة، وأيضاً تحرير الذاكرة بإتاحة مساحة في ذاكرة الوصول العشوائي (RAM) عندما لم تعد البرامج أو البيانات بحاجة إليها.

#### مهام إدارة الذاكرة:

- **تخصيص الذاكرة:** تخصيص ذاكرة كافية للبرامج قيد التشغيل.
- **تنظيم الذاكرة:** تقسيم الذاكرة إلى أقسام مختلفة تُستخدم لأغراض مختلفة.
- **إعادة استخدام الذاكرة:** تحرير الذاكرة التي لم تعد قيد الاستخدام وإتاحتها للبرامج الأخرى.
- **حماية الذاكرة:** منع البرامج من الوصول إلى ذاكرة البرامج الأخرى أو ذاكرة النظام.

#### أهمية إدارة الذاكرة:

- **ضمان التشغيل السلس:** تضمن إدارة الذاكرة توفر مساحة كافية لتشغيل جميع البرامج والبيانات في نفس الوقت.
- **تحسين الأداء:** تساعد إدارة الذاكرة في تقليل وقت الوصول إلى البيانات، مما يحسن من أداء النظام.
- **منع تسربات الذاكرة:** تساعد إدارة الذاكرة في منع تسربات الذاكرة، وهي مشكلة يمكن أن تؤدي إلى تعطل النظام.

#### مكونات إدارة الذاكرة:

- **وحدة إدارة الذاكرة (MMU):** هي مكون في وحدة المعالجة المركزية (CPU) مسؤول عن ترجمة العناوين الافتراضية للذاكرة إلى عناوين مادية.
- **نظام التشغيل:** يلعب دوراً رئيسياً في إدارة الذاكرة من خلال تخصيص الذاكرة للبرامج وتحريرها، وتطبيق تقنيات مثل التجزئة والتخزين المؤقت.
- **البرامج:** تتحمل البرامج مسؤولية استخدام الذاكرة بكفاءة، وتجنب التسربات، وإعادة استخدام الذاكرة المحرمة.

## تقنيات إدارة الذاكرة:

- التخصيص الثابت: تخصيص مساحة ثابتة من الذاكرة للبرامج والبيانات قبل تشغيلها.
- التخصيص الديناميكي: تخصيص مساحة من الذاكرة للبرامج والبيانات أثناء تشغيلها حسب الحاجة.
- التجزئة أو التقسيم (**Partitioning**): تقسيم الذاكرة إلى أجزاء أصغر لتلبية احتياجات البرامج المختلفة..
- الاستبدال (**Swapping**): نقل بعض البيانات من ذاكرة الوصول العشوائي (**RAM**) إلى القرص الثابت عندما تكون الذاكرة ممتلئة ، أو إخراج البرنامج غير المستخدمة من الذاكرة وإعادة استخدام مساحتها.
- الجمع (**Compaction**): دمج المناطق الفارغة في ذاكرة الوصول العشوائي (**RAM**) لتحسين استخدامها.
- جمع القمامة (**Garbage Collection**): تحدد وتحرر الذاكرة التي لم تعد قيد الاستخدام تلقائياً من قبل نظام التشغيل.
- التخزين المؤقت: تخزين البيانات التي يتم استخدامها بشكل متكرر في ذاكرة سريعة للوصول إليها بشكل أسرع.

## أدوات إدارة الذاكرة:

- مدير المهام: أداة تساعد المستخدمين على مراقبة استخدام الذاكرة وإغلاق البرامج التي تستهلك الكثير من الذاكرة.
- محلل الذاكرة: أداة تساعد المطورين [على تحديد مشاكل إدارة الذاكرة في البرنامج.

## ميزات إدارة الذاكرة الفعالة:

- تحسين الأداء: ضمان توفر ذاكرة كافية للبرامج قيد التشغيل، مما يحسن من أداء النظام.
- استقرار النظام: منع حدوث أخطاء وتحميد النظام بسبب نقص الذاكرة.
- الأمان: منع البرامج الضارة من الوصول إلى ذاكرة البرنامج الأخرى أو ذاكرة النظام.

## عيوب إدارة الذاكرة:

- تعقيد النظام: تزيد تقنيات إدارة الذاكرة من تعقيد نظام التشغيل والبرامج.
- التأثير على الأداء: قد تؤثر بعض تقنيات إدارة الذاكرة على أداء النظام، مثل عملية الاستبدال.

## مخاطر سوء إدارة الذاكرة:

- انهيار النظام: قد يؤدي نقص الذاكرة إلى انهيار النظام.
- التجزئة: يمكن أن تصبح ذاكرة الوصول العشوائي (**RAM**) مجزأة، مما يعني أن هناك العديد من المساحات الصغيرة غير متجاورة وغير المستخدمة من الذاكرة ، مما قد يمنع تخصيص ذاكرة كافية للبرامج الكبيرة.
- تسربات الذاكرة: قد تؤدي بعض الأخطاء في إدارة الذاكرة إلى تسربات الذاكرة عندما لا يتم تحرير الذاكرة التي لم تُعد قيد الاستخدام، مما يؤدي إلى استنفاد الذاكرة بمرور الوقت.
- ثغرات الأمان: قد تُعرض بعض تقنيات إدارة الذاكرة النظام لثغرات الأمان باستخدام هذه التقنيات في الهجمات الأمنية.
- الأداء البطيء: الوصول إلى البيانات المخزنة في ذاكرة بطئ، مثل ذاكرة القرص الصلب، يمكن أن يؤدي إلى بطء الأداء.

## نصائح لتحسين إدارة الذاكرة عامة:

- إغلاق البرامج غير المستخدمة: تحرير ذاكرة البرامج المستخدمة من قبل البرامج غير النشطة.
- ترقية ذاكرة الوصول العشوائي (**RAM**): زيادة كمية ذاكرة الوصول العشوائي المتوفرة لتشغيل المزيد من البرامج بشكل سلس.
- استخدام أدوات تحليل الذاكرة: تتبع استخدام البرامج للذاكرة وتحديد التسربات.
- كتابة برمجيات بكفاءة: استخدام تقنيات البرمجة الجيدة لضمان استخدام الذاكرة بكفاءة.

## نصائح لإدارة الذاكرة في C++:

- تجنب تسربات الذاكرة: التأكد من تحرير جميع الذاكرة المخصصة ديناميكياً عند عدم الحاجة إليها.
- استخدام مكتبات إدارة الذاكرة: استخدام مكتبات إدارة الذاكرة لتسهيل إدارة الذاكرة مثل **std::vector** و **Boost.SmartPointers** و **std::string**.
- مراقبة استخدام الذاكرة: استخدام أدوات مراقبة استخدام الذاكرة لتحديد أي تسربات أو مشاكل أخرى في إدارة الذاكرة.
- استخدام أدوات تصحيح الأخطاء: استخدام أدوات تصحيح الأخطاء للكشف عن تسربات الذاكرة.

## أمثلة على تقنيات إدارة الذاكرة من لغة C++:

- استخدام `new` و `delete`: لتخفيض وتحرير الذاكرة بشكل صريح من قبل المبرمج.
- `new`: تخصيص ذاكرة ديناميكية. - `delete`: تحرير ذاكرة مُخصصة ديناميكياً.
- استخدام `std::vector`: لتخزين مجموعة من العناصر ذات الحجم الديناميكي من نفس النوع.
- يُدير `std::vector` الذاكرة تلقائياً.
- استخدام `std::string`: لتخزين سلسلة نصية ذات الحجم الديناميكي.
- يُدير `std::string` الذاكرة تلقائياً.
- مُخصصات الذاكرة الذكية: مثل `std::shared_ptr` و `std::unique_ptr`، تساعد في إدارة الذاكرة بشكل تلقائي.
- وتجنب تسربات الذاكرة.
- جمع القمامنة: بعض مكتبات C++ توفر تقنيات جمع القمامنة لإدارة الذاكرة بشكل تلقائي.

## أمثلة توضيحية على تقنيات إدارة الذاكرة من لغة C++:

```
int* p = new int; // تخصيص ذاكرة لمتغير من نوع int
*p = 10; // تخزين قيمة في المتغير
delete p; // تحرير الذاكرة المخصوصة للمتغير
std::unique_ptr<int> p2(new int); // استخدام مخصوص الذاكرة الذكية std::unique_ptr
*p2 = 20; // تخزين قيمة في المتغير
// لا حاجة لتحرير الذاكرة، يتم تحريرها تلقائياً عند انتهاء صلاحية p2
```

## أمثلة على تقنيات إدارة الذاكرة في أنظمة التشغيل:

- Windows: يستخدم نظام Windows تقنية "الجمع القمامنة" لتحرير الذاكرة التي لم تعد قيد الاستخدام تلقائياً.
- Linux: يستخدم نظام Linux تقنية "الاستبدال" لنقل بعض البيانات من ذاكرة الوصول العشوائي (RAM) إلى القرص الثابت عندما تكون الذاكرة ممتلئة.

## خلاصة:

إدارة الذاكرة هي عملية مهمة في البرمجة وأيضاً هي عملية أساسية في أنظمة التشغيل والبرامج. تهدف إدارة الذاكرة إلى ضمان استخدام ذاكرة الوصول العشوائي (RAM) بشكل فعال ، مما يحسن من أداء النظام واستقراره وأمانه. مع تجنب استفادتها أو حدوث تسربات. هناك العديد من تقنيات إدارة الذاكرة المختلفة، ويجب على المبرمجين استخدامها بشكل مناسب لكتابة برامج قوية وفعالة.

## ملاحظات:

- تختلف تقنيات إدارة الذاكرة بين لغات البرمجة المختلفة. وأيضاً تختلف باختلاف نظام التشغيل المستخدم.
- من المهم فهم قواعد لغة البرمجة المستخدمة لإدارة الذاكرة بشكل صحيح.
- يمكن للمبرمجين استخدام تقنيات إدارة الذاكرة في برامجهم لتحسين استخدام الذاكرة وكفاءة النظام.
- يمكن للمستخدمين تغيير بعض إعدادات إدارة الذاكرة، مثل حجم ذاكرة التخزين المؤقت.

## تطبيق من تطبيقات المؤشرات (Pointers) :

### ؟ التخصيص الديناميكي للذاكرة (Dynamic Memory Allocation)

**مقدمة:** الموارد دائماً محدودة وقيمة، لهذا نسعى دائماً لاستخدامها بكفاءة. وهذا ينطبق على الذاكرة في الحواسيب أيضاً. المتغيرات التي نستخدمها في البرمجة تحتاج إلى تخصيص مساحة تخزين لها في الذاكرة حتى يتمكن الحاسوب من التعامل معها. ولكن كيف يتم هذا التخصيص؟

إنشاء متغير في الحاسوب يمر بعدة مراحل، وليس مجرد عملية بسيطة كما قد نتصور. فهم تخصيص الذاكرة وأنواعها مهم لكتابه برامج فعالة وتجنب مشاكل الذاكرة.

#### ؟ مراحل إنشاء متغير:

- **التعريف:** عند تعريف متغير في الكود، نحدد اسمه ونوعه (مثل `int`)، لكن لم يتم تخصيص أي مساحة له في الذاكرة بعد.

- **التجميع:** يقوم المترجم (`compiler`) بتحويل الكود إلى لغة الآلة، ويخصص مكاناً للمتغير في قسم معين من الذاكرة بناءً على نوعه.

- **التشغيل:** عند تشغيل البرنامج، يتم تحميله في الذاكرة وتُخصص فعلياً المساحة المخصصة للمتغير، وتصبح جاهزة للاستخدام.

### ؟ تخصيص الذاكرة الثابت (Static) والمتغير (Dynamic) في C++

عندما نفكّر في إنشاء شيء ما، نتخيل عادةً بناءً من الصفر. لكن عندما ينشئ الحاسوب متغيراً جديداً، لا يحدث الأمر بهذه الطريقة تماماً. بالنسبة له، فإن إنشاء المتغير أقرب إلى التخصيص، حيث يقوم بتعيين خلية ذاكرة من مجموعة كبيرة من الخلايا الموجودة مسبقاً لهذا المتغير. **فكرة الأمر مثلاً يتم تخصيص غرفة فندقية لشخص ما من عدة غرف فارغة موجودة مسبقاً.** هذا التشبيه يوضح كيف يقوم الحاسوب بتخصيص الذاكرة للمتغيرات.

#### ما هو تخصيص الذاكرة الثابت (Static Memory Allocation)؟

- تخصيص الذاكرة الثابت مناسب للمتغيرات المعروفة مسبقاً، عندما نقوم بإعلان متغيرات، يقوم المترجم بتخصيص مساحة تخزين لها في الذاكرة ( تماماً كما قمنا بجز الغرف في التشبيه السابق ) يحدث هذا قبل تنفيذ البرنامج، ولا يمكننا استخدام هذه الطريقة لتخصيص متغيرات جديدة أثناء تشغيل البرنامج.

#### ما هو تخصيص الذاكرة المتغير أو (التخصيص الديناميكي للذاكرة) (Dynamic Memory Allocation)

- على عكس التخصيص التلقائي أو الثابت الذي يحدث للمتغيرات الثابتة وال محلية.
- يتم استخدام مفهوم التخصيص الديناميكي للذاكرة (`Dynamic Memory Allocation`) عندما يكون حجم الذاكرة المطلوب غير معروف في وقت التجميع (`compile time`). وذلك لتحقيق مرونة أكبر في إدارة الذاكرة حسب الحاجة أثناء تشغيل البرنامج.

#### تنفيذ التخصيص الديناميكي للذاكرة (Dynamic Memory Allocation) :

- يتم تنفيذ التخصيص الديناميكي للذاكرة عن طريق المؤشرات (Pointers) بتخصيص مساحة معينة يدوياً من قبل المبرمج و تخزين متغيرات بها أثناء تشغيل او تنفيذ البرنامج وذلك باستخدام الأمر `New` في `C++`. وبعد الانتهاء من استخدام الذاكرة، يجب تحريرها باستخدام الأمر `Delete` ، لتجنب تسربات الذاكرة. ويمكن استخدام المؤشرات الفارغة (`void pointers`) لتخصيص الذاكرة لأي نوع من البيانات، وبعد تخصيص الذاكرة، يجب القيام بعملية تحويل النوع (`type casting`) لاستخدام الذاكرة المخصصة مع نوع بيانات محدد.

#### مثال توضيحي:

إعلان مؤشر من نوع مناسب - `:declare an int pointer -`

```
int* ptr;
```

تخصيص الذاكرة باستخدام `new` - `:dynamically allocate memory -`

```
ptr = new int;
```

الوصول إلى الذاكرة المخصصة - (تعيين قيمة للذاكرة) `(assigning value to the memory)`

```
*ptr = 10;
```

تحرير الذاكرة عند الانتهاء باستخدام `delete` - `:deallocate the memory -`

```
delete ptr;
```

## ملاحظات هامة:

- يُستخدم عامل **new** في لغة C++ لتخصيص مساحة تخزين ديناميكياً، أي أثناء تشغيل البرنامج، إذ تقوم بإرجاع مؤشر إلى الذاكرة المخصصة. هذا يعني أنه يمنحك القدرة على تحديد حجم الذاكرة المطلوبة في الوقت الحالي، وليس تحديدها مسبقاً عند تعریف المتغير.
- يجب تحرير الذاكرة المخصصة بإستخدام **Delete** عند الانتهاء من استخدامها، وذلك لتفادي تسربات الذاكرة (**Memory Leaks**).
- كل **new** مقابلاً لها **delete** حتى تستفاد من **dynamic memory**.
- استخدام المؤشرات الفارغة: يمكن استخدام المؤشرات الفارغة (**void\***) لتخصيص ذاكرة لأي نوع من البيانات، حيث لا ترتبط بنوع محدد.
- يجب الحذر عند استخدام المؤشرات الفارغة (**Void Pointers**) في التخصيص الديناميكي للذاكرة، حيث تتطلب تحويل النوع (**Type Casting**) بحرص لتفادي الأخطاء.
- تحويل النوع: بعد تخصيص الذاكرة باستخدام **new**، يجب تحويل المؤشر الفارغ إلى نوع مؤشر محدد يتوافق مع نوع البيانات المراد تخزينها في الذاكرة قبل استخدامها.
- وبعد الانتهاء من استخدام الذاكرة، يجب تحريرها باستخدام أمر مثل **delete** ، لتجنب تسربات الذاكرة.

## عبارة:

"The new operator is used to return the address of the dynamically allocated memory, which is already stored in the pointer a"

توضح عملية تخصيص الذاكرة الديناميكي باستخدام عامل **new** في C++ . ويمكن شرحها بالتفصيل كالتالي:

**المتغير a :** يحتوي هذا المتغير على **مؤشر**، وليس قيمة مباشرة. وهذا يعني أنه **يُشير إلى** موقع محدد في الذاكرة، وليس قيمة مخزنة في ذلك الموقع.

**استخدام عامل new :** عندما تستخدم عامل **new** مع نوع معين من البيانات (مثل **int** أو **double**)، فهو يقوم بتخصيص مساحة تخزين جديدة في الذاكرة لهذا النوع من البيانات.

**إرجاع عنوان الذاكرة :** لا يعيد عامل **new** القيمة التي تم تخزينها في الذاكرة المخصصة، بل **يعيد عنوان** تلك الذاكرة.

**تخزين العنوان في a :** يتم تخزين عنوان الذاكرة المخصصة ديناميكياً في المتغير **a**، مما يعني أن **a الآن يُشير إلى** بداية تلك الذاكرة المخصصة. **معنـى آخر**، بدلاً من تخزين قيمة مباشرة في المتغير **a**، يتم استخدام عامل **new** لتخصيص مساحة تخزين جديدة ديناميكياً، ثم **تخزين عنوان** تلك المساحة في **a** بحيث يمكن الوصول إلى البيانات المخزنة لاحقاً.

## ملاحظة مهمة:

من الضروري تحرير الذاكرة المخصصة ديناميكياً بعد الانتهاء من استخدامها، وذلك باستخدام عامل **delete**. وإلا ستحدث تسربات للذاكرة، مما يؤدي إلى مشاكل وتباطؤ في البرنامج.

## لماذا يُطلق عليه تخصيص ديناميكي؟

- لأن عملية التخصيص تحدث بطريقة ديناميكية أثناء تشغيل البرنامج، وليس بشكل ثابت مسبقاً.

## أين يتم تخزين الذاكرة المخصصة ديناميكياً؟

- أما المتغيرات غير الثابتة وال محلية فتُخزن في منطقة الـ **(Stack)**.
- يتم تخزين الذاكرة المخصصة ديناميكياً في منطقة الـ **(Heap)**.

هناك طريقتان رئيسيتان لتخفيض الذاكرة للمتغيرات:

- التخصيص الثابت (على Stack):** يستخدم للمنطقة المعروفة بالـ **(Stack)** لتخفيض المتغيرات المحلية والمتغيرات الثابتة. يتم تخصيص الذاكرة بشكل تلقائي عند دخول الدوال وإلغاء تخصيصها عند الخروج.
- التخصيص الديناميكي (على Heap):** يستخدم منطقة الـ **(Heap)** لتخفيض المتغيرات التي **نخص** لها الذاكرة يدوياً باستخدام أوامر مثل **new** في C++. مسؤولية المبرمج هي **تحرير** الذاكرة بعد الانتهاء باستخدامها باستخدام **delete** لتجنب تسربات الذاكرة.

## الفرق بين التخصيص الثابت (Static) والتخصيص المتغير (Dynamic)

**التوقيت:** يحدث التخصيص الثابت (Static) قبل تشغيل البرنامج، بينما يحدث التخصيص المتغير (Dynamic) أثناء التنفيذ أو التشغيل.

**المرونة:** التخصيص الثابت (Static) (محدود) أقل مرونة لأنه يقتصر على المتغيرات المعلنة مسبقاً، بينما يتيح التخصيص المتغير (Dynamic) (مرونة) إنشاء متغيرات جديدة وفق الحاجة.

**المسؤولية:** المسؤولية عن تحرير الذاكرة للبرنامج في التخصيص الثابت (Static) (أقل مرونة)، بينما يتحمل المبرمج مسؤولية تحرير الذاكرة للمتغيرات المخصصة ديناميكياً (Dynamic) (أكثر مرونة) بعد الاستخدام لمنع تسربات الذاكرة.

## لماذا نحتاج إلى تخصيص ديناميكي للذاكرة أثناء تشغيل البرنامج غير التخصيص الثابت؟

- رغم أن التخصيص الثابت للذاكرة يستطيع القيام بالمهام في بعض الحالات، إلا أن هناك حاجة ملحة إلى طريقة أخرى لتخصيص الذاكرة، وهي التخصيص الديناميكي.

### 1. عدم معرفة حجم البيانات (حجم الذاكرة المطلوب) مسبقاً:

في بعض الحالات، لا نستطيع تحديد حجم البيانات المطلوبة مسبقاً، مثل:

- قراءة بيانات غير محددة الطول من المستخدم (مثل ملف نصي).
  - إنشاء هياكل بيانات ديناميكية مثل قوائم أو أشجار مرتبطة، حيث يعتمد حجمها على المدخلات.
- في هذه الحالات، لا يمكننا استخدام التخصيص الثابت لأننا لا نعرف مقدار الذاكرة المطلوب.

### 2. إنشاء هياكل بيانات معقدة:

- بعض هياكل البيانات، مثل القوائم والأشجار المرتبطة، تنمو وتتقلص بشكل ديناميكي أثناء تشغيل البرنامج.
- التخصيص الثابت غير مناسب لهذه الهياكل لأننا لا نعرف حجمها النهائي مسبقاً.
- التخصيص الديناميكي يسمح لنا بتخصيص الذاكرة لهذه الهياكل حسب الحاجة، مما يجعلها أكثر مرونة وفعالية.

### 3. كفاءة استخدام الذاكرة:

- في التخصيص الثابت، تخصص مقداراً ثابتاً من الذاكرة لمتغير حتى لو لم نستخدمه بالكامل، على سبيل المثال، إذا قمت بتخصيص مصفوفة لـ 100 عنصر لكنك تحتاج فقط إلى 20 عنصر، فإن 80 عنصراً ستظل مشغولة دون استخدام، هذا يمكن أن يؤدي إلى إهدار الذاكرة، خاصة إذا كان حجم البيانات المطلوبة أقل بكثير من الحجم المخصص.
- التخصيص الديناميكي يسمح لنا بتخصيص الذاكرة فقط للمقدار المطلوب، مما يحسن كفاءة استخدام الذاكرة ويقلل من إهدارها.

### 4. سهولة الإضافة والحذف في الهياكل الديناميكية (Dynamic structures),

- في القوائم والهيئات الأخرى التي يتم إنشاؤها ديناميكياً، يمكن إضافة أو حذف بيانات بسهولة عن طريق تعديل الروابط بين العناصر، وهذا غير ممكن مع التخصيص الثابت الذي يتطلب إعادة ترتيب البيانات واستهلاك المزيد من الذاكرة.

## بعض الأمثلة على استخدام التخصيص الديناميكي للذاكرة:

- **القوائم المرتبطة (Linked List)**: تتكون القوائم المرتبطة من عقد منفصلة يتم تخصيص الذاكرة لها ديناميكياً، حيث يربط كل عقد بالعقد التالي باستخدام مؤشرات.
- **الأشجار (Tree)**: مثل القوائم المرتبطة، تتطلب الأشجار أيضاً تخصيصاً ديناميكياً للذاكرة لعقدها، لأن حجم شجرة غير معروف مسبقاً ويمكن أن ينموا أو يتقلص أثناء تشغيل البرنامج.
- **المصفوفات الديناميكية**: على عكس المصفوفة التي يحدد المترجم حجمها، فإن المصفوفات الديناميكية يتم تخصيص الذاكرة لها ديناميكياً باستخدام **new**، مما يتيح للمبرمج تغيير حجمها بناءً على احتياجات البرنامج.

### خلاصة:

- بعد فهم تخصيص الذاكرة الثابت والمتغير أمراً مهماً لكتابة برامج C++ فعالة. استخدم التخصيص الثابت للمتغيرات المعروفة مسبقاً و استخدم التخصيص المتغير عندما تحتاج إلى إنشاء متغيرات ديناميكية. تذكر الحفاظ على إدارة سلية للذاكرة وتحريرها عند عدم الحاجة لمنع الأخطاء.

```
#include <iostream>
using namespace std;

int main()
{
    //declare an int pointer
    int* ptrN;
    float* ptrF;

    //dynamically allocate memory
    ptrN = new int;
    ptrF = new float;

    //assigning value to the memory
    *ptrN = 10;
    *ptrF = 2.4;

    cout << "\n\tTo print *ptrN : " << *ptrN << endl;
    cout << "\n\tTo print *ptrF : " << *ptrF << endl;

    //deallocate the memory
    delete ptrN;
    delete ptrF;
    cout << "\n-----\n";
    cout << "\n\tTo print *ptrN after delete : " << *ptrN << endl;
    cout << "\n\tTo print *ptrF after delete : " << *ptrF << endl;

    return 0;
}
```





# Lesson #44 - Dynamic Arrays: new and delete

## تعريف المصفوفة الديناميكية (Dynamic Array):

- هي مصفوفة يتم تخصيص حجمها بشكل تلقائي أثناء التشغيل (run time) ويمكن تغيير حجمها لاحقاً في البرنامج.
- وهي بنية بيانات تُستخدم لتخزين مجموعة من البيانات من نفس النوع.
- على عكس المصفوفات الثابتة، التي يكون حجمها ثابتاً عند الإنشاء، ولا يمكن تغييره لاحقاً.
- توفر المصفوفات الديناميكية مرونة أكبر في التعامل مع البيانات ذات الحجم غير المعروف مسبقاً.

## استخدامات المصفوفات الديناميكية:

- تخزين بيانات ذات حجم غير معروف مسبقاً: مثل بيانات مدخلة من المستخدم أو بيانات يتم استخراجها من ملف.
- تخزين بيانات يتم إضافتها أو إزالتها بشكل متكرر: مثل قائمة جهات الاتصال أو قائمة المشتريات.
- إنشاء هياكل بيانات معقدة: مثل القوائم المرتبطة (Linked list) أو الأشجار (Trees).

## أمثلة على استخدام المصفوفات الديناميكية:

- تخزين درجات الطلاب في فصل دراسي: يمكن استخدام مصفوفة ديناميكية لتخزين درجات الطلاب في فصل دراسي، حيث يمكن إضافة درجات جديدة للطلاب أو إزالتها لاحقاً.
- إنشاء قائمة جهات اتصال: يمكن استخدام مصفوفة ديناميكية لإنشاء قائمة جهات اتصال، حيث يمكن إضافة جهات اتصال جديدة أو إزالتها من القائمة لاحقاً.
- تخزين بيانات مستخدمي موقع إلكتروني: يمكن استخدام مصفوفة ديناميكية لتخزين بيانات مستخدمي موقع إلكتروني، مثل أسماء المستخدمين وكلمات المرور، حيث يمكن إضافة مستخدمين جدد أو إزالة مستخدمين من الموقع لاحقاً.

## Dynamic Arrays: New and Delete

### خطوات عمل المصفوفة الديناميكية:

#### 1. تعريف المؤشر:

يتم تعريف مؤشر من نوع البيانات الذي سيتم تخزينه في المصفوفة.

```
int* pArray;
```

#### 2. الإنشاء (تخصيص الذاكرة):

يتم إنشاء المصفوفة الديناميكية بتخصيص مساحة ذاكرة جديدة في منطقة الـ(heap) ويتم ربطها بمؤشر وذلك باستخدام الكلمة المفتاحية new مع تحديد نوع البيانات وحجمها.

على سبيل المثال، ينشئ الكود التالي مصفوفة ديناميكية من 5 عناصر من نوع int:

```
int* pArray = new int[5];
```

#### 3. تهيئة العناصر:

يمكن تهيئة عناصر المصفوفة عند إنشائها باستخدام قيم محددة أو تركها فارغة.

```
for (int i = 0; i < 5; i++)  
{  
    pArray[i] = i;  
}
```

#### 4. الوصول إلى العناصر:

يتم استخدام المؤشر والفهرس للوصول إلى عناصر المصفوفة الديناميكية في مساحة الذاكرة المخصصة، حيث يتم حساب موقع كل عنصر بناءً على حجم البيانات ونوعها. الفهرس هو رقم العنصر الذي تريد الوصول إليه. على سبيل المثال، يُستخدم الكود التالي لقراءة قيمة من العنصر الثاني في المصفوفة:

```
int value = pArray[1];
```

يمكن أيضًا استخدام المؤشرات للتغيير قيم عناصر المصفوفة.

#### 5. تغيير حجم المصفوفة:

يمكن تغيير حجم المصفوفة الديناميكية أثناء التشغيل باستخدام تقنيات إعادة تخصيص الذاكرة تتضمن هذه التقنيات استخدام الكلمات المفتاحية **new** و **delete**. يتم تخصيص مساحة ذاكرة جديدة بحجم مختلف، ويتم نسخ عناصر المصفوفة القديمة إلى المساحة الجديدة. ثم يتم تحرير الذاكرة المخصصة للمصفوفة القديمة.

على سبيل المثال، ينشئ الكود التالي مصفوفة ديناميكية جديدة بحجم 10 عناصر ويقوم بنسخ عناصر المصفوفة القديمة إليها:

```
int* pNewArray = new int[10];  
  
for (int i = 0; i < 5; i++)  
{  
  
    pNewArray[i] = pArray[i];  
  
}
```

#### 6. تحرير الذاكرة:

عند الانتهاء من استخدام المصفوفة الديناميكية، يجب تحرير الذاكرة المخصصة للمصفوفة القديمة باستخدام عامل **[delete]** يمنع ذلك تسريبات الذاكرة.

```
delete[] pArray;
```

ثم تحرير الذاكرة المخصصة للمصفوفة الجديدة بعد استخدامها

```
delete[] pNewArray;
```

#### ملاحظات:

- تتطلب المصفوفات الديناميكية كتابة كود أكثر تعقيدًا من المصفوفات الثابتة.
- يجب الحرص على إدارة الذاكرة بشكل صحيح عند استخدام المصفوفات الديناميكية لتجنب تسريبات الذاكرة.
- يجب دائمًا تحرير الذاكرة المخصصة للمصفوفة الديناميكية عند الانتهاء من استخدامها.
- يمكن أن تكون عملية تغيير حجم المصفوفة الديناميكية معقدة، خاصة إذا كانت تحتوي على بيانات كبيرة.

#### مميزات المصفوفات الديناميكية:

- مرونة أكبر في التعامل مع البيانات ذات الحجم غير المعروف مسبقًا.
- إمكانية تغيير حجم المصفوفة أثناء التشغيل.

#### عيوب المصفوفات الديناميكية:

- تتطلب كتابة كود أكثر تعقيدًا من المصفوفات الثابتة.
- تتطلب إدارة ذاكرة أكثر حرصًا لتجنب تسريبات الذاكرة.

#### اختيار نوع المصفوفة:

- يعتمد اختيار نوع المصفوفة على احتياجاتك الخاصة وحجم البيانات التي ستتعامل معها.
- إذا كان حجم البيانات معروفاً مسبقاً ولا يتغير، فإن المصفوفة الثابتة هي الخيار الأفضل.
- أما إذا كان حجم البيانات غير معروف مسبقاً أو قد يتغير، فإن المصفوفة الديناميكية هي الخيار المناسب.

## الاختلافات بين المصفوفات الثابتة والдинاميكية في لغة C++ :

### المصفوفات الثابتة (Static Arrays)

**تحديد الحجم تلقائياً:** يتم تحديد حجم المصفوفة الثابتة تلقائياً أثناء التهيئة، بناءً على عدد القيم المُعطاة داخل الأقواس المربعة. في المثال المذكور، تم تهيئة المصفوفة بـ 5 قيم، لذلك سيكون حجمها 5.

**تخصيص الذاكرة على الـ Stack:** يتم تخصيص الذاكرة للمصفوفة الثابتة على منطقة الذاكرة المعروفة باسم الـ (Stack)، وهي منطقة ذاكرة ذات حجم محدود ومخصصة للمتغيرات المحلية والدوال.

**حجم ثابت:** لا يمكن تغيير حجم المصفوفة الثابتة بعد تعريفها؛ يبقى حجمها ثابتاً طوال فترة تشغيل البرنامج.

### المصفوفات الديناميكية (Dynamic Arrays)

**تخصيص الذاكرة أثناء التنفيذ:** يتم تخصيص الذاكرة للمصفوفة الديناميكية أثناء تشغيل البرنامج، باستخدام الكلمة الأساسية `new`.

**تحديد الحجم صراحةً:** يجب تحديد حجم المصفوفة الديناميكية بشكل صريح عند التخصيص بين قوسين مربعين، مثل `new int[5]`.

**تخصيص الذاكرة على الركام (Heap):** يتم تخصيص الذاكرة للمصفوفة الديناميكية على منطقة الذاكرة المعروفة باسم الـ (Heap)، وهي منطقة ذاكرة أكبر وأكثر مرونة من الـ (Stack).

**إمكانية تغيير الحجم:** يمكن تغيير حجم المصفوفة الديناميكية لاحقاً، باستخدام عامل `delete[]` لتحرير الذاكرة الحالية، ثم تخصيص كتلة (مساحة) ذاكرة جديدة بالحجم المطلوب باستخدام `new` مرة أخرى.

**تحرير الذاكرة:** عند الانتهاء من استخدام المصفوفة الديناميكية، يجب تحرير الذاكرة المخصصة لها باستخدام عامل `delete[]` لتجنب تسريبات الذاكرة.

### توصيات:

- اختيار نوع المصفوفة يعتمد على احتياجاتك الخاصة وحجم البيانات التي ستتعامل معها.
- استخدم المصفوفات الثابتة عندما تعرف حجم البيانات مسبقاً وتريد كفاءة أعلى في الاستخدام لكن أقل مرونة وأمنة أكثر بخصوص الوصول إلى العناصر، حيث يتم التحقق من حدودها تلقائياً.
- استخدم المصفوفات الديناميكية عندما تحتاج إلى مرونة أكبر في تغيير حجم المصفوفة أثناء تشغيل البرنامج مما يجعلها مناسبة للتعامل مع بيانات ذات حجم غير معروف مسبقاً.
- إذا كنت بحاجة إلى تغيير حجم المصفوفة أثناء التشغيل أو كنت غير متأكد من الحجم المطلوب، فإن المصفوفات الديناميكية ضرورية.
- كن حذراً عند استخدام المصفوفات الديناميكية لتجنب تسريبات الذاكرة، وتأكد من تحرير الذاكرة باستخدام `delete[]` عند الانتهاء من استخدامها.

## الفرق الرئيسي بين المصفوفات الثابتة والمصفوفات الديناميكية في لغة C++ :

المصفوفة الديناميكية	المصفوفة الثابتة	الخاصة
صراحة باستخدام المعامل new	تلقائياً عند التهيئة	تحديد الحجم
أثناء تشغيل البرنامج (run time)	أثناء عملية الترجمة (compile time)	تحصيص الذاكرة
غير ثابت (يمكن تعديله) وقابل للتغيير	ثابت (محدد مسبقاً) وغير قابل للتغيير	حجم المصفوفة
ممكн باستخدام New و []	غير ممكн	تغيير الحجم
أثناء وقت التشغيل	أثناء التجميع	وقت التخصيص
في ذاكرة (Heap)	في ذاكرة (Stack)	موقع الذاكرة
صعب	سهل	إدارة الذاكرة
أكبر	أقل	المرونة
أكبر	أقل	التعقيد
قيم محددة من المستخدم	صفر أو سلاسل فارغة	تهيئة العناصر
ممكн (باستخدام تقنيات إعادة تحصيص الذاكرة)	غير ممكن	تعديل العناصر
ممكنة (باستخدام عامل [delete])	غير ممكنة	إزالة المصفوفة
غير ممكنة	غير ممكنة	قراءة/كتابة العناصر بعد الإزالة
لا يمكن قراءة أو كتابة العناصر	يمكن تهيئتها مرة أخرى	التحكم بعد التدمير

يوضح هذا البرنامج عملية تحصيص بـ(New) وتحرير بـ>Delete) لمصفوفة ديناميكية.

1. طلب إدخال عدد الطالب:

يطلب البرنامج من المستخدم إدخال عدد الطالب ويحذن هذه القيمة في المتغير num.

```
int num;
cout << "Enter total number of students: ";
cin >> num;
```

2. تعريف المؤشر:

يتم تعريف مؤشر من نوع البيانات الذي سيتم تخزينه في المصفوفة.

تعريف مؤشر ptr من نوع float

```
float* ptr;
```

3. تحصيص الذاكرة للمصفوفة ديناميكياً:

يتم تحصيص الذاكرة للمصفوفة الديناميكية في منطقة الـ (Heap) في الذاكرة.

يستخدم البرنامج عامل new لتحصيص مساحة تخزين ديناميكياً لمصفوفة من قيم من نوع float. حجم هذه المساحة يعتمد على قيمة num التي أدخلها المستخدم.

```
// memory allocation of num number of floats
ptr = new float[num];
```

## 4. إدخال البيانات وتخزينها في المصفوفة:

يتم استخدام علامة النجمة (\*) مع المؤشر **ptr** للوصول إلى عناصر المصفوفة. البرنامج يدور في حلقة يعتمد على قيمة **للتكرار** ، بحيث يدخل قيمة **float** لكل عنصر في المصفوفة باستخدام **[ptr][i]** .

يتم تخزين عنوان الذاكرة المخصصة في المتغير **ptr** - بمعنى: العنوان الذي يشير إلى **بداية** مساحة التخزين المخصصة حديثاً يتم تخزينه في المتغير (**ptr + 0** اللذي هو **ptr**).

**مثال:** يشير **0** إلى **أول** عنصر في المصفوفة ، ويشير **1** إلى **ثاني** عنصر في المصفوفة ، ويشير **2** إلى **ثالث** عنصر في المصفوفة... وهكذا....

```
cout << "Enter GPA of students." << endl;
for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ":" ;
    cin >> *(ptr + i);
}
```

## 5. طباعة عناصر المصفوفة:

بعد إدخال جميع البيانات، يطبع البرنامج قيم **float** المخزنة في كل عنصر من عناصر المصفوفة باستخدام نفس علامة النجمة (\*) مع نفس المؤشر **ptr** للوصول إلى العناصر.

**مثال:** يشير **0** إلى **أول** عنصر في المصفوفة ، ويشير **1** إلى **ثاني** عنصر في المصفوفة ، ويشير **2** إلى **ثالث** عنصر في المصفوفة... وهكذا....

```
cout << "\nDisplaying GPA of students." << endl;
for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ":" << *(ptr + i) << endl;
}
```

## 6. تحرير الذاكرة المخصصة:

بعد الانتهاء من استخدام المصفوفة، من المهم تحرير الذاكرة المخصصة لها لتجنب تسربات الذاكرة. يستخدم البرنامج **delete[] ptr** لتحرير الذاكرة. لاحظ استخدام الأقواس المربعة [] بعد **delete** لأننا نتعامل مع تحرير ذاكرة مصفوفة وليس متغير مفرد.

```
// ptr memory is released
delete[] ptr;
```

### ملاحظات مهمة:

- يستخدم عامل **new** لخصيص مساحة تخزين ديناميكية لمصفوفة **float** بناءً على **عدد محدد** من المستخدم.
- يستخدم علامة النجمة (\*) مع المؤشر **ptr** للوصول إلى عناصر المصفوفة وإدخال البيانات وطباعتها.
- نقوم **بحرير** الذاكرة المخصصة للمصفوفة باستخدام **.delete[] ptr**.
- يوفر هذا البرنامج مثلاً على كيفية استخدام تخصيص وتحريز الذاكرة الديناميكية بشكل صحيح في لغة **C++** .

```

#include <iostream>
using namespace std;

int main() {

    int num;
    cout << "\n\tEnter total number of students: ";
    cin >> num;

    float* ptr;

    // memory allocation of num number of floats
    ptr = new float[num];

    cout << "\n\tEnter GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "\tStudent" << i + 1 << ": ";
        cin >> *(ptr + i);
    }

    cout << "\n\tDisplaying GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "\tStudent" << i + 1 << ": " << *(ptr + i) << endl;
    }

    // ptr memory is released
    delete[] ptr;

    return 0;
}

```

Microsoft Visual Studio Debug Console

```

Enter total number of students: 3
Enter GPA of students.
Student1: 56
Student2: 76
Student3: 98

Displaying GPA of students.
Student1: 56
Student2: 76
Student3: 98

```



# Lesson #45 - Stack vs Heap

## RAM: Memory

4

Heap: any dynamic variables/objects/arrays..etc

3

Stack: local variables/functions/pointers

2

Static / Global

1

Source Code/ Instructions

### تقسيم الذاكرة العشوائية (RAM):

الذاكرة العشوائية (RAM) هي ذاكرة الوصول السريع التي يستخدمها المعالج لتخزين البيانات والتعليمات البرمجية التي يتم استخدامها حاليا. تُقسم الذاكرة العشوائية إلى أقسام مختلفة لتسهيل إدارة الذاكرة وتحسين الأداء.

#### أقسام الذاكرة العشوائية:

- ذاكرة النظام (System Memory): تُستخدم ذاكرة النظام لتخزين نظام التشغيل والبرامج قيد التشغيل.
- ذاكرة التطبيقات (Application Memory): تُستخدم ذاكرة التطبيقات لتخزين بيانات وبرامج المستخدم.
- ذاكرة الفيديو (Video Memory): تُستخدم ذاكرة الفيديو لتخزين الصور ومقاطع الفيديو التي يتم عرضها على الشاشة.
- ذاكرة القرص الصلب (Hard Disk Memory): تُستخدم ذاكرة القرص الصلب لتخزين البيانات بشكل دائم، حتى عند إيقاف تشغيل الكمبيوتر.

#### آليات تقسيم الذاكرة:

- ال التقسيم الثابت: يتم تخصيص حجم ثابت لكل قسم من أقسام الذاكرة عند بدء تشغيل الكمبيوتر.
- ال التقسيم الديناميكي: يتم تخصيص الذاكرة لأقسام مختلفة حسب الحاجة، مما يسمح باستخدام الذاكرة بشكل أكثر كفاءة.

#### عوامل تؤثر على تقسيم الذاكرة:

- نظام التشغيل: يُحدد نظام التشغيل كيفية تقسيم الذاكرة وإدارة استخدامها.
- البرامج قيد التشغيل: تؤثر كمية البيانات والتعليمات البرمجية التي تحتاجها البرامج قيد التشغيل على كيفية تقسيم الذاكرة.
- الأجهزة: تؤثر كمية الذاكرة العشوائية المتوفرة على كيفية تقسيمها.

#### خلاصة:

- تقسيم الذاكرة العشوائية هو عملية مهمة لضمان استخدام الذاكرة بشكل كفء وتحسين أداء الكمبيوتر. تُقسم الذاكرة العشوائية إلى أقسام مختلفة لتسهيل إدارة الذاكرة وتلبية احتياجات نظام التشغيل والبرامج قيد التشغيل.

## ذاكرة التطبيقات او البرامج (Application Memory)

- ذاكرة البرامج هي جزء مخصص بشكل ديناميكي من ذاكرة الوصول العشوائي (RAM).
- في هذه الذاكرة يتم تخزين كل ما يحتاجه البرنامج ليعمل بشكل صحيح.
- يتم استخدام هذا الجزء لتشغيل البرنامج التي يستخدمها المستخدم ، وعندما يتم إغلاق البرنامج، يتم تحرير ذاكرة البرنامج وإعادتها إلى النظام.

### مميزات ذاكرة التطبيقات:

- динамيكية:** يتم تخصيص ذاكرة البرنامج بشكل ديناميكي، حيث يتم تخصيص المزيد من الذاكرة للبرامج التي تحتاجها.
- مؤقتة:** يتم تحرير ذاكرة البرنامج وإعادتها إلى النظام عند إغلاق البرنامج.
- محصصة:** تُخصص ذاكرة البرنامج لكل برنامج على حدة، مما يمنع البرامج من التدخل في بعضها البعض.

### كيف تعمل ذاكرة التطبيقات؟

- عندما يتم تشغيل برنامج، يقوم نظام التشغيل بتخصيص مساحة من ذاكرة البرنامج للبرنامج.
- يتم تخزين التعليمات البرمجية والبيانات التي يستخدمها البرنامج في هذه المساحة المخصصة في ذاكرة البرنامج.
- يقوم البرنامج بتحميل التعليمات البرمجية والبيانات التي يحتاجها من ذاكرة البرنامج إلى وحدة المعالجة المركزية (CPU) لمعالجتها.
- عندما ينتهي البرنامج من العمل أو يتم إغلاقه ، يقوم النظام بتحرير ذاكرة البرنامج وإعادتها إلى النظام.

### أنواع ذاكرة التطبيقات:

يمكن تقسيم ذاكرة البرنامج إلى أربعة أقسام رئيسية بناءً على نوع البيانات المخزنة فيها:

- ذاكرة الكود (Code Memory)** أو ذاكرة التعليمات البرمجية : تُستخدم لتخزين التعليمات البرمجية التي تُشكل البرنامج.
- ذاكرة البيانات (Data Memory)**: تُستخدم لتخزين البيانات التي يستخدمها البرنامج أثناء التشغيل.
- ذاكرة المكدس (Stack Memory)**: تُستخدم لتخزين المتغيرات المحلية لوظائف البرنامج مؤقتًا أثناء تشغيل البرنامج.
- ذاكرة الرَّكَام (Heap Memory)**: تُستخدم لتخزين البيانات ذات الحجم الديناميكي التي يتم تخصيصها ديناميكياً أثناء تشغيل البرنامج مثل القوائم والسلالسل.

### ما هي العوامل التي تؤثر على حجم ذاكرة التطبيقات التي يحتاجها البرنامج؟

- حجم البرنامج:** كلما زاد حجم البرنامج، زادت ذاكرة البرنامج التي يحتاجها.
- تعقيد البرنامج:** كلما زاد تعقيد البرنامج، زادت ذاكرة البرنامج التي يحتاجها.
- كمية البيانات التي يستخدمها البرنامج:** كلما زادت كمية البيانات التي يستخدمها البرنامج، زادت ذاكرة البرنامج التي يحتاجها.
- عدد البرامج التي تعمل في نفس الوقت:** كلما زاد عدد البرامج التي تعمل في نفس الوقت، زادت ذاكرة البرنامج المستخدمة.

### بعض المشاكل التي قد تحدث مع ذاكرة التطبيقات:

- نقص ذاكرة البرنامج:** قد يحدث نقص في ذاكرة البرنامج إذا كان البرنامج بحاجة إلى ذاكرة أكثر مما هو متاح.
- تسربات ذاكرة البرنامج:** إذا لم يتم تحرير ذاكرة البرنامج بشكل صحيح عند إغلاق البرنامج، فقد تحدث تسربات ذاكرة البرنامج، مما قد يؤدي إلى تدهور أداء النظام.

### نصائح لإدارة ذاكرة التطبيقات بشكل فعال:

- إغلاق البرامج التي لا يتم استخدامها:** يساعد ذلك على تحرير ذاكرة البرنامج التي يمكن استخدامها من قبل البرامج الأخرى.
- تحديث البرنامج:** قد تتضمن التحديثات تحسينات في استخدام ذاكرة البرنامج أو إصلاحات لتسربات ذاكرة البرنامج.
- استخدام أدوات مراقبة ذاكرة البرنامج:** تساعد هذه الأدوات على مراقبة استخدام ذاكرة البرنامج وتحديد البرامج التي تستهلك الكثير من الذاكرة.
- استخدام برامج مضادة للفيروسات:** يمكن للبرامج الضارة استهلاك الكثير من ذاكرة البرنامج.
- ترقية ذاكرة الوصول العشوائي (RAM):** إذا كانت ذاكرة البرنامج غير كافية لتشغيل البرنامج، يمكن ترقية ذاكرة الوصول العشوائي (RAM) .

## ملاحظات:

- يُعد **Stack** و **Heap** من أهم هياكل البيانات في علوم الحاسوب، ويجب على كل مبرمج فهم كيفية عملها واستخدامها بشكل صحيح.
- يُقسم التطبيق قِسم ذاكرة البرامج الخاص به إلى أقسام فرعية لتنظيم مختلف أنواع البيانات التي يستخدمها.
- ذاكرة البرامج هي ساحة عمل البرنامج، حيث يتم تخزين كل ما يحتاجه البرنامج ليعمل بشكل صحيح، وتنقسم إلى أقسام مختلفة لتسهيل التنظيم والوصول إلى البيانات.
- تتفاوت نسبة استخدام كل قسم من ذاكرة البرنامج حسب نوع البرنامج واحتياجاته.
- من المهم إدارة ذاكرة البرامج بكفاءة، حيث يمكن أن يؤدي عدم كفيتها إلى بطء أو توقف البرنامج.
- قد تختلف طريقة عمل ذاكرة البرامج اعتماداً على نظام التشغيل المستخدم.
- يمكن للمستخدمين تغيير بعض إعدادات ذاكرة البرنامج، مثل الحد الأقصى لحجم ذاكرة البرامج (**heap**) المخصصة لكل برنامج.

## أقسام ذاكرة التطبيقات او البرامج : (Application Memory Partitions)

1

### Source Code/ Instructions

#### 1. كود المصدر/التعليمات : (Source Code / Instructions)

- هذا الجزء من ذاكرة البرنامج يخزن التعليمات البرمجية الموجودة في الكود للبرنامج نفسه، وهي الأوامر التي يخبر بها البرنامج الكمبيوتر ما يجب القيام به. وهذه التعليمات البرمجية مكتوبة بلغة برمجة مثل **Python** أو **C++**.
- تشكل التعليمات البرمجية هيكل البرنامج وتحدد كيفية معالجته للبيانات.
- يتم تحميل هذه التعليمات البرمجية في ذاكرة البرنامج عند بدء تشغيل البرنامج.
- يشبه هذا الجزء وصفة طعام، حيث تحتوي على التعليمات خطوة بخطوة بخطوة لإنجاز مهمة معينة.
- تُترجم التعليمات إلى لغة يفهمها المعالج أثناء عملية الترجمة (**Compile Time**).
- لا يتغير هذا القسم أثناء تشغيل التطبيق، ولكنه يقرأ وتفسر من قبل المعالج لتنفيذ المهام المطلوبة.
- الموقع: يتم تخزين تعليمات المصدر عادةً على القرص الصلب، بينما يتم تحميل التعليمات المترجمة في ذاكرة البرنامج عند تشغيل البرنامج.

2

## Static / Global

### 2. بيانات ثابتة / عامة (Static / global)

- في هذا القسم، يتم تخزين المتغيرات الثابتة (Constants) التي لا تتغير خلال تشغيل البرنامج، بالإضافة إلى المتغيرات العامة (Global Variables) التي يمكن الوصول إليها من أي جزء من البرنامج.
- يحتوي هذا القسم على البيانات الثابتة والمعروفة مسبقاً التي يستخدمها البرنامج.
- تشمل أمثلة البيانات الثابتة قيم مثل القيمة "π" أو نص ثابت مثل "مرحباً بالعالم!"، بينما تشمل أمثلة البيانات العامة المتغيرات التي يمكن الوصول إليها من أي مكان في البرنامج.
- لا يتغير حجم هذه البيانات أثناء تشغيل البرنامج.
- الموقع: عادةً ما يتم تخزين المتغيرات الثابتة وال العامة في قسم من ذاكرة البرامج يسمى "ذاكرة البيانات".

3

## Stack: local variables/functions/pointers

### 3. الـ(Stack): متغيرات محلية (Pointers)/دوال (Functions)/local variables

- يكون أكبر من سابقيه ، يقوم نظام التشغيل بتخصيص مساحة تخزين في الذاكرة تعرف بالـ(Stack) وهو بنية بيانات (Data Structure) تشبه كومة من الأطباق، حيث يتم تخزين المعلومات بطريقة "أول ما يدخل هو آخر ما يخرج" (LIFO).
- يستخدم الـ(Stack) لتخزين المتغيرات المحلية (Local Variables) الخاصة بالدوال (Functions)، أي المتغيرات التي تنشأ مع استدعاء للدالة وتحذف عند انتهاء عمل تلك الدالة.
- يتم تخصيص الذاكرة في الـ(Stack) تلقائياً عند بدء تشغيل الدالة، ويتم تحريرها تلقائياً عند انتهاء الدالة.
- كما يُستخدم لتخزين عناوين الذاكرة (المؤشرات- Pointers) هو مفتاح للوصول إلى البيانات المخزنة في الـ(Heap)، (هو منفذ لأخذ مساحة من الـ Heap عند الحاجة إلى مساحة أكبر في الذاكرة).
- حجم البيانات المخزنة في الـ(Stack) يكون محدد مسبقاً ولا يتغير أثناء تشغيل البرنامج.
- لا يناسب الـ(Stack) لتخزين البيانات ذات الحجم динاميكي.

## Heap: any dynamic variables/objects/arrays..etc

### 4. الـ(Heap) : أي متغيرات ديناميكية (Dynamic Variables)/كائنات(Objects)/مصفوفات(Arrays)

- الرَّكَام هو بنية بيانات (Data Structure) مرنَّة تسمح بـتخصيص الذاكرة ديناميكيًا، أي أثناء تشغيل البرنامج. حيث يمكن إضافة وإزالة البيانات ديناميكيًا.
- يستخدم الـ(Heap) لـتخزين البيانات ذات الحجم الديناميكي أو غير المعروض مسبقاً، (مثل المتغيرات داخل الدوال التي يتم إنشاؤها باستخدام الكلمة المفتاحية New ، وحذفها أو تحريرها باستخدام الكلمة المفتاحية Delete) أو المصفوفات التي يتم إنشاؤها ديناميكيًا أو تغيير حجمها أثناء تشغيل البرنامج ، أو كائنات البرمجة (التي تجمع البيانات والوظائف معًا) التي يتم إنشاؤها وتحريرها ديناميكيًا.
- يتطلب تخصيص وتحرير الذاكرة في الـ(Heap) يدوياً من قبل المبرمج باستخدام الكلمات المفتاحية New و Delete.
- بسبب طبيعة التخصيص الديناميكي، يجب تحرير الذاكرة المخصصة في الـ(Heap) يدوياً لمنع تسربات الذاكرة.
- يُعد الـ(Heap) مناسباً لـتخزين أي نوع من البيانات، بما في ذلك المتغيرات والمسارات والأرقام والهياكل والمصفوفات.
- من المهم التمييز بين استخدام الـ(Stack) والـ(Heap) في إدارة ذاكرة البرامج.

### نصائح:

- يُفضل استخدام الـ(Stack) للمتغيرات المحلية التي لها حجم ثابت ومعروض مسبقاً.
- يُفضل استخدام الـ(Heap) للمتغيرات ديناميكية الحجم التي تحتاج إلى إنشاء أو حذف أثناء التشغيل وغير المعروض مسبقاً.
- يجب التأكد من تحرير الذاكرة المخصصة للـ(Heap) بالمعامل Delete يدوياً لمنع تسربات الذاكرة.

### موجز:

- يستخدم قسم الكود المصدر/التعليمات لـتخزين الشفرة البرمجية الثابتة للبرنامج.
- يستخدم القسم الثابت/العام لـتخزين المتغيرات الثابتة وال العامة ذات الحجم الثابت.
- يستخدم الـ(Stack) لـتخزين المتغيرات المحلية والوظائف والمؤشرات، ويتم تخصيص الذاكرة له مسبقاً.
- يستخدم الـ(Heap) لـتخزين أي متغيرات ديناميكية، كائنات، ومصفوفات، وتتطلب تحرير الذاكرة يدوياً.

- يشير مصطلح "Stack" في البرمجة إلى مكان في الذاكرة يستخدم لـ تخزين المتغيرات المحلية والمعلمات وعنوانين العودة للدوال.
- بينما يشير مصطلح "Heap" إلى مكان في الذاكرة يستخدم لـ تخزين المتغيرات الديناميكية.
- يتم تخصيص الذاكرة في Stack بشكل تلقائي من قبل المترجم، بينما يتم تخصيص الذاكرة في Heap يدوياً من قبل المبرمج.
- يتم التعامل مع الـ Heap بشكل أبطأ من الـ Stack .
- يحدث نقص الذاكرة بشكل أكثر احتمالاً في الـ Stack، بينما يحدث الانقسام في الذاكرة بشكل أكثر احتمالاً في الـ Heap .

## -: Memory Allocation #

هي بكل بساطة، عملية تقوم بتخصيص مساحة "معينه" في الذاكرة ليتم استخدامها أو كما قال lavanya marichamy (هو جزء من المعلومات يتخزن في الذاكرة) .

## -: Execution و Compilation #

الـ **compilation**: هي عملية لتحويل لغة برمجة (high-level language) إلى لغة برمجة (low-level machine-code) ← لينتج ملفاً تطبيقياً (compile-time).

الـ **execution**: هي العملية التي تكون بعد compilation وهي تنفيذ البرنامج ← (run-time).

## -: Memory allocation #

الـ **Static Allocation** - compilation : هي مساحة من الذاكرة يتم تخصيصها (internal linkage) بعد execution ولا يتم تحريرها "المبرمج لا يجدر عليه القلق من هذه الناحية لكن عليه أن يقلق من المساحة فالموضوع يحدث في Stack وليس Heap" ( تستخدم في Stack ).

الـ **Dynamic Allocation** - memory Stack : هي مساحة في Memory Stack وليس Heap يتم تخصيصها من dynamic memory و يجدر عليه تحريرها " function free() لتفادي memory leak ". المبرمج لا تستطيع الوصول له إلا عن طريق المؤشرات [Pointers] ( تستخدم في Heap ).



# Lesson #46 - Access Elements

## الوصول إلى عناصر ال Vector في لغة C++

طرق الوصول إلى عناصر ال Vector: الدالة at() وعامل الفهرسة []

الدالة : at()

- هي دالة تُستخدم للوصول إلى عنصر محدد في المتّجّه.
- تشبه وظيفتها عامل الفهرسة [] ، لكنها تُصدر استثناءً من نوع std::out\_of\_range إذا كان الفهرس المُقدّم غير صالح أو غير موجود.
- يمكن استخدامها لقراءة أو كتابة قيمة العنصر.
- يمكن أن تكون مفيدة في بعض الحالات لمنع حدوث أخطاء، وأيضاً في الحالات التي نريد فيها كتابة كود أكثر وضوحاً وسهولة في القراءة.
- قد تكون دالة at() أبطأ من عامل الفهرسة [] في بعض الحالات.

بنية الجملة:

vectorname.at(index)

المعلمات:

اسم المتّجّه: vectorname

فهرس العنصر المراد الوصول إليه، يبدأ من 0 وينتهي عند 1 - size().

النتيجة: تُرجع قيمة العنصر الموجود في الفهرس المُحدّد.

مثال:

```
vector<int> vnumbers = { 1,2,3,4,5 };

cout << "Element at index 2 = " << vnumbers.at(2) << endl;
// الوصول الى الفهرس 2 وطباعة قيمته (3)

cout << "Element at index 3 = " << vnumbers.at(3) << endl;
// الوصول الى الفهرس 3 وطباعة قيمته (4)
```

عامل الفهرسة [] :

- هو عبارة عن عامل يُستخدم للوصول إلى عنصر محدد في المتّجّه.
- يتم تحديد العنصر باستخدام فهرسه بين قوسين [].
- يبدأ الفهرس من 0 وينتهي عند حجم المتّجّه - 1.

بنية الجملة:

vectorname[index]

المعلمات:

اسم المتّجّه: vectorname

فهرس العنصر المراد الوصول إليه، يبدأ من 0 وينتهي عند 1 - size().

النتيجة: تُرجع قيمة العنصر الموجود في الفهرس المُحدّد.

مثال:

```
vector<int> vNum = { 1,2,3,4,5 };

cout << "Element at index 1 = " << vNum[1] << endl;
// الوصول الى الفهرس 1 وطباعة قيمته (2)

cout << "Element at index 4 = " << vNum[4] << endl;
// الوصول الى الفهرس 4 وطباعة قيمته (5)
```

## مقارنة بين الدالة `at()` و عامل الفهرسة [ ] :

دالة <code>at()</code>	عامل الفهرسة ( <code>operator[]</code> )	الخاصية
الوصول إلى عناصر المتجل	الوصول إلى عناصر المتجل	الوظيفة
<code>std::out_of_range</code> يُصدر استثناءً من نوع إذا كان الفهرس غير صالح	لا يُصدر استثناءات	إصدار الاستثناءات
قد يكون أبطأ في بعض الحالات	قد يكون أسرع في بعض الحالات	الأداء
أكثر تعقيداً	أسهل في الاستخدام	سهولة الاستخدام

### الاختيار بين الدالة `at()` و عامل الفهرسة [ ] :

- إذا كنت متأكداً من أن الفهرس صحيح، فاستخدم عامل الفهرسة [ ] لأنها أسرع.
- إذا لم تكن متأكداً من صحة الفهرس، فاستخدم دالة `at()` لمنع حدوث أخطاء.

#### ملخص:

- الدالة `at()` وعامل الفهرسة [ ] هما طريقتان للوصول إلى عناصر المتجل في لغة C++.
- عامل الفهرسة [ ] هو أكثر شيوعاً، بينما دالة `at()` أكثر أماناً.
- يعتمد اختيار الطريقة المناسبة على احتياجاتك الخاصة.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> Num{ 3, 4, 9, 1, 6 };

    cout << "\n\n Using .at(i) \n";

    cout << "Element at index 0 = " << Num.at(0) << endl;
    cout << "Element at index 1 = " << Num.at(1) << endl;
    cout << "Element at index 2 = " << Num.at(2) << endl;
    cout << "Element at index 3 = " << Num.at(3) << endl;
    cout << "Element at index 4 = " << Num.at(4) << endl;

    cout << "\n\n Using [i] \n";

    cout << "Element at index 0 = " << Num[0] << endl;
    cout << "Element at index 1 = " << Num[1] << endl;
    cout << "Element at index 2 = " << Num[2] << endl;
    cout << "Element at index 3 = " << Num[3] << endl;
    cout << "Element at index 4 = " << Num[4] << endl << endl;

    return 0;
}
```

```
Using .at(i)
Element at index 0 = 3
Element at index 1 = 4
Element at index 2 = 9
Element at index 3 = 1
Element at index 4 = 6

Using [i]
Element at index 0 = 3
Element at index 1 = 4
Element at index 2 = 9
Element at index 3 = 1
Element at index 4 = 6
```



# Lesson #47 - Change Elements

## تغيير عناصر الـ **Vector** في لغة C++

طرق تغيير عناصر الـ **Vector**: الدالة **at()** و عامل الفهرسة **[ ]**

الدالة **at()**

```
vector<int>vNum = { 1,2,3,4,5 };
cout << vNum.at(4) << endl;
// الوصول الى الفهرس 4 وطباعة قيمته (5)

vNum.at(4) = 30;
// تغيير القيمة (5) للفهرس 4 بالقيمة (30)

cout << vNum.at(4) << endl;
// الوصول الى الفهرس 4 وطباعة قيمته (30)
```

عامل الفهرسة **[ ]**:

```
vector<int>vNum = { 1,2,3,4,5 };
cout << vNum[4] << endl;
// الوصول الى الفهرس 4 وطباعة قيمته (5)

vNum[4] = 30;
// تغيير القيمة (5) للفهرس 4 بالقيمة (30)

cout << vNum[4] << endl;
// الوصول الى الفهرس 4 وطباعة قيمته (30)
```

## مثال توضيحي:

يمنع تغيير قيمة المتغير (i) داخل حلقة التكرار.

الرمز (i) يربط متغير حلقة التكرار (i) بالمجموعة التي نريد تكرارها (vNum).

```

int main()
{
    vector<int> vNum = { 1,2,3,4,5 };
    cout << "\n\tinitial vector : ";
        Ranged for loop // استعمال const يعمل lock لقيم العناصر (i) في ال Vector مع استعمال ال Ref & Ref دائماً مع ال
    for (const int &i : vNum)
    {
        cout << i << " ";
    }
    cout << endl;
    cout << "\n\tUpdated Vector : ";
    for (int &i : vNum)
    {
        i = 20;
        cout << i << " ";
    }
    vNum.at(1) = 23;
        تغيير كل قيم العناصر (i) بالقيمة 20 // at() بالدالة
    vNum[2] = 56;
        تغيير قيمة الفهرس 1 بالقيمة 23 بعامل الفهرسة []
    vNum.at(3) = 98;
        تغيير قيمة الفهرس 2 بالقيمة 56 بعامل الفهرسة []
    cout << endl;
    cout << "\n\tUpdated Vector : ";
        for loop // استعمال const يعمل lock لقيم العناصر (i) في ال Vector مع استعمال ال Ref & Ref دائماً مع ال
    for (const int &i : vNum)
    {
        cout << i << " ";
    }
    cout << endl;
    return 0;
}

```

```

Microsoft Visual Studio Debug Console

initial vector : 1 2 3 4 5
Updated Vector : 20 20 20 20 20
Updated Vector : 20 23 56 98 20

```



# Lesson #48 - Vector Iterators

## ما هي مكررات المتجه (Vector Iterators) في لغة ++C

### ما هي مكررات المتجه (Iterators):

مكررات المتجهات هي آلية تسمح لك بالوصول إلى عناصر المتجه والتنقل بينها الواحد تلو الآخر (الفة، لفة) وتعديلها بطريقة مرنة. على غرار المؤشرات في C++, يمكنهم الإشارة إلى موقع محدد في الذاكرة حيث يتم تخزين عنصر في المتجه. ولكن، على عكس المؤشرات، تتمتع مكررات المتجه بالعديد من المزايا التي يجعلها أكثر أماناً وفعالية في التعامل مع المتجه.

### كيف تعمل مكررات المتجه:

- كل متجه في C++ يوفر مجموعة خاصة به من المكررات.
- يشير المكرر إلى موضع عنصر معين في المتجه.
- يمكنك تحريك المكرر للأمام أو للخلف للوصول إلى عناصر أخرى في المتجه.
- لا تشير المكررات مباشرة إلى عناوين الذاكرة للمتجه، مما يجعلها أكثر أماناً من المؤشرات القياسية.

### ميزات استخدام مكررات المتجه:

- الأمان:** تمنع المكررات الوصول غير المصرح به إلى عناصر المتجه، كما تقلل فرص حدوث الأخطاء المتعلقة بالإدارة الخاطئة للذاكرة.
- الراحة:** توفر المكررات طريقة بسيطة وواضحة للتنقل بين عناصر المتجه، دون الحاجة إلى حساب عناوين الذاكرة يدوياً.
- الفعالية:** تستخدم المكررات عمليات خاصة محسنة للتنقل بين عناصر المتجه، مما يجعلها فعالة في الاستخدام.
- التكرار:** مكررات المتجه تتيح لك التكرار على جميع عناصر المتجه بطريقة منتظمة باستخدام حلقة **for**. مما يجعل التعامل مع البيانات أكثر مرونة.
- الوصول إلى العناصر:** يمكنك استخدام المكررات للوصول إلى أي عنصر في المتجه والعمل عليه مباشرةً.
- التعديل:** يمكنك تعديل قيمة عنصر في المتجه مباشرة باستخدام المكرر الخاص به. مما يوفر طريقة فعالة لتحديث البيانات.

### طرق استخدام مكررات المتجه:

- مكرر البداية (begin iterator):** يشير إلى العنصر الأول في المتجه.
- مكرر النهاية (end iterator):** يشير إلى العلامة بعد العنصر الأخير في المتجه (لا يشير إلى العنصر الأخير نفسه).
- مكرر قراءة فقط (const iterator):** يسمح لك بقراءة قيم عناصر المتجه ولكن لا يسمح بتعديلها.
- مكرر عكسي (reverse iterator):** يتيح لك تكرار عناصر المتجه بترتيب عكسي.

### باستخدام حلقة for:

يمكنك استخدام حلقة **for** مع المكررات للتنقل بين عناصر المتجه وتنفيذ عمليات على كل عنصر.

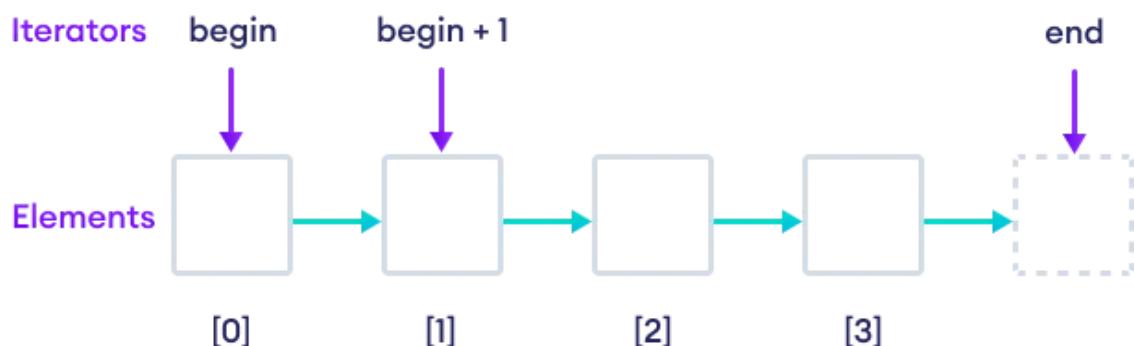
مثال توضيحي:

```
vector<int> vNum = { 1,2,3,4,5 };

// Declare iterator
vector<int>::iterator iNum;

// Use iterator with for loop
for (iNum = vNum.begin(); iNum < vNum.end(); iNum++)
{
    cout << *iNum << " ";
}
```

// تكرار جميع عناصر المتجه باستخدام مكرر البداية والنهاية



## Vector Iterators

### مقارنة مع المؤشرات في C++

- الأمان:** مكررات المتوجه أكثر أماناً من المؤشرات لأنها لا يمكن أن تشير إلى ذاكرة غير صالحة.
- سهولة الاستخدام:** مكررات المتوجه توفر واجهة أبسط للتنقل بين عناصر المتوجه مقارنة بالتعامل اليدوي مع الذاكرة باستخدام المؤشرات.
- القيود:** مكررات المتوجه لها بعض القيود مقارنة بالحرية الكاملة التي توفرها المؤشرات، حيث لا يمكن استخدامها على سبيل المثال للوصول إلى عناصر خارج نطاق المتوجه.

### ملخص:

مكررات المتوجهات هي أداة قوية للتعامل مع المتوجهات في لغة C++. فهم يوفرون طريقة آمنة وفعالة للوصول إلى عناصر المتوجه وتعديلها. نوصي باستخدام مكررات المتوجهات بدلاً من المؤشرات عند التعامل مع المتوجهات لتجنب أخطاء الذاكرة والحصول على كود أكثر قابلية للقراءة.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vNum = { 1,2,3,4,5 };

    // Declare iterator
    vector<int>::iterator iNum;

    cout << "\n\t";

    // Use iterator with for loop
    for (iNum = vNum.begin(); iNum < vNum.end(); iNum++)
    {
        cout << *iNum << " ";
    }

    cout << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

1 2 3 4 5



# Lesson #49 - Exceptions Handling: Try, Catch

❗ في حال كتبنا هذا الكود

( حيث أنشأنا متغير vector من نوع int يحمل 5 عناصر وعندما اردنا طباعة العنصر السادس اعطانا هذا ال Exception )

```
int main()
{
    vector<int> vNum = { 2,3,4,5,6 };

    cout << vNum.at(6);

    return 0;
}
```

سوف يظهر خطأ (Exception) كما بالصورة أسفل

```
2002 [[noreturn]] static void _Xrange() {
2003     _Xout_of_range("invalid vector subscript");
2004 }
2005 } x
2006
2007 #if __I
2008 C Exception Unhandled
2009 Unhandled exception at 0x00007FFE5E2CF19 in Test.exe:
2010 Microsoft C++ exception: std::out_of_range at memory location
2011 Show Call Stack | Copy Details | Start Live Share session...
2012 ⌂ Exception Settings
2013
2014     _Pnext = &_Temp->_Mynextiter;
2015 } else { // orphan the iterator
VSO-1269037
```

## مفهوم معالجة الأخطاء او معالجة الاستثناءات (Exceptions Handling)

❗ إندر: لا يستخدم إلا في الضرورة القصوى حيث هذه الطريقة تؤدي إلى بطء البرنامج

ما هي الاستثناءات؟

الاستثناءات هي مشاكل غير متوقعة تحدث أثناء تنفيذ البرنامج.

معالجة الأخطاء ( Exceptions Handling ) يقصد منها كتابة الكود الذي قد يسبب أي مشكلة في البرنامج بطريقة تضمن أنه إذا حدث الخطأ المتوقع أو أي خطأ آخر فإن البرنامج لن يعلق أو يتم إغلاقه بشكل فجائي.

❗ ظهور خطأ في البرنامج بشكل مفاجئ هو أمر سيئ جداً لأنه يؤدي إلى نفور عدد كبير من المستخدمين وعدم رغبتهم في العودة إلى استخدام هذا البرنامج مجدداً.

أنواع الأخطاء:

- أخطاء لغوية ( Syntax Errors ) ويقصد بها أن تخالف مبادئ اللغة مثل أن تعرف شيء بطريقة خاطئة أو تنسى وضع فاصلة منقوطة.
- أخطاء تحدث أثناء تشغيل البرنامج يقال لها إستثناءات ( Exceptions ) مما يؤدي إلى تعليقه وإيقافه بشكل غير طبيعي.
- أخطاء منطقية ( Logical Errors ) ويقصد منها أن الكود يعمل بدون أي مشاكل ولكن نتيجة تشغيل هذا الكود غير صحيحة.

❗ إذًا، أي خطأ برمجي يحدث معك أثناء تشغيل البرنامج يقال له إستثناء ( Exception ) حتى إن كان إسم الخطأ يحتوي على كلمة Error . بمعنى آخر، أي Error يظهر لك أثناء تشغيل البرنامج يعتبر Exception .

## بعض الأسباب التي تسبب حدوث إستثناء:

- في حال كان البرنامج يتصل بالشبكة وفجأة إنقطع الإتصال.
- في حال كان البرنامج يحاول قراءة معلومات من ملف نصي، وكان هذا الملف غير موجود.
- في حال كان البرنامج يحاول إنشاء أو حذف ملف ولكنه لا يملك صلاحية لفعل ذلك.
- في حال كانت عملية حسابية وكان من ضمن العمليات القسمة على صفر.
- في حال الوصول إلى ذاكرة غير صالحة.
- في حال الوصول إلى عنصر غير موجود في مجموعة بيانات.
- في حال حدوث أخطاء منطقية في الكود.
- في حال حدوث خطأ في عملية قراءة/كتابة لملف

## يمكن أن تشمل معالجة الاستثناء ما يلي:

- طباعة رسالة خطأ للمستخدم.
- تسجيل معلومات الخطأ في ملف.
- إنهاء البرنامج بشكل آمن.

## ؟ معلومة تقنية:

في حال كان الكود الذي كتبته يحتوي على أخطاء لغوية ( Syntax Errors ) لا بد من أن تصلّحها كلها حتى يستطيع المترجم تحويل الكود الذي كتبته لكود يفهمه الحاسوب ومن ثم ينفذه لك. أي لا يمكنك حماية البرنامج من Exceptions موجودة في الكود نفسه بل يمكنك حمايته من المشاكل التي قد تحدث وقت عمل هذه الكود.

## ميزات معالجة الاستثناءات:

- تحسين قابلية قراءة الكود: تجعل معالجة الاستثناءات الكود أكثر قابلية للفهم من خلال فصل التعامل مع الأخطاء عن المنطق الرئيسي للبرنامج.
- تحسين قابلية صيانة الكود: تُسهل معالجة الاستثناءات العثور على الأخطاء وإصلاحها.
- تحسين موثوقية البرنامج: تجعل معالجة الاستثناءات البرنامج أكثر موثوقية من خلال منع تعطل البرنامج عند حدوث أخطاء.

## عيوب معالجة الاستثناءات:

- يمكن أن تؤدي إلى زيادة حجم الكود وتعقيده.
- يمكن أن تؤثر على أداء البرنامج.

## الجملتين catch و try

التقاط الإستثناء ( Exception Catching ) عبارة عن طريقة تسمح لك بحماية البرنامج من أي كود تشك بأنه قد يسبب أي خطأ و لتحقيق هذا الأمر نستخدم الجملتين **try** و **catch**.

### متى نستخدم catch / try ؟

نستخدم **catch / try** عندما نعتقد أن كودا معيناً قد يرمي استثناءً. الاستثناء هو نوع خاص من الخطأ الذي يتوقف فيه البرنامج عن التنفيذ بشكل طبيعي.

### طريقة عمل نموذج catch / try

- يتم تنفيذ التعليمات داخل كتلة **try**.
- إذا حدث استثناء، فإن البرنامج ينتقل إلى معالج **catch** المناسب.
- يتم تنفيذ تعليمات معالج **catch** لمعالجة الاستثناء.
- بعد تنفيذ معالج **catch** أو انتهاء كتلة **try** بدون رمي استثناء، يستمر البرنامج في التنفيذ بشكل طبيعي.

**بشكل عام**، أي كود مشكوك فيه يجب وضعه بداخل حدود الجملة **try**.

أي مشكلة تحدث في الجملة **try** يتم معالجتها في حدود الجملة **catch** الخاصة بها كالتالي.

```
try
{
    // Protected Code
    // هنا نكتب الأوامر التي قد تسبب إستثناء
}
catch(ExceptionType e)
{
    // Error Handling Code
    // هنا نكتب أوامر تحدد للبرنامج ماذا يفعل إذا قامت الـ try برمي إستثناء
}
```

- الكود الذي نضعه بداخل الجملة **try** يسمى **Protected Code** وهذا يعني أن البرنامج محمي من أي خطأ قد يحدث بسبب هذا الكود.
- الكود الذي نضعه بداخل الجملة **catch** يسمى **Error Handling Code** ويقصد منها الكود الذي سيعالج الإستثناء الذي قد يتم التقاطه.
- عندما تستخدم الجملة **try** حتى لو لم تضع بداخلها أي كود، فأنتم مجبون على وضع الجملة **catch** بعدها.
- كما أنه بإمكانك وضع أكثر من جملة **catch** في حال كان الكود قد يسبب أكثر من خطأ.

### ملاحظات:

- يجب أن تكون كل كتلة **try** مصحوبة بكتلة **catch** واحدة على الأقل.
- يمكن استخدام معملات متعددة في كتلة **catch** للتقطاط أنواع مختلفة من الاستثناءات.
- يمكن رمي عدة أنواع مختلفة من الاستثناءات.
- يمكن كتابة عدة كتل **catch** لمعالجة أنواع مختلفة من الاستثناءات.
- يفضل استخدام معالجة الاستثناءات في الحالات غير المتوقعة التي يصعب توقعها مسبقاً.
- قد تؤثر معالجة الاستثناءات على أداء البرنامج.

### مثال توضيحي 1:

- في المثال التالي قمنا بإنشاء دالة إسمها **(divide)** عند استدعاءها نمرر لها رقمين فتقوم بإرجاع ناتج قسمة العدد الأول على العدد الثاني.
- هنا في حال كان الرقم الثاني الذي تم تمريره للدالة (المقسوم عليه) يساوي صفر سيعمل رمي إستثناء عبارة عن نص عادي مفاده بأنه في الرياضيات لا يمكن القسمة على صفر.
- بما أن الدالة **(divide)** قد تسبب حدوث خطأ عندما يتم استدعاؤها قمنا بوضعها بداخل **try / catch**

```

// هنا قمنا بتعريف دالة إسمها divide() عند استدعاءها نمرر لها عددين فتقوم بإرجاع ناتج قسمة العدد الأول على العدد الثاني
double divide(double a, double b)
{
    // في حال كان العدد الثاني الذي سيتم تمريره للباراميتر يساوي 0 سيتم رمي استثناء b
    if (b == 0)
    {
        cout << "\n\t";
        throw "Math Error, you can't divide by 0";
    }
    // إذا لم يتم رمي استثناء سيتم إرجاع ناتج القسمة
    return a / b;
}

int main()
{
    // هنا قمنا باستدعاء الدالة divide() وتمرير 0 مكان الباراميتر الثاني مما سيؤدي لرمي استثناء
    try
    {
        cout << divide(5, 0) << endl;
    }
    // الإستثناء الذي سيتم رميته سيكون عبارة عن نص (سلسلة من الأحرف) و هذه الأحرف سيتم تمريرها كقيمة للمتغير e
    catch (char const* e)
    {
        // هنا قمنا بطباعة نص الإستثناء الذي تم رميته و تخزينه في المتغير e
        cout << e << endl;
    }
    // هنا سيتم تنفيذ الأمر التالي بشكل عادي جداً لأن الإستثناء الذي حدث في السابق تم معالجته
    cout << "\n\tThe program is still working properly :)\n" << endl;
    return 0;
}

```

سنحصل على النتيجة التالية عند التشغيل:

Math Error, you can't divide by 0  
The program is still working properly :)

### معلومة مهمة

سبب جعل نوع الباراميتر e يكون **char const\*** بالتحديد هو أننا لاحظنا في المثال السابق أن النص الذي يتم رميه كاستثناء، يكون نوعه كذلك.

## معالجة أي نوع من الاستثناءات باستخدام (... ) في C++

في بعض الحالات، قد لا نعرف مسبقاً نوع الاستثناء الذي قد يثيره كتلة التعليمات داخل `try`. في هذه الحالة، لا يمكننا استخدام معالجات `catch` محددة لأنواع معينة من الاستثناءات. بدلاً من ذلك... نستخدم الرمز (... ) توفر لغة C++ ميزة تسمح باستقبال جميع أنواع الاستثناءات داخل كتلة `catch` التي قد تُرثى ضمن كتلة `try` لمعالجتها. وهذا ما يُعرف بـ "catch all" أو معالجة الاستثناءات الشاملة.

### استخدام (... ) مناسب في الحالات التالية:

- عند التعامل مع أخطاء غير متوقعة تماماً ولا يمكنك التحكم بها.
- عند كتابة كود معالجة عامة للتعافي من أي خطأ وضمان عدم انتهاء البرنامج بشكل مفاجئ.

### ملاحظات حول معالج الاستثناءات (... ) :

- استخدام (... ) قد يكون مفيداً في بعض الحالات، ولكن الأفضل تحديد أنواع الاستثناءات المتوقعة ومعالجتها بشكل منفصل لتحسين قابلية قراءة الكود والأداء.
- لاتستخدم (... ) بشكل عام لمعالجة جميع الاستثناءات دون معالجة محددة لتجنب إخفاء الأخطاء.

### مثال توضيحي 2:

```
int main()
{
    vector<int> vNum = { 1,2,3,4,5 };

    try
    {
        cout << vNum.at(5);

    }
    catch (...)
    {
        cout << "\n\tSorry! This element not found" << endl;
    }
    return 0;
}
```

سنحصل على النتيجة التالية عند التشغيل:

Sorry! This element not found



# Lesson #50 - String Object: (Common Methods)

## String Object: (Common Methods)

شرح بعض أساليب كائن السلسلة (string) في C++ مع أمثلة:

`str = string , ind = index , pos = position, len = length , Ch = char`

**:length() .1**

تعيد طول السلسلة (عدد الأحرف).

مثال:

```
std::string str = "مرحبا";
int length = str.length(); // يعطي 5
```

**:at(ind) .2**

تعيد حرفًا محدداً في الموضع pos داخل السلسلة.

مثال:

```
std::string str = "مرحبا";
char ch = str.at(2); // يعطي 'ر'
```

**:append(str) .3**

تضيف السلسلة الفرعية str إلى نهاية السلسلة الحالية.

مثال:

```
std::string str = "مرحبا";
str.append("العالم"); // تصبح "مرحبا بالعالم"
```

**:insert(pos, str) .4**

تُدخل السلسلة الفرعية str في الموضع pos داخل السلسلة.

مثال:

```
std::string str = "مرحبا";
str.insert(3, "بالعالم"); // تصبح "مرحبا بالعالم"
```

### **:substr(pos, len) .5**

تعيد جزءاً من السلسلة يبدأ من الموضع **pos** بطول **len**.

مثال:

```
std::string str = "مرحبا بالعالم";
std::string substring = str.substr(7, 6);           // يعطي "العالم"
```

### **:push\_back('Ch') .6**

تضيف حرفًا إلى نهاية السلسلة.

مثال:

```
std::string str = "مرحبا";
str.push_back('!');                                // تصبح "مرحبا!"
```

### **:pop\_back() .7**

تحذف آخر حرف من السلسلة.

مثال:

```
std::string str = "مرحبا";
str.pop_back();                                    // تصبح "مرحب"
```

### **: find(str) or if str.find(str == strnpos) .8**

تبحث عن السلسلة الفرعية **str** داخل السلسلة وتعيد موضعها الأول، أو **string::npos** إذا لم توجد.

مثال:

```
std::string str = "مرحبا بالعالم";
size_t pos = str.find("العالم");                  // يعطي 7
```

### **:clear() .9**

تفرغ السلسلة من جميع محتوياتها.

مثال:

```
std::string str = "مرحبا";
str.clear();                                       // تصبح ("سلسلة فارغة")
```

```

#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    string str1 = "I am Ahmad AbdelRahim, I learn C++";

    // 1- length : Prints the length of the string
    cout << "\n\tLength of str1 : " << str1.length() << endl;

    // 2- at() : Returns the letter at position 8
    cout << "\n\tReturns the letter at position 8 of str1 : "
        << str1.at(8) << endl;

    // 3- append : Adds another string to the end of string
    cout << "\n\tAdds another string to the end of str1 : "
        << str1.append(" In Programming Advices") << endl;

    // 4- insert : inserts "I am 53 years old, " at position 23
    cout << "\n\tInserts another string at position 23 of str1 : "
        << str1.insert(23, "I am 53 years old, ") << endl;

    // 5- substr : Prints all the next 23 letters from position 17.
    cout << "\n\tPrints from position 23 , length 17 letters of str1 : "
        << str1.substr(23, 17) << endl;

    // 6- push_back : Adds one character to the end of the string
    str1.push_back('!');
    cout << "\n\tAdds one character to the end of str1 : " << str1 << endl;

    // 7- pop_back : Removes one character from the end of the string
    str1.pop_back();

    cout << "\n\tRemoves one character from the end of str1 : " << str1 << endl;

    // 8- find : finds Ahmad in the string
    cout << "\n\tFinds Ahmad in str1 : " << str1.find("Ahmad") << endl;

    // 8- find : finds ahmad in the string
    cout << "\n\tFinds ahmad in str1 : " << str1.find("ahmad") << endl;

    if (str1.find("ahmad") == str1npos)
        cout << "\n\tThis word not found" << endl;

    // 9- clear : clears all string letters.
    str1.clear();
    cout << "\n\tClears all str letters" << str1 << endl;

    return 0;
}

```

Microsoft Visual Studio Debug Console

```

Length of str1 : 34
Returns the letter at position 8 of str1 : a
Adds another string to the end of str1 : I am Ahmad AbdelRahim, I learn C++ In Programming Advices
Inserts another string at position 23 of str1 : I am Ahmad AbdelRahim, I am 53 years old, I learn C++ In Programming Advices
Prints from position 23 , length 17 letters of str1 : I am 53 years old
Adds one character to the end of str1 : I am Ahmad AbdelRahim, I am 53 years old, I learn C++ In Programming Advices!
Removes one character from the end of str1 : I am Ahmad AbdelRahim, I am 53 years old, I learn C++ In Programming Advices
Finds Ahmad in str1 : 5
Finds ahmad in str1 : 18446744073709551615
This word not found
Clears all str letters

```

No.	Category	Functions and Operators	Functionality
1.	String Length	<b>length() or size()</b>	It will return the length of the string.
		طول السلسلة <b>size()</b> أو <b>length()</b>	تُرجع عدد الأحرف في السلسلة
		<code>string text = "مرحبا"; int length = text.length();</code>	// يعطي 5
2.	Accessing Characters	<b>Indexing (using array[index])</b>	To access individual characters using array indexing.
		فهرسة مثل المصفوفات <b>[]</b>	الوصول إلى حرف محدد بواسطة الفهرس (الرقم الترتيبى)
		<b>at()</b>	Used to access a character at a specified index.
3.	Appending and Concatenating Strings	الوصول إلى الأحرف <b>at(index)</b>	الوصول إلى حرف محدد بواسطة المؤشر (أكثر أماناً من الفهرسة)
		<code>string str = "مرحبا"; char ch = str[2]; // غير آمن، يفضل استخدام at() char ch = str.at(2);</code>	// يعطي "ر" //
		<b>+ Operator</b>	<b>+ operator</b> is used to concatenate two strings.
4.	String Comparison	<b>== Operator</b>	تلحق سلسلة بأخرى بحيث تصبح سلسلة واحدة.
		مشغل الجمع <b>==</b>	وتحرج <b>true</b> إذا تساوتا و <b>false</b> إذا لم يتتساوا
		<b>compare()</b>	The <b>compare()</b> function returns an integer value indicating the comparison result.
5.	Comparing Two Strings	تضييف سلسلة فرعية (str) إلى نهاية سلسلة أخرى. <b>append(str)</b>	تضييف سلسلة فرعية (str) إلى نهاية سلسلة أخرى.
		<code>string firstName = "أحمد"; string lastName = "خالد"; string fullName = firstName + " " + lastName; fullName.append(" (مطور برمجيات)");</code>	// باستخدام + // باستخدام append()
		<b>== Operator</b>	You can compare strings using the == operator.
6.	Comparing Two Strings	مقارنة سلسلتين <b>==</b>	تقارن السلسلتين
		<b>compare()</b>	وتحرج <b>true</b> إذا تساوتا و <b>false</b> إذا لم يتتساوا
		<b>compare(str)</b>	The <b>compare()</b> function returns an integer value indicating the comparison result.
7.	Comparing Two Strings	مقارنة سلسلتين وتحرج قيمة عددية: سالبة إذا كانت الأولى أكبر، موجبة إذا كانت أصغر، 0 إذا كانت متساويتين. <b>compare(str)</b>	تقارن السلسلتين وتحرج قيمة عددية: سالبة إذا كانت الأولى أكبر، موجبة إذا كانت أصغر، 0 إذا كانت متساويتين.
		<code>string str1 = "تفاحة"; string str2 = "موز"; if (str1 == str2) {     cout &lt;&lt; "السلسلتان متساويتان" &lt;&lt; endl; } else {     cout &lt;&lt; "السلسلتان غير متساويتان" &lt;&lt; endl; }</code>	// باستخدام ==
		<code>int result = str1.compare(str2); if (result == 0) {     cout &lt;&lt; "السلسلتان متساويتان" &lt;&lt; endl; } else if (result &lt; 0) {     cout &lt;&lt; "str1 تسبق str2" &lt;&lt; endl; } else {     cout &lt;&lt; "str1 تلي str2" &lt;&lt; endl; }</code>	// أو باستخدام compare()

	Substrings	<b>substr()</b>	Use the substr() function to extract a substring from a string.
5.	استخراج جزء من السلسلة	<b>substr(pos, len)</b>	تُخرج جزء من السلسلة يبدأ من الموضع pos بطول len.
	<code>string text = "مرحبا بالعالم"; string sub = text.substr(7, 5);</code>		// تعطي "العالم"
6.	Searching	<b>find()</b>	The find() function returns the position of the first occurrence of a substring.
	البحث عن نص داخل السلسلة	<b>find(str)</b>	تبعد عن سلسلة فرعية محددة وتُعيد موضعها الأول، أو string::npos إذا لم تُجد.
7.	<code>string searchIn = "لغة برمجة C++"; size_t position = searchIn.find("برمجة"); if (position != std::string::npos) {     cout &lt;&lt; "وجدت عند الموضع " &lt;&lt; position &lt;&lt; endl; } else {     cout &lt;&lt; "لم تُجد" &lt;&lt; std::endl; }</code>		
	Modifying Strings	<b>replace()</b>	Use the replace() function to modify a part of the string.
8.	تعديل جزء من السلسلة	<b>replace(str1, str2)</b>	تبدل جميـع تكرارات السلسلة str2 بسلسلة str1 الفرعية.
	<code>string text = ".أحب القطط"; text.replace(7, 4, "الكلاب");</code>		// استبدل "أحب القطط" بـ "أحب الكلاب"
7.	adds a substring at a specified position	<b>insert()</b>	The insert() function adds a substring at a specified position.
	إدراج نص فرعـي	<b>insert(pos, str)</b>	يُدرج سلسلة فرعية str في الموضع pos.
7.	<code>string text = ".أحب الكلاب"; text.insert(6, "الصغيرة");</code>		// يدرج "الصغيرة" في الموضع 6
	remove a part of the string	<b>erase()</b>	Use the erase() function to remove a part of the string.
8.	حـذف جـزء من السـلسلـة	<b>erase(pos, len)</b>	يـحـذـف جـزـء من السـلـسلـة يـبـدـأ من المـوضـع pos بـطـول len.
	<code>string text = ".أحب الكلاب الصغيرة"; text.erase(6, 8); // ".أحب الكلاب."</code>		
8.	Conversion	<b>c_str()</b>	To obtain a C-style string from a std::string, you can use the c_str() function.
	تحويل السلسلة إلى C-string	<b>c_str()</b>	ترجـع نسـخـة من السـلـسلـة كـ C-string (مصفـوفـة مـحتـويـاتـها نـصـية)
8.	<code>string str = "مرحبا"; const char* cstr = str.c_str(); // بـنـمـط "مرحـبا"</code>		



# Lesson #51 - Some CCTYPE Functions

(مكتبة ctype الخاصة بـ C++ و المكتبة ctype.h الخاصة بـ C) مع أمثلة:

مثلاً تحتوي مكتبة `string.h` على وظائف مدمجة لمعالجة السلسل النصية في لغة C/C++, فإن مكتبة `ctype.h` في C أو مكتبة `cctype` في C++ تحتوي على وظائف مدمجة للتعامل مع الأحرف الفردية - بمعنى آخر - مجموعة من الوظائف للتحقق من خصائص الأحرف وتغييرها.

## أنواع الأحرف:

أحرف قابلة للطباعة: هي الأحرف التي يتم عرضها على الشاشة (مثل الحروف والأرقام والرموز الشائعة).  
أحرف تحكم: هي أحرف خاصة - لا يتم عرضها على الشاشة - التي يتم استخدامها لإجراء عمليات محددة. ، ولكنها تؤثر على سلوك النص (مثل الضغط على زر التحكم (Ctrl) مع حرف آخر وعلامات التبويب والمسافات وسطور جديدة).

استخدامات (مكتبة ctype الخاصة بـ C++ و المكتبة ctype.h الخاصة بـ C) :

يجب تمرير قيم من نوع عدد صحيح (`integer`) إلى وظائف هذه المكتبة. إذا قمت بتمرير حرف بدلاً عن قيمته الصحيحة، يتم تحويل الحرف إلى قيمته الرقمية المقابلة (ASCII) قبل تمريرها إلى الوظيفة. تعمل الوظائف التالية الموجودة في (مكتبة ctype الخاصة بـ C++ و المكتبة ctype.h الخاصة بـ C) على الأحرف العادية، بينما توجد وظائف أخرى خاصة بالأحرف العريضة من نوع `wchar_t`.

هذه المكتبة مفيدة في العديد من التطبيقات، مثل:

- التحقق من صحة إدخال المستخدم.
- تحويل النص إلى حروف كبيرة أو صغيرة.
- البحث عن أنماط في النص.

## ملاحظة:

- هذه الوظائف تعمل فقط مع الأحرف ذات البايت الواحد من نوع `char`.
- تأكد من استخدام نوع البيانات المناسب (`char`) لتخزين الحرف.
- هذه الوظائف لا تعمل مع السلسل النصية من نوع `string`.
- يجب عليك استخدام وظائف فئة `string` للتعامل مع السلسل النصية من هذا النوع.
- تُعد مكتبة `cctype.h` مكتبة قديمة، ويفضل استخدام مكتبة `locale` في بعض الحالات.
- من المهم اختيار الوظائف المناسبة لاستخدامها مع نوع الحرف المناسب.

## مكتبات أخرى:

توفر مكتبة `cstring.h` وظائف أخرى للتعامل مع السلسل النصية.  
توفر مكتبة `wctype.h` وظائف مماثلة للتعامل مع الأحرف العريضة.

## جدول الوظائف:

رقم	الوظيفة	الوصف	قيمة الإرجاع
1	toupper()	تحول الحرف إلى حرف كبير (a to A)	حرف كبير مطابق للحرف الصغير الذي تم تمريره.
2	tolower()	تحول الحرف إلى حرف صغير (A to a)	حرف صغير مطابق للحرف الكبير الذي تم تمريره.
3	isupper()	تحقق من كون الحرف حرفًا كبيرًا (A-Z)	0 إذا لم يكن الحرف حرفًا كبيرًا، قيمة غير صفرية إذا كان كذلك. cout << "Is A Character is Uppercase (Yes '>= 1' / No '!= 0') ? : " << isupper('A') << endl; → 1 cout << "Is A Character is Uppercase (Yes '>= 1' / No '!= 0') ? : " << isupper('a') << endl; → 0
4	islower()	تحقق من كون الحرف حرفًا صغيرًا (a-z)	0 إذا لم يكن الحرف حرفًا صغيرًا، قيمة غير صفرية إذا كان كذلك. cout << "Is a Character is lowercase (Yes '>= 1' / No '!= 0') ? : " << islower('a') << endl; → 1 cout << "Is a Character is lowercase (Yes '>= 1' / No '!= 0') ? : " << islower('A') << endl; → 0
5	isdigit()	تحقق من كون الحرف رقمًا (9-0)	0 إذا لم يكن الحرف رقمًا، قيمة غير صفرية إذا كان كذلك. cout << "Is 8 Character is Digit (Yes '>= 1' / No '!= 0') ? : " << isdigit('8') << endl; → 1 cout << "Is a Character is Digit (Yes '>= 1' / No '!= 0') ? : " << isdigit('a') << endl; → 0
6	ispunct()	تحقق من كون الحرف علامة ترقيم	0 إذا لم يكن الحرف علامة ترقيم، قيمة غير صفرية إذا كان كذلك. cout << "Is & Char is punctuation (Yes '>= 1' / No '!= 0') ? : " << ispunct('&') << endl; → 1 cout << "Is a Char is punctuation (Yes '>= 1' / No '!= 0') ? : " << ispunct('a') << endl; → 0

## تابع جدول الوظائف:

رقم	الوظيفة	الوصف	قيمة الإرجاع
7	isalnum()	تحقق من كون الحرف حرفًا رقميًّا أو حرفًا (a-z, A-Z, 0-9)	0 إذا لم يكن الحرف حرفًا رقميًّا أو حرفًا، قيمة غير صفرية إذا كان كذلك.
8	isalpha()	تحقق من كون الحرف حرفًا (a-z, A-Z)	0 إذا لم يكن الحرف حرفًا، قيمة غير صفرية إذا كان كذلك.
9	isblank()	تحقق من كون الحرف مسافة بيضاء (مسافة أو علامة تبويب)	0 إذا لم يكن الحرف مسافة بيضاء، قيمة غير صفرية إذا كان كذلك.
10	iscntrl()	تحقق من كون الحرف حرف تحكم مثل (Ctrl+A)	0 إذا لم يكن الحرف حرف تحكم، قيمة غير صفرية إذا كان كذلك.
11	isgraph()	تحقق من كون الحرف قابلًا للطباعة (غير مسافة بيضاء أو حرف تحكم)	0 إذا لم يكن الحرف قابلًا للطباعة، قيمة غير صفرية إذا كان كذلك.
12	isprint()	تحقق من كون الحرف قابلًا للطباعة (غير حرف تحكم)	0 إذا لم يكن الحرف قابلًا للطباعة، قيمة غير صفرية إذا كان كذلك.
13	isspace()	تحقق من كون الحرف مسافة بيضاء (مسافة أو علامة تبويب أو سطر جديد)	0 إذا لم يكن الحرف مسافة بيضاء، قيمة غير صفرية إذا كان كذلك.

```
#include <iostream>
#include <vector>
#include <string>
#include <cctype>

using namespace std;

int main()
{
    char a, b;

    a = toupper('a');
    cout << "\n\tPrint a Character Uppercase : " << a << endl;

    b = tolower('B');
    cout << "\n\tPrint B Character Lowercase : " << b << endl;

    // Upper case characters (A to Z)
    cout << "\n\tIs A Character is Uppercase (Yes '>= 1' / No '= 0') ? : " << isupper('A') << endl;
    cout << "\n\tIs A Character is Uppercase (Yes '>= 1' / No '= 0') ? : " << isupper('a') << endl;

    // Lower case characters (a to z)
    cout << "\n\tIs a Character is lowercase (Yes '>= 1' / No '= 0') ? : " << islower('a') << endl;
    cout << "\n\tIs a Character is lowercase (Yes '>= 1' / No '= 0') ? : " << islower('A') << endl;

    // Digits (0123456789)
    cout << "\n\tIs 8 Character is Digit (Yes '>= 1' / No '= 0') ? : " << isdigit('8') << endl;
    cout << "\n\tIs a Character is Digit (Yes '>= 1' / No '= 0') ? : " << isdigit('a') << endl;

    // Punctuations Characters (~!@#$%^&*()_+=?><:{[]};'./')
    cout << "\n\tIs & Char is punctuation (Yes '>= 1' / No '= 0') ? : " << ispunct('&') << endl;
    cout << "\n\tIs a Char is punctuation (Yes '>= 1' / No '= 0') ? : " << ispunct('a') << endl;

    return 0;
}
```

```
Print a Character Uppercase : A
Print B Character Lowercase : b
Is A Character is Uppercase (Yes '>= 1' / No '= 0') ? : 1
Is A Character is Uppercase (Yes '>= 1' / No '= 0') ? : 0
Is a Character is lowercase (Yes '>= 1' / No '= 0') ? : 2
Is a Character is lowercase (Yes '>= 1' / No '= 0') ? : 0
Is 8 Character is Digit (Yes '>= 1' / No '= 0') ? : 4
Is a Character is Digit (Yes '>= 1' / No '= 0') ? : 0
Is & Char is punctuation (Yes '>= 1' / No '= 0') ? : 16
Is a Char is punctuation (Yes '>= 1' / No '= 0') ? : 0
```



# Lesson #52 - Write Mode: Write Data To File

قبل شرح التعامل مع الملفات من فتح ملف والكتابة فيه والإضافة إليه والقراءة منه يجب معرفة :-

## معالجة الملفات ( Files Handling ) في لغة C++

- تلعب **معالجة الملفات** دوراً مهماً في البرمجة بلغة C++, حيث تتيح لك إنشاء ملفات جديدة والقيام بعمليات كقراءة محتوى ملف وعرضه في البرنامج ، إنشاء نسخة منه، تعديل محتواه او حذفه سواء كانت الملفات نصوص او صور او اصوات او فيديوهات .

## كيف تستخدم معالجة الملفات في C++ ؟

- تُستخدم معالجة الملفات في C++ باستخدام مكتبة **fstream**. توفر هذه المكتبة مجموعة من الوظائف لقراءة وإضافة وكتابة البيانات من وإلى الملفات.

## مكتبة معالجة الملفات : fstream

- هي مكتبة مدمجة في لغة C++ توفر أنواعاً مختلفة من الكائنات لمعالجة الملفات أو مجموعة من الوظائف والخصائص التي تمكّنك من إنشاء ملفات جديدة، فتح ملفات موجودة، قراءة البيانات منها، وكتابة بيانات جديدة إليها، وإغلاقها بمجرد الانتهاء من العمل عليها.
- لا تُستخدم مباشرة لقراءة أو كتابة البيانات، بل تُستخدم لإنشاء كائنات تُستخدم بدورها لقراءة أو كتابة البيانات.
- للتتعامل مع الملفات يجب تضمين المكتبة **<fstream>** في رأس البرنامج باستخدام الأمر التالي:

```
#include <fstream>
```

- لفتح ملف باستخدام مكتبة **fstream**، تحتاج إلى إنشاء كائن من النوع المناسب (**ifstream** أو **ofstream**) وتعيين اسم الملف له.

## أنواع الكائنات التي توفرها مكتبة fstream :

توفر مكتبة **fstream** ثلاثة أنواع من الكائنات:

**ifstream** : يُستخدم لفتح الملفات للقراءة.

**ofstream** : يُستخدم لفتح الملفات للكتابة.

**fstream** : يُستخدم لفتح الملفات للقراءة والكتابة.

## الكائن : fstream

- هو كائن يُنشئ باستخدام مكتبة **fstream** لتمثيل ملف محدد.

- يتم إنشاؤه باستخدام اسم الملف ووضع الفتح (قراءة، كتابة، قراءة وكتابة).

- يُستخدم مباشرة لقراءة البيانات من الملف وكتابة البيانات في الملف.

- يحتوي على الوظائف التالية:

### وظيفة () open()

- عند استخدام الوظيفة () **open** مع كائن **fstream**، فإنّها تُستخدم لفتح ملف مرتبط بهذا الكائن.

### وظيفة () is\_open()

- هي وظيفة تُستخدم للتحقق من فتح الملف بنجاح.

- ترجع قيمة **true** إذا كان الملف مفتوحاً، وقيمة **false** إذا لم يكن مفتوحاً.

### قراءة البيانات من الملفات:

- () **getline()** لقراءة سطر واحد من الملف.

### كتابة البيانات إلى الملفات:

- operator >> لكتابة البيانات إلى الملف باستخدام مشغلي الإدخال/الإخراج.

### إغلاق الملفات:

- **close()** لإغلاق الملف.

## إنشاء كائن:

نقوم بإنشاء كائن من نوع `fstream` لتمثيل الارتباط بالملف الذي نرغب بالعمل عليه. على سبيل المثال:  
`fstream NewFile("data.txt");`  
هذا الكود يقوم بإنشاء كائن باسم `NewFile` مرتبط بالملف "data.txt".

## فتح الملف:

للقراءة من ملف أو إدخال بيانات فيه، تحتاج إلى فتحه أولاً. يمكن تحقيق ذلك باستخدام كائنات `ifstream` (للقراءة)، `ofstream` (للكتابة والإضافة)، أو `fstream` (للقراءة والكتابة)، أو `open()` مدمجة (للقراءة والكتابة). تحتوي جميع هذه الكائنات على دالة `open()` مدمجة فيها.

### بنية الدالة `open()`:

```
open(filename, mode);
```

حيث : **الParameters** :

**FileName**: اسم الملف الذي تريد فتحه.

**Mode**: وضع فتح الملف (مثل `"ios::in"` للقراءة أو `"ios::out"` للكتابة). على سبيل المثال:  
`NewFile.open("data.txt", ios::out); // فتح الكتابة`

## أوضاع مختلفة لفتح الملفات لتلبية احتياجاتك المتنوعة عند التعامل مع البيانات:

- **ios::in** (وضع القراءة): يفتح الملف للقيام بعمليات القراءة فقط. يمنع هذا الوضع إجراء أي تعديلات على محتويات الملف.
- **ios::out** (وضع الكتابة): يفتح الملف للكتابة. سيتم الكتابة فوق أي بيانات موجودة. استخدم هذا الوضع بحذر لأنه قد يؤدي إلى فقدان البيانات.
- **ios::app** (وضع الإلحاد): يفتح الملف للكتابة، ولكن تتم إضافة البيانات الجديدة إلى نهاية المحتوى الموجود. يحافظ هذا الوضع على البيانات الأصلية للملف.
- **ios::ate** (وضع الإلحاد): مشابه لـ `ios::app`، ولكن المؤشر يوضع في النهاية في البداية. لا تزال عمليات القراءة غير مسموحة.
- **ios::binary** (الوضع الثنائي): يعامل الملف على أنه سلسلة من البيانات، متجاهلاً ترميز النص القياسي. هذا الوضع ضروري للعمل مع البيانات غير النصية مثل الصور أو الملفات الثنائية.
- **ios::trunc** (وضع التخفيض): مشابه لـ `ios::out`، ولكن إذا كان الملف موجوداً، يتم مسح محتوياته قبل الكتابة.
- **ios::nocreate**: يفتح الملف فقط إذا كان موجوداً بالفعل. ستؤدي محاولة فتح ملف غير موجود إلى خطأ.
- **ios::noreplace**: يفتح الملف فقط إذا لم يكن موجوداً. ستؤدي محاولة فتح ملف موجود إلى خطأ.

## دمج الأوضاع لمزيد من المرونة:

- تتيح لك C++ دمج أوضاع متعددة باستخدام عامل الـ `bitwise OR` (" | ") لإنشاء سلوكيات فتح أكثر تخصيصاً:
- **ios::in | ios::out** (وضع القراءة والكتابة): يفتح الملف للقراءة والكتابة في نفس الوقت. يجب استخدام هذا الوضع بحذر شديد، حيث يمكن أن تؤدي عمليات الكتابة العرضية إلى إتلاف البيانات.
- **ios::app | ios::ate** (الإلحاد في النهاية والقراءة): يفتح الملف لإضافة البيانات ويسمح بالقراءة من بداية الملف.

### :fstream عيوب استخدام

قد تكون بطيئة عند التعامل مع الملفات الكبيرة.  
لا تدعم بعض ميزات أنظمة التشغيل المتقدمة.

### :fstream مزايا استخدام

سهولة الاستخدام.  
توفر وظائف قوية لقراءة وكتابة البيانات.  
تدعم مختلف أنواع البيانات.  
تدعم فتح الملفات في أوضاع مختلفة.

### مكتبات معالجة الملفات:

يعتمد اختيار مكتبة معالجة الملفات على احتياجاتك.  
**<fstream>** : مشتق من جملة **File Stream** ، توفر وظائف لفتح الملفات وقراءتها وكتابتها، مناسبة للمبتدئين للمهام البسيطة. توفر ثلاثة أنواع من الكائنات: **ifstream** و **ofstream** و **fstream**.

**<iostream>** : مشتق من جملة **Input Output Stream** ، توفر وظائف لعمليات الإدخال والإخراج، مناسبة للمبتدئين للمهام البسيطة. توفر وظائف مثل **cin** و **cout** و **cerr**.

**<stdio.h>** : مكتبة من C توفر وظائف عمليات الإدخال والإخراج، مناسبة للمطورين ذوي الخبرة للمهام المتقدمة.  
تتوفر وظائف مثل **fopen()** و **fclose()** و **scanf()** و **printf()**.

تميز **stdio.h** بوجودها في جميع بيئات C، لكنها قد تكون أكثر تعقيداً من **fstream** و **iostream**.

**Boost.IO** : مكتبة خارجية توفر وظائف متقدمة لقراءة وكتابة البيانات في **C++** ، مناسبة للمستخدمين ذوي الخبرة للمهام المتقدمة التي تتطلب وظائف غير موجودة في المكتبات الأخرى.  
تتوفر **Boost.IO** فئات مثل **ios::streambuf** و **ios::ofstream** و **ios::ifstream**.

### ( وضع الكتابة ) Write Mode: Write Data To File ios::out

- وضع الكتابة **ios::out** هو أحد أوضاع فتح الملفات في مكتبة **fstream** في **C++**.
  - يُستخدم هذا الوضع لفتح ملف جديد للكتابة فيه إذا لم يكن موجوداً. إذا كان الملف موجوداً بالفعل، فسيتم حذف أي محتوى موجود في الملف قبل كتابة أي بيانات جديدة.
  - إغلاق الملف بعد الانتهاء من الكتابة.
- استخدم هذا الوضع بحذر لأنه قد يؤدي إلى فقدان البيانات.

### - خطوات إنشاء وفتح ملف بالترتيب التالي:

- يتم تضمين مكتبي **iostream** و **fstream** اللذتين للتعامل مع الإدخال والإخراج والملفات.
- ينشأ كائن من نوع **fstream** ويُسمى **NewFile**.
- تستخدم دالة **open()** لفتح ملف جديد يُسمى **data.txt** في وضع الكتابة (**ios::out**).
- لمعرفة ما إن كان يمكنك التعامل مع الملف أم لا ، نستخدم الجمل الشرطية **if** و **else** بكل سهولة كالتالي:
  - A. نستخدم شرط **if** بداخله اسم الملف متبع بالدالة: (**NewFile.open()**) للتأكد من نجاح إنشاء الملف.
  - B. نكتب النص المراد إضافته إلى الملف باستخدام عامل الإدراج (**>>**).
- إغلاق الملف باستخدام الدالة (**NewFile.close()**).
- ثم **else** إذا لم يتم إنشاء الملف ويتم فيها عرض رسالة خطأ.

مثال توضيحي:-

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    string filename = "data.txt";

    fstream NewFile;

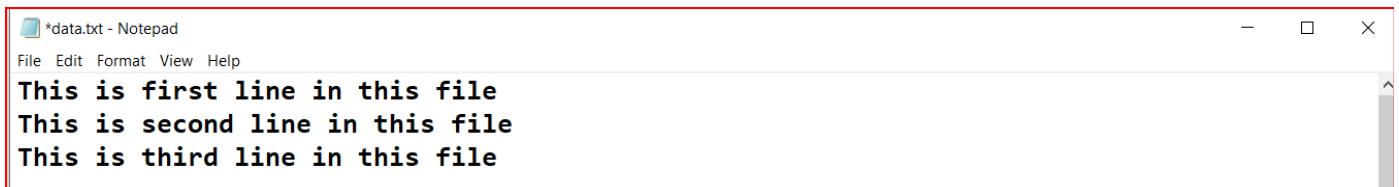
    NewFile.open(filename, ios::out);

    if (NewFile.is_open())
    {
        cout << "File Will Open" << endl;

        NewFile << "This is first line in this file" << endl;
        NewFile << "This is second line in this file" << endl;
        NewFile << "This is third line in this file" << endl;

        NewFile.close();
    }
    else
    {
        NewFile << "Error creating file: " << filename << endl;
    }

    return 0;
}
```

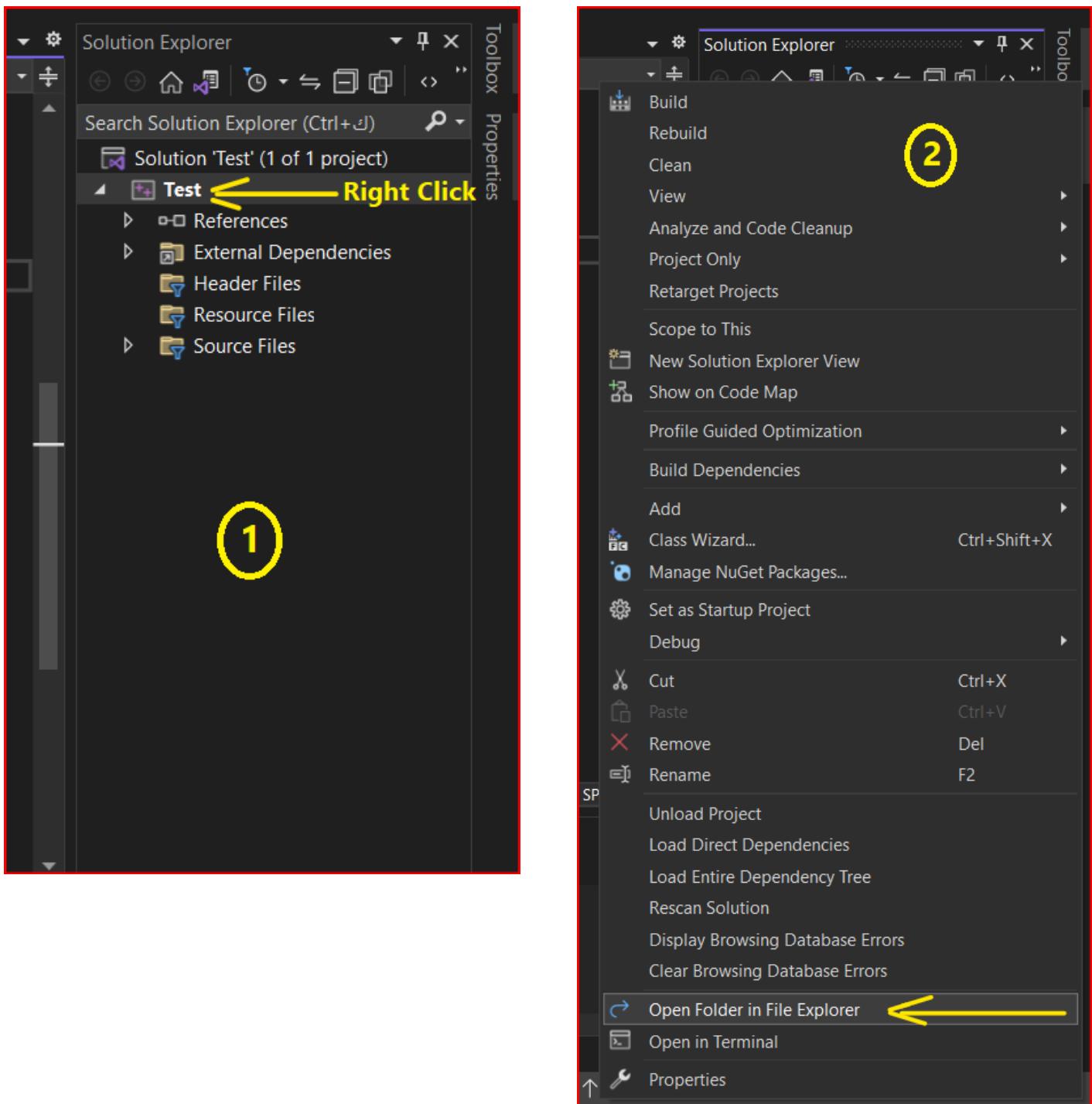


### أهمية إغلاق الملف عند الإنتهاء منه !

عند الإنتهاء من التعامل مع أي ملف، قم بإغلاقه على الفور لأن ذلك من شأنه تحسين أداء البرنامج حيث سيختفف من حجم المساحة المحفوظة للملف في الذاكرة بالإضافة إلى أنه تصبح قادر على التعامل مع هذا الملف بشكل مباشر من خارج برنامتك.

الكلاسات الثلاثة **ifstream** و **ofstream** و **fstream** جميعها تحتوي على دالة إسمها **close()** نستخدمها لإغلاق الملف.

**إذاً** لإغلاق الإتصال مع أي ملف مفتوح، يجب أن تستدعي الدالة **(close())** من الكائن الذي بالأساس فتحت الملف من خلاله.





# Lesson #53 - Append Mode: Append Data to File

: (Append Mode: Append Data To File) **ios::app**  
تم إضافة البيانات الجديدة إلى نهاية المحتوى الموجود. يحافظ هذا الوضع على البيانات الأصلية لملف.

خطوات إلحاد أو إضافة البيانات إلى ملف بالترتيب التالي:

- يتم تضمين مكتبي **fstream** و **iostream** اللذين للتعامل مع الإدخال والإخراج والملفات.
- ينشأ كائن من نوع **fstream** و يُسمى **NewFile**.
- نستخدم دالة **open()** لفتح الملف **data.txt** مع وضع **.ios::app**.
- يضمن هذا الوضع إضافة البيانات الجديدة إلى نهاية الملف دون التأثير على المحتوى الموجود.
- لمعرفة ما إن كان يمكنك التعامل مع الملف أم لا، نستخدم الجمل الشرطية **if** و **else** بكل سهولة كالتالي:
  - A. نستخدم شرط **if** بداخله اسم الملف متبع بالدالة: (**NewFile.is\_open()**) للتأكد من نجاح فتح الملف.
  - B. نكتب النص المراد إضافته إلى الملف باستخدام عامل الإدراج (**>>**).
  - C. إغلاق الملف باستخدام الدالة **NewFile.close()**.
  - D. ثم **else** إذا لم يتم إنشاء الملف ويتم فيها عرض رسالة خطأ.
- **من الأفضل** دمج الوضعين **ios::out** و **ios::app** باستخدام العامل **أو ( | )**.

مثال توضيحي:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    fstream NewFile;

    NewFile.open("Data.txt", ios::out | ios::app);

    if (NewFile.is_open())
    {
        NewFile << "This is a new line\n";
        NewFile << "This is another new line" << endl;

        NewFile.close();
    }

    else
    {
        cout << "Error opening file" << endl;
    }

    return 0;
}
```

## كيف يعمل وضع الإلحاد أو وضع الإضافة (ios::app)؟

- عند فتح ملف في هذا الوضع (ios::app) يتم وضع مؤشر الملف في نهاية المحتوى الموجود.
- أي بيانات جديدة يتم كتابتها إلى الملف ستضاف بعد هذا المؤشر - أي - يتم إضافتها بعد أي بيانات موجودة.
- لا يتم حذف أو استبدال أي بيانات موجودة.

### فوائد استخدام وضع الإلحاد:

- سهولة إضافة بيانات جديدة:** لا تحتاج إلى إعادة كتابة المحتوى الموجود لإضافة بيانات جديدة.
- كفاءة عالية:** لا يتم مسح المحتوى الموجود عند إضافة بيانات جديدة، مما يوفر وقتاً ومساحة.

### أمثلة لاستخدامات المتعددة لوضع الإلحاد:

- كتابة سجلات جديدة إلى ملف سجل.
- تحديث معلومات المستخدم في ملف قاعدة البيانات.
- دمج ملفات متعددة في ملف واحد.

### مزايا استخدام وضع الإضافة:

- سهل الاستخدام.
- يسهل إضافة بيانات جديدة إلى الملف دون الحاجة إلى إعادة كتابته بالكامل.
- مثالي لتسجيل البيانات أو إنشاء سجلات.
- لا يتطلب إعادة كتابة المحتوى الموجود.

### عيوب استخدام وضع الإضافة:

- لا يمكنك استخدام وضع الإضافة لقراءة البيانات من الملف.
- قد يؤدي استخدام وضع الإضافة إلى زيادة حجم الملف بشكل كبير.
- لا يمكنك إضافة البيانات في أي مكان داخل الملف.
- لا يمكنك حذف البيانات من الملف.

### ملاحظات مهمة:

- تأكد من أن الملف الذي تريد إلحاد البيانات إليه موجود بالفعل.
- لا يمكنك استخدام وضع الإضافة لقراءة البيانات من الملف.
- يمكنك إضافة أي نوع من البيانات إلى الملف، مثل الأعداد الصحيحة أو الحروف أو السلسل النصية.
- تأكد من إغلاق الملف بعد الانتهاء من الكتابة إليه.
- يمكن دمج أوضاع متعددة باستخدام عامل **أو (||)**.



# Lesson #54 - Read Mode: Print File Content

: (وضع القراءة) (Read Mode: Print File Content) `ios::in`

وضع القراءة `ios::in` هو أحد أوضاع فتح الملفات في `fstream`، ويسمح بقراءة البيانات من الملف فقط باستخدام دالة `getline()` وطباعتها على الشاشة.

خطوات قراءة البيانات من ملف بالترتيب التالي:

- يتم تضمين المكتبات `iostream` و `fstream` او `string` اللازميين للتعامل مع الإدخال والإخراج والملفات والنصوص .
- إنشاء كائن يُسمى `NewFile` من الفئة `fstream` ، ويربط هذا الكائن بالملف المراد قراءته باستخدام دالة `open()`.
- نستخدم دالة `open()` لفتح الملف `data.txt` مع وضع الفتح هو `ios::in` للقراءة فقط.
- لمعروفة ما إن كان يمكن التعامل مع الملف أم لا ، نستخدم الجمل الشرطية `if` و `else` .
- نستخدم الشرط `if` متبع بالدالة: ( `NewFile.is_open()` ) للتأكد من نجاح فتح الملف.
- إنشاء متغير `line` من نوع `string` لتخزين السطر المقتروء فيه.
- نستخدم الحلقة التكرارية `while` متبوعة بالدالة `getline(NewFile, line)` التي تحتوي على مصدر إدخال هو الملف (`NewFile`) لقراءة سطر واحد منه وتخزينه في المتغير `line` .
- وداخل الحلقة `while` تقوم الدالة `cout` مع عامل الاستخراج `<<` بطباعة السطر الذي تم تخزينه في المتغير `line` على الشاشة.
- ستقوم الحلقة بالتكرار لعند انعدام الشرط ( وهو عدم وجود سطر آخر مخزن في المتغير ) ثم يتم إيقاف الحلقة `while` .
- إغلاق الملف باستخدام الدالة `NewFile.close()` .
- ثم `else` إذا لم يفتح الملف ويتم فيها عرض رسالة خطأ.

مثال توضيحي:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void PrintFileContents(string filename)
{
    fstream NewFile;
    NewFile.open(filename, ios::in); // Read Mode
    if (NewFile.is_open())
    {
        string line;
        while (getline(NewFile, line))
        {
            cout << line << endl;
        }

        NewFile.close();
    }
    else
    {
        cout << "Error opening file" << endl;
    }
}

int main()
{
    PrintFileContents("Data.txt");

    return 0;
}
```

## فوائد استخدام وضع القراءة:

- قراءة البيانات من الملف: يمكنك قراءة محتوى الملف بأكمله أو جزء منه بأي ترتيب تريده.
- معالجة البيانات: يمكنك معالجة البيانات التي تم قرائتها من الملف باستخدام خوارزميات C++.
- كفاءة عالية: يتم تحميل البيانات في الذاكرة فقط عند الحاجة، مما يوفر وقتاً ومساحة.
- استخدامات متعددة: يمكن استخدام وضع القراءة لعرض محتوى الملف، أو البحث عن بيانات محددة، أو استخراج البيانات، أو تحليل البيانات، أو معالجة البيانات بطرق مختلفة.

## ملاحظات مهمة:

- تأكد من أن الملف الذي تريده قراءته موجود بالفعل.
- لا يمكنك استخدام وضع القراءة لكتابية بيانات جديدة إلى الملف.
- تأكد من إغلاق الملف بعد الانتهاء من القراءة.

## أمثلة لاستخدام وضع القراءة:

- عرض محتوى ملف نصي.
- البحث عن معلومات محددة في ملف.
- استخراج وتحليل البيانات من ملف CSV.
- معالجة بيانات الملف باستخدام خوارزميات C++.
- قراءة أسماء المستخدمين من ملف تكوين.
- تحميل بيانات خريطة لعبة من ملف.
- معالجة بيانات إحصائية من ملف بيانات.

## ملاحظات مهمة في استخدام الدالة :getline()

- استخدام دالة `getline` لقراءة أي نوع من البيانات، مثل الأرقام أو الأحرف أو الكلمات.
- تقوم دالة `getline` بقراءة سطر واحد من الملف أو بقراءة جميع الأحرف من مصدر الإدخال حتى تصادف علامة سطر جديد (`\n`) أو نهاية الملف. ثم تقوم ب تخزين جميع الأحرف المقروءة في المتغير النصي المحدد.
- استخدام الحلقة `while` مع الدالة `getline` لقراءة جميع أسطر الملف. ستستمر الحلقة في قراءة الأسطر من الملف طالما كانت هناك أسطر متبقية.
- التأكد من أن المتغير كبير بما يكفي لتخزين السطر المقرؤ بالكامل.
- إذا كان السطر الذي تريده قراءته يحتوي على فراغات، فتأكد من استخدام نوع بيانات `.string`.

## عيوب دالة :getline

- لا تقرأ أكثر من سطر واحد في المرة.
- تتطلب معرفة حجم السطر مسبقاً.

```

#include <iostream>                                // Including the input/output stream library
#include <fstream>                                 // Including the file stream library
#include <string>                                  // Including the string handling library
using namespace std;
void displayFileContent(const string& filename) // Function to display the content of a file
{
    ifstream file(filename);                      // Open file with given filename for reading
    string line;                                   // Declare a string to store each line of text
    if (file.is_open()) {                          // Check if the file was successfully opened
        cout << "File content:" << endl;           // Displaying a message indicating file content
        while (getline(file, line))               // Read each line from the file
        {
            cout << line << endl;                 // Display each line of the file
        }
        file.close();                            // Close the file
    }
    else
    {
        cout << "Failed to open the file." << endl; // Display an error message if file opening failed
    }
}
int main() {

    displayFileContent("new_test.txt"); // Display content of "new_test.txt" before any modification
    cout << endl;
    ofstream outputFile;                // Declare an output file stream object
    outputFile.open("new_test.txt", ios::app); // Open the file in append mode
    displayFileContent("new_test.txt"); // Display content of "new_test.txt" after opening in append mode
    cout << endl;
    if (outputFile.is_open()) {          // Check if the file was successfully opened
        string newData;                // Declare a string to store new data entered by the user
        cout << "Enter the data to append: "; // Prompt the user to enter data
        getline(cin, newData);          // Read the new data from the user
        outputFile << newData << endl;    // Get user input for new data
        outputFile.close();             // Append the new data to the file
        cout << "Data appended successfully." << endl; // Write the new data to the file
        displayFileContent("new_test.txt"); // Close the file
        cout << endl;                  // Display a success message
    }
    else
    {
        cout << "Failed to open the file." << endl; // Display an error message if file opening failed
    }
    return 0;                                // Return 0 to indicate successful execution
}

```

File content:  
This is first line in this file

File content:  
This is first line in this file

Enter the data to append: This is second line in this file  
Data appended successfully.

File content:  
This is first line in this file  
This is second line in this file



# Lesson #55

## Load Data From File To Vector

### تحميل بيانات الـ File وتخزينها بـ Vector

تحميل بيانات من **File** وتخزينها إلى **Vector** باستخدام **fstream** مع مود القراءة **ios::in** ، ثم عرض البيانات على الشاشة باستخدام حلقة **ranged for loop**.

- نستطيع أن نكتب **data** على **file** عن طريق الـ **ios::out** (Write Mode)
- نستطيع أن نضيف أو نلحق **data** على **file** عن طريق الـ **ios::app** (Append Mode)
- نستطيع أن نقرأ **data** من **file** ونطبعها على الشاشة عن طريق الـ **ios::in** (Read Mode)
- لكن الـ **fstream** لا نستطيع من خلاله تغيير او حذف سجل محدد من الملف.

من خلال استغلالنا للمعلومات الموجودة لدينا من (**كتابة** على الـ **file** ومحو محتوياته، **والاضافة** على الـ **file** وأيضاً **القراءة** من الـ **file**) باستخدام هذه المعلومات يمكن ان نحل هذه المشكلة التي تمثل في تغيير او حذف سجل محدد من اي ملف.

عشان احل المشكلة دي نعمل:

نقرأ محتويات الـ **File** بدل ما نطبعهم على الشاشة... نعرف **Vector** .. ونعمل **Push\_back** لكل الـ **data lines** للـ **Vector** ... مرة واحدة بنعمل ده .. باقدر انا امشي على الـ **Vector** زي ما بدبي .. اعدل اللي بدبي ايه ، الغي اللي بدبي ايه ، وبس اخلص التعديل على الـ **Vector** اللي هو الوسيط .. بارجع امشي بـ **Loop** بالـ **write** .. باعمل **write** للـ **Direct File** كمان مرة .. كأني عملت **update** على الـ **File** ..

الهدف من هذا البرنامج.. باستخدام هذه الادوات افتح **File** ثم اقرأ محتوياته واضيفهم على الـ **Vector**... بيصير عندي الـ **Vector** معباً كامل بمحظيات الـ **File** ... خلينا نشوف كيف هذا الكود بينكتب..

المكونات:

للتعامل مع الإدخال والإخراج.	:	<b>iostream</b>
للتعامل مع الملفات.	:	<b>fstream</b>
للتعامل مع النصوص.	:	<b>string</b>
هيكل بيانات ديناميكي لتخزين البيانات المقروءة.	:	<b>vector</b>
وضع القراءة لفتح الملفات.	:	<b>ios::in</b>
الحلقة تستمر في قراءة الأسطر من الملف طالما كانت هناك أسطر متبقية.	:	<b>while</b>
استخدم حلقة <b>while</b> مع <b>getline</b> لقراءة كل سطر من الملف وتخزينه في متغير <b>line</b> .	:	<b>getline</b>
دالة لقراءة أي نوع من البيانات، مثل الكلمات أو الجمل.	:	<b>push_back(line)</b>
أضف السطر المقروء ( <b>line</b> ) إلى نهاية الـ <b>Vector</b> ( <b>vDialogContent</b> )	:	<b>Ranged for loop</b>
عرض البيانات على الشاشة: استخدم حلقة <b>for</b> على الـ <b>Vector</b> ( <b>vDialogContent</b> ) لعرض كل سطر على الشاشة باستخدام <b>cout</b>	:	

## خطوات تحميل البيانات من ملف الى المتوجه بالترتيب التالي:

- يتم تضمين المكتبات `iostream` او `string` او `vector` اللازميين للتعامل مع الإدخال والإخراج والملفات والنصوص والمتوجهات.
- في الـ `main()` : نعرف `Vector` باسم `vFileContent` من نوع `string` .. لانه بيتخزن فيه `Lines`.
- ثم ننشأ `Procedure` باسم `LoadDataFromFileToVector` يأخذ `2 Parameters` - هما ( اسم الملف الذي سوف نقرأ منه البيانات `filename` من نوع `string` و يكون الـ `vector` بـ `by-ref` .. لأن كل تعديل في الـ `Main` ( `vFileContent` ) داخلي الـ `vFileContent` ) `Vector` في الـ `Procedure` .  
داخلي الـ `Procedure` :
- إنشاء كائن يُسمى `NewFile` من الفئة `fstream`، وربط هذا الكائن بالملف المراد قراءته باستخدام دالة `open()`.
- نستخدم دالة `open()` لفتح الملف `data.txt` `data` في وضع القراءة `ios::in` .
- لمعرفة ما إن كان يمكن التعامل مع الملف أم لا ، نستخدم الجمل الشرطية `if` و `else` .
- نستخدم الشرط `if` متبع بالدالة: ( `NewFile.is_open()` ) للتأكد من نجاح فتح الملف.
- إنشاء متغير `Line` من نوع `string` لتخزين السطر المقتول فيه.
- نستخدم الحلقة التكرارية `while` متبوعة بالدالة `getline(NewFile, Line)` التي تحتوي على مصدر إدخال `(NewFile)` لقراءة كل سطر من الملف وتخزينه في متغير مؤقت `Line` من نوع `string` .
- وداخل الحلقة `while` نضيف بـ `push_back` كل `Line` نقرأه من الـ `File` على الـ `vFileContent` .
- ستقوم الحلقة بالتكرار لعند انعدام الشرط ( وهو عدم وجود سطر آخر مخزن في المتغير ) ثم يتم إيقاف الحلقة `.while` .
- إغلاق الملف باستخدام الدالة `NewFile.close()` بعد الانتهاء من القراءة لمنع حدوث مشاكل.
- في الـ `main` بمجرد تنفيذ الـ `function` صار الـ `vFileContent` `Vector` معبأ بالبيانات.
- نستخدم حلقة `for` التكرارية مع نطاق ( `ranged-for` ) تتضمن المتغير `Line` من نوع `string` الذي يمثل كل عنصر في الـ `vFileContent` `Vector` عند كل تكرر.
- المتغير `Line` في الـ `ranged for loop` تكون `by-ref` علشان السرعة ، وكمان ما يتتسخش واحد جديد كل مرة.
- داخلي الـ `Ranged For Loop` نطبع المتغير `Line` باستخدام `cout` لإظهار المحتوى على الشاشة.
- سيتم تكرار هذه العملية لكل عنصر في الـ `vFileContent` `Vector` .

مثال توضيحي:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

void LoadDataFromFileToVector(string filename, vector<string> &vFileContent)
{
    fstream NewFile;
    NewFile.open(filename, ios::in);
    if (NewFile.is_open())
    {
        string Line;
        while (getline(NewFile, Line))
        {
            vFileContent.push_back(Line);
        }
        NewFile.close();
    }
}

int main()
{
    vector <string> vFileContent;
    LoadDataFromFileToVector("New_Test.txt", vFileContent);
    for (string& Line : vFileContent)
    {
        cout << Line << endl;
    }
    return 0;
}
```

```
Microsoft Visual Studio Debug Console
This is first line in this file
This is second line in this file
Ahmad
Samah
Heba
Mahmoud
Mariam
Shahd
```



## Lesson #56

# Save Vector To File

### حفظ بيانات الـ **Vector** في الـ **File** - Save Vector To File

حفظ بيانات الـ **Vector** إلى الـ **File** باستخدام **fstream** مع مود الكتابة **ios::out** ، باستخدام حلقة التكرار **.ranged for loop**.

#### خطوات حفظ البيانات من المتوجه إلى الملف بالترتيب التالي:

- يتم تضمين المكتبات **iostream** و **fstream** او **vector** او **string** الالازمين للتعامل مع الإدخال والإخراج والملفات والنصوص والمتوجهات.
- في الـ **(main)** : نعرف **Vector** باسم **vDialogContent** من نوع **string** ونضع به قيم أولية **(initial values)**.
- ثم ننشأ **Procedure** باسم **SaveVectorToFile** يأخذ **2 Parameters** - هما ( اسم الملف **(filename)** الذي سوف ننشأه للكتابة فيه و الـ **vDialogContent Vector** ولا يأخذ الـ **by-ref** .. لأن نريد قراءة البيانات ونحفظه على الملف فقط - بمعنى - لن نعدل شئ داخل الغانكشن ).  
داخل الـ **Procedure** :
- إنشاء كائن يسمى **NewFile** من الفئة **fstream**، وربط هذا الكائن بالملف المراد قراءته باستخدام دالة **open()**.
- نستخدم دالة **open()** لفتح الملف **filename** في وضع الكتابة **ios::out** .
- الـ **ios::out** .. عند فتح ملف باستخدام **ios::out**، يتم مسح أي محتوى موجود مسبقاً في الملف. هذا يعني أن أي بيانات موجودة في الملف ستُفقد. لكي نتمكن من حفظ البيانات في ملف باستخدام **fstream**، يجب فتح الملف في وضع **ios::out**. إذا لم يكن الملف موجوداً، فسيتم إنشاؤه. إذا كان الملف موجوداً بالفعل، فسيتم مسح محتواه الحالي، ثم ينتظر البرنامج حتى يتم كتابة جميع البيانات الجديدة في الملف قبل المتابعة.
- لمعرفة ما إن كان التعامل مع الملف أم لا ، نستخدم الجمل الشرطية **if** و **else** .
- قبل البدء بكتابة البيانات في الملف، نستخدم الشرط **if** متبوع بالدالة: **( NewFile.is\_open() )** للتأكد من أن الملف قد تم فتحه بنجاح.
- نستخدم حلقة **for** للتنتقل في عناصر الـ **Vector** ، في كل تكرار يتم معالجة كل عنصر في المتوجه بشكل منفصل باستخدام متغير مؤقت **Line** من نوع **string** مع وضع **by-ref** مما يوفر سرعة في الوصول إلى البيانات.
- وضعنا شرط إذا كان السطر فارغاً، فهذا يعني أننا لا نريد تخزينه في الملف.
- يتم التحقق من وجود بيانات في العنصر الحالي. إذا كان العنصر يحتوي على بيانات، يتم كتابته في الملف باستخدام المتغير **line** .
- إغلاق الملف باستخدام الدالة **( NewFile.close() )** بعد الانتهاء من القراءة لمنع حدوث مشاكل.

مثال توضیحی:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

void SaveVectorToFile(string filename, vector<string> vFileContent)
{
    fstream NewFile;

    NewFile.open(filename, ios::out);

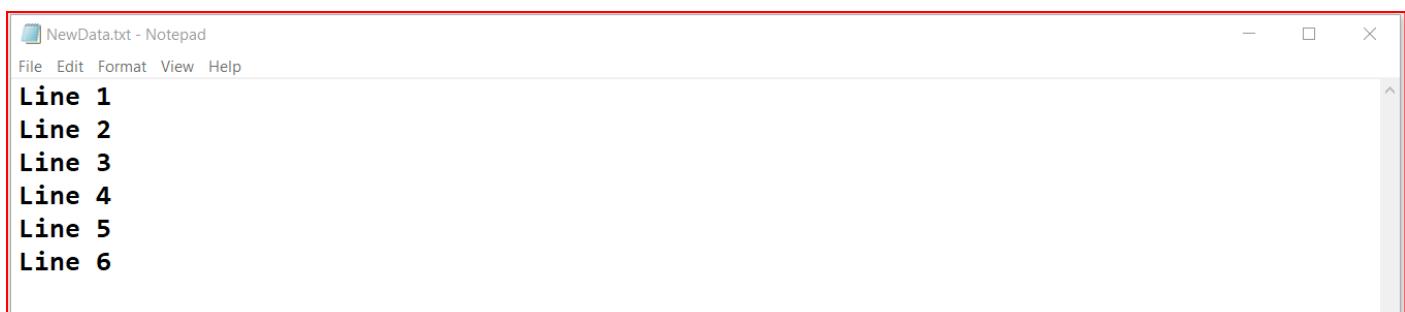
    if (NewFile.is_open())
    {
        for (string & line : vFileContent)
        {
            if (line != "")
            {
                NewFile << line << endl;
            }
        }

        NewFile.close();
    }
}

int main()
{
    vector<string> vFileContent = { "Line 1", "Line 2", "Line 3", "Line 4",
"Line 5", "Line 6" };

    SaveVectorToFile("NewData.txt", vFileContent);

    return 0;
}
```





## Lesson #57 :

# Delete Record From File

### **Delete Record From File**

لكي نتمكن من تعديل أو حذف سجل من ملف، نحتاج إلى وسيط مؤقت لتخزين البيانات من الملف ثم معالجتها وتحميلها مرة أخرى بالمحظى الجديد إلى الملف. ولأجل ذلك اخترنا المتوجه (**vector**) كوسيل لسهولة استخدامه وكفاءته.

- يتم تحميل محتويات الملف إلى **Vector** باستخدام حلقة **while** لقراءة كل سطر من الملف وإضافته إلى **Vector**.
- لا يوجد ما يمنع من البحث عن سجل معين في **Vector** وتعديلها. يمكن استخدام حلقة **for** للتنقل عبر عناصر **Vector** ومقارنة كل عنصر مع السجل الذي نبحث عنه.
- يمكن حفظ محتويات **Vector** بعد التعديل في الملف باستخدام حلقة **for** لكتابية كل عنصر من **Vector** في الملف.
- يتم استبدال محتوى الملف الحالي بالمحظى الجديد.

### **تضمين المكتبات المستخدمة:**

- **iostream** : تستخدم للتعامل مع الادخال والخروج، مثل (**cout**) طباعة النصوص على الشاشة و(**cin**) قراءة البيانات من المستخدم.
- **fstream** : تستخدم للتعامل مع الملفات (**المعالجة الملفات**)، مثل فتحها وقراءتها وكتابتها.
- **string** : تستخدم للتعامل مع النصوص وتخزينها.
- **vector** : تستخدم لتخزين وإدارةمجموعات من البيانات، مثل مجموعة من النصوص.

### **مساحة التسمية:**

• **using namespace std;**: تستخدم لتجنب الحاجة إلى إضافة **std::** قبل مكونات المكتبة القياسية.

### **المكونات الرئيسية:**

#### **الدوال:**

الدالة (**PrintFileContent(string filename)**) : تقرأ محتويات الملف وتطبعها على الشاشة.  
الدالة (**LoadDataFromFileToVector(string filename, vector<string>&vFileContent)**)  
تقرأ محتويات الملف وتحفظها في **vector** من نوع **string**.  
الدالة (**SaveVectorFromFile(string filename, vector<string>vFileContent)**)  
تحفظ محتويات **vector** من النصوص في ملف.  
الدالة (**DeleteRecordFromFile(string filename, string record)**)  
.SaveVectorToFile و **LoadDataFromFileToVector** تحذف سجلاً محدداً من الملف باستخدام دالة

## شرح تفصيلي للدوال الموجودة في الكود:

### الدالة الرئيسية () : main()

تطبع رسالة "File Before Delete Record" على الشاشة.

تستدعي الدالة PrintDialogContent لطباعة محتويات الملف NewFile.txt على الشاشة قبل حذف السجل.

تستدعي الدالة DeleteRecordFromFile لحذف السطر أو السجل الذي يحتوي على "Line6" من الملف NewFile.txt.

تطبع رسالة "nFile After Delete Record\" على الشاشة.

تستدعي الدالة PrintDialogContent لطباعة محتويات الملف NewFile.txt على الشاشة بعد حذف السجل.

### الدالة PrintDialogContent(string filename)

تقوم بطباعة كل سطر من الملف المحدد على الشاشة.

```
void PrintDialogContent(string filename)
{
    fstream Data1;
    Data1.open(filename, ios::in);

    if (Data1.is_open())
    {
        string line;

        while (getline(Data1, line))
        {
            cout << line << endl;
        }

        Data1.close();
    }
}
```

تأخذ اسم ملف كمدخل (filename).

تفتح ملف filename في وضع القراءة (ios::in) باستخدام fstream.

يتحقق ما إذا تم فتح الملف بنجاح.

إذا تم فتحه بنجاح، تستخدم حلقة while لقراءة سطراً تلو الآخر من الملف باستخدام getline.

يطبع كل سطر مقتول على الشاشة باستخدام cout.

تغلق الملف بعد الانتهاء.

**الدالة** `: LoadDataFromFileToVector(string filename, vector<string>&vFileContent)`  
تقرأ محتويات الملف وتحفظها في `vector` من نوع `string`

```
void LoadDataFromFileToVector(string filename, vector<string>&vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::in);

    if (Data1.is_open())
    {
        string line;
        while (getline(Data1, line))
        {
            vFileContent.push_back(line);
        }

        Data1.close();
    }
}
```

تأخذ اسم ملف (`filename`) ومتوجه من النصوص (`vFileContent`) كمدخل.  
تفتح ملف `filename` في وضع قراءة (`ios::in`) باستخدام `fstream`.

يتتحقق ما إذا تم فتح الملف بنجاح.

إذا تم فتحه بنجاح، تستخدم حلقة `while` لقراءة كل سطر من الملف باستخدام `getline` وتخزنه في المتوجه `vFileContent` باستخدام `push_back`.  
تغلق الملف بعد الانتهاء.

**الدالة** `: SaveVectorFromFile(string filename, vector<string>vFileContent)`  
تحفظ محتويات `vector` من النصوص في ملف.

```
void SaveVectorFromFile(string filename, vector<string>vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::out);

    if (Data1.is_open())
    {
        for (string &line : vFileContent)
        {
            if (line != "")
            {
                Data1 << line << endl;
            }
        }

        Data1.close();
    }
}
```

تأخذ اسم ملف (`filename`) ومتوجه من النصوص (`vFileContent`) كمدخل.  
تفتح ملف `filename` في وضع كتابة (`ios::out`) باستخدام `fstream`.

يتتحقق ما إذا تم فتح الملف بنجاح.

إذا تم فتحه بنجاح، تستخدم حلقة `for` للتكرار على كل سطر في `.vector vFileContent` للنقرار على كل سطر في `".`.

يتتحقق ما إذا كان السطر فارغاً ("").

إذا لم يكن السطر فارغاً، تكتب كل سطر من المتوجه إلى الملف باستخدام `<<`.  
تغلق الملف بعد الانتهاء.

## الدالة DeleteRecordFromFile(string filename, string record)

تحذف سجلاً محدداً من الملف باستخدام دالة `SaveVectorToFile` و `LoadDataFromFileToVector`

```
void DeleteRecordFromFile(string filename, string record)
{
    vector<string> vFileContent;

    LoadDataFromFileToVector(filename, vFileContent);

    for (string &line : vFileContent)
    {
        if (line == record)
        {
            line = "";
        }
    }

    SaveVectorFromFile(filename, vFileContent);
}
```

تأخذ اسم ملف (`filename`) والسجل المراد حذفه (`record`) كمدخل.

تنشئ `vector vFileContent` فارغاً لحفظ محتوى الملف مؤقتاً.

يستدعي الدالة `LoadDataFromFileToVector` لقراءة محتوى الملف وحفظه في `vector vFileContent`.

تستخدم حلقة `for` للتكرار على كل سطر (`line`) في `vector vFileContent` تبحث عن السجل المراد حذفه مع استخدام `(& " by-ref")`.

تحقق ما إذا كان السطر الحالي مطابقاً للسجل المراد حذفه (`record`).

إذا وجدت السجل، تزدفه عن طريق تعينه إلى سلسلة فارغة ("") مما يحذف السجل فعلياً.

يستدعي الدالة `SaveVectorToFile` لحفظ محتوى `vector` المعدل (الذي لا يحتوي على السجل المحدد) في الملف الأصلي.

## محتوى الملف قبل حذف السجل "Line6"



```
*NewFile.txt - Notepad
File Edit Format View Help
Line1
Line2
Line3
Line4
Line5
Line6
Line7
Line8
Ln 8, Col 6 100% Windows (CRLF) UTF-8
```

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

void PrintFileContent(string filename)
{
    fstream Data1;
    Data1.open(filename, ios::in);
    if (Data1.is_open()) {
        string line;
        while (getline(Data1, line))
        {
            cout << line << endl;
        }
        Data1.close();
    }
}
void LoadDataFromFileToVector(string filename, vector<string>& vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::in);
    if (Data1.is_open()) {
        string line;
        while (getline(Data1, line))
        {
            vFileContent.push_back(line);
        }
        Data1.close();
    }
}
void SaveVectorFromFile(string filename, vector<string>vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::out);
    if (Data1.is_open())
    {
        for (string& line : vFileContent)
        {
            if (line != "")
            {
                Data1 << line << endl;
            }
        }
        Data1.close();
    }
}
void DeleteRecordFromFile(string filename, string record)
{
    vector<string>vFileContent;
    LoadDataFromFileToVector(filename, vFileContent);
    for (string& line : vFileContent)
    {
        if (line == record)
        {
            line = "";
        }
    }
    SaveVectorFromFile(filename, vFileContent);
}
int main()
{
    cout << "File Before Delete Record : " << endl;
    PrintFileContent("NewFile.txt");
    DeleteRecordFromFile("NewFile.txt", "Line6");
    cout << "\nFile After Delete Record : " << endl;
    PrintFileContent("NewFile.txt");
}

```

## طباعة محتوى الملف قبل حذف السجل "Line6" وبعد حذف السجل "Line6"

```
Microsoft Visual Studio Debug Console
File Before Delete Record :
Line1
Line2
Line3
Line4
Line5
Line6
Line7
Line8

File After Delete Record :
Line1
Line2
Line3
Line4
Line5
Line7
Line8
```

## محتوى الملف بعد حذف السجل "Line6"

```
NewFile.txt - Notepad
File Edit Format View Help
Line1
Line2
Line3
Line4
Line5
Line7
Line8

Ln 1, Col 1 100% Windows (CRLF) UTF-8
```



# Lesson #58 :

## Update Record In File

### Update Record In File

هذا الكود يتعامل مع ملف نصي، إذ يقرأ محتوياته كاملة ويخزنها في متوجه. ثم يبحث عن السجل المراد تحريره ويقوم بتحديثه بالسجل الجديد. بعد ذلك، يحفظ محتويات المتوجه المعدلة مرة أخرى إلى الملف ويطبع محتويات الملف قبل وبعد التحديث.

#### تضمين المكتبات المستخدمة:

- **iostream** : تستخدم للتعامل مع الادخال والاخراج، مثل (**cout**) طباعة النصوص على الشاشة و(**cin**) قراءة البيانات من المستخدم.
- **fstream** : تستخدم للتعامل مع الملفات (**لمعالجة الملفات**)، مثل فتحها وقراءتها وكتابتها.
- **string** :
- **vector** :

#### مساحة التسمية:

تستخدم لتجنب الحاجة إلى إضافة **using namespace std;** قبل مكونات المكتبة القياسية.

#### المكونات الرئيسية:

##### الدوال:

الدالة (**PrintFileContent(string filename)**) : تقرأ محتويات الملف وتطبعها على الشاشة.  
الدالة (**LoadDataFromFileToVector(string filename, vector<string>&vFileContent)**) : تقرأ محتويات الملف وتحفظها في **vector<string>** من نوع **string**.  
الدالة (**SaveVectorToFile(string filename, vector<string>vFileContent)**) : تحفظ محتويات **vector<string>** من النصوص في ملف.  
الدالة (**UpdateRecordInFile(string filename, string record, string update)**) : تحدث سجلاً محدداً من الملف باستخدام دالة **SaveVectorToFile** و **LoadDataFromFileToVector**.

## شرح تفصيلي للدوال الموجودة في الكود:

الدالة الرئيسية : `main()`

تطبع رسالة "File Before Delete Record" على الشاشة.

تستدعي الدالة `PrintDialogContent` لطباعة محتويات الملف `NewFile.txt` على الشاشة قبل تحديد السجل.

تستدعي الدالة `UpdateRecordInFile` لتحديث السطر الذي يحتوي على "Line6" إلى القيمة الجديدة "LINE6".

تطبع رسالة "nFile After Delete Record\" على الشاشة.

تستدعي الدالة `PrintDialogContent` لطباعة محتويات الملف `NewFile.txt` على الشاشة بعد تحديد السجل.

الدالة : `PrintDialogContent(string filename)`

تقوم بطباعة كل سطر من الملف المحدد على الشاشة.

```
void PrintDialogContent(string filename)
{
    fstream Data1;
    Data1.open(filename, ios::in);

    if (Data1.is_open())
    {
        string line;

        while (getline(Data1, line))
        {
            cout << line << endl;
        }

        Data1.close();
    }
}
```

تأخذ اسم ملف كمدخل (`filename`).

تفتح ملف `filename` في وضع القراءة (`ios::in`) باستخدام `fstream`.

يتحقق ما إذا تم فتح الملف بنجاح.

إذا تم فتحه بنجاح، تستخدم حلقة `while` لقراءة سطراً تلو الآخر من الملف باستخدام `getline`.

يطبع كل سطر مقرؤه على الشاشة باستخدام `cout`.

تغلق الملف بعد الانتهاء.

**الدالة** : LoadDataFromFileToVector(string filename, vector<string>&vFileContent)  
تقرأ محتويات الملف وتحفظها في **vector** من نوع **string**

```
void LoadDataFromFileToVector(string filename, vector<string>&vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::in);

    if (Data1.is_open())
    {
        string line;
        while (getline(Data1, line))
        {
            vFileContent.push_back(line);
        }

        Data1.close();
    }
}
```

تأخذ اسم ملف (**filename**) ومتوجه من النصوص (**vFileContent**) كمدخل.  
تفتح ملف **filename** في وضع قراءة (**ios::in**) باستخدام **fstream**.  
يتتحقق ما إذا تم فتح الملف بنجاح.  
إذا تم فتحه بنجاح، تستخدم حلقة **while** لقراءة كل سطر من الملف باستخدام **getline** وتخزنه في المتوجه **vFileContent** باستخدام **push\_back**.  
تغلق الملف بعد الانتهاء.

**الدالة** : SaveVectorFromFile(string filename, vector<string>vFileContent)  
تحفظ محتويات **vector** من النصوص في ملف.

```
void SaveVectorFromFile(string filename, vector<string>vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::out);

    if (Data1.is_open())
    {
        for (string &line : vFileContent)
        {
            if (line != "")
            {
                Data1 << line << endl;
            }
        }

        Data1.close();
    }
}
```

تأخذ اسم ملف (**filename**) ومتوجه من النصوص (**vFileContent**) كمدخل.  
تفتح ملف **filename** في وضع كتابة (**ios::out**) باستخدام **fstream**.  
يتتحقق ما إذا تم فتح الملف بنجاح.  
إذا تم فتحه بنجاح، تستخدم حلقة **for** للتكرار على كل سطر في **.vector vFileContent** إلى الملف باستخدام **<<**.  
يتتحقق ما إذا كان السطر فارغاً ("").  
إذا لم يكن السطر فارغاً، تكتب كل سطر من المتوجه إلى الملف باستخدام **<<**.  
تغلق الملف بعد الانتهاء.

```

الدالة : UpdateRecordInFile(string filename, string record, string update)
تحدد سجلاً محدداً من الملف باستخدام دالة .SaveVectorToFile و LoadDataFromFileToVector

void UpdateRecordInFile(string filename, string record, string update)
{
    vector<string> vFileContent;

    LoadDataFromFileToVector(filename, vFileContent);

    for (string &line : vFileContent)
    {
        if (line == record)
        {
            line = update;
        }
    }

    SaveVectorFromFile(filename, vFileContent);
}

```

تأخذ اسم ملف (filename) والسجل المراد تحريره (record) والسجل الجديد (update) كمدخل.  
تنشئ الدالة `vector vFileContent` فارغاً لحفظ محتوى الملف مؤقتاً.

يستخدم حلقة `for` للتكرار على كل سطر (line) في `vector vFileContent` لقراءة محتوى الملف وحفظه في `.vector vFileContent` يسدي الدالة `LoadDataFromFileToVector` لقراءة محتوى الملف وحفظه في `vFileContent` تبحث عن السجل المراد حذفه مع استخدام ("&" by-ref).

تحقق ما إذا كان السطر الحالي مطابقاً للسجل المراد حذفه (record).  
إذا وجدت السجل، تقوم بتحديثه بتغيير قيمته إلى القيمة الجديدة المرسلة باستخدام update.  
يسدي الدالة `SaveVectorToFile` لحفظ محتوى `vector` المعدل في الملف الأصلي.

### محتوى الملف قبل تحديث السجل "Line6"



```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

void PrintFileContent(string filename)
{
    fstream Data1;
    Data1.open(filename, ios::in);
    if (Data1.is_open())
    {
        string line;
        while (getline(Data1, line))
        {
            cout << line << endl;
        }
        Data1.close();
    }
}

void LoadDataFromFileToVector(string filename, vector<string>& vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::in);
    if (Data1.is_open())
    {
        string line;
        while (getline(Data1, line))
        {
            vFileContent.push_back(line);
        }
        Data1.close();
    }
}

void SaveVectorFromFile(string filename, vector<string> vFileContent)
{
    fstream Data1;
    Data1.open(filename, ios::out);
    if (Data1.is_open())
    {
        for (string& line : vFileContent)
        {
            if (line != "")
            {
                Data1 << line << endl;
            }
        }
        Data1.close();
    }
}

void UpdateRecordInFile(string filename, string record, string update)
{
    vector<string> vFileContent;
    LoadDataFromFileToVector(filename, vFileContent);
    for (string& line : vFileContent)
    {
        if (line == record)
        {
            line = update;
        }
    }
    SaveVectorFromFile(filename, vFileContent);
}

int main()
{
    cout << "File Before Delete Record : " << endl;
    PrintFileContent("NewFile.txt");
    UpdateRecordInFile("NewFile.txt", "Line6", "LINE6");
    cout << "\nFile After Delete Record : " << endl;
    PrintFileContent("NewFile.txt"); }

```

طباعة محتوى الملف قبل تحديث السجل "LINE6" وبعد تحديث السجل "LINE6"

```
Microsoft Visual Studio Debug Console
File Before Delete Record :
Line1
Line2
Line3
Line4
Line5
Line6
Line7
Line8

File After Delete Record :
Line1
Line2
Line3
Line4
Line5
LINE6
Line7
Line8
```

محتوى الملف بعد تحديث السجل "LINE6"

```
NewFile.txt - Notepad
File Edit Format View Help
Line1
Line2
Line3
Line4
Line5
LINE6
Line7
Line8

Ln 1, Col 1 100% Windows (CRLF) UTF-8
```



# Lesson #59 :

## Date & Time: Local / UTC Time

### <ctime> (time.h)

#### Functions

##### Time manipulation

معالجة الوقت:

<b>clock()</b>	Clock Program (function)	تحصل على عدد وحدات الزمن التي انقضت منذ بدء تشغيل البرنامج.
<b>difftime()</b>	Return difference between two times (function)	تحسب الفرق بين نقطتين زمنيتين من نوع <code>time_t</code> وتعيده بالثواني.
<b>mktime()</b>	Convert tm structure to time_t (function)	تحول هيكل بيانات tm إلى نقطة زمنية من نوع <code>time_t</code>
<b>time()</b>	Get current time (function)	تحصل على الوقت الحالي كنقطة زمنية من نوع <code>time_t</code>

##### Conversion

التحويل:

<b>asctime()</b>	Convert tm structure to string (function)	تحول هيكل بيانات tm إلى سلسلة نصية قابلة للطباعة تمثل التاريخ والوقت بتنسيق محدد مسبقاً.
<b>ctime()</b>	Convert time_t value to string (function)	تحول نقطة زمنية من نوع <code>time_t</code> إلى سلسلة نصية قابلة للطباعة تمثل التاريخ والوقت بتنسيق محدد مسبقاً
<b>gmtime()</b>	Convert time_t to tm as UTC time (function)	تحول نقطة زمنية من نوع <code>time_t</code> إلى هيكل بيانات tm يمثل التاريخ والوقت بتوقيت UTC
<b>localtime()</b>	Convert time_t to tm as local time (function)	تحول نقطة زمنية من نوع <code>time_t</code> إلى هيكل بيانات tm يمثل التاريخ والوقت بالتوقيت المحلي.
<b>strftime()</b>	Format time as string (function) (str, maxsize, format, timeptr)	تنسيق التاريخ والوقت في سلسلة نصية وفقاً للتنسيق المحدد.

##### Macro constants

ثوابت الماكرو:

<b>CLOCKS_PER_SEC</b>	Clock ticks per second (macro)	يمثل عدد وحدات الزمن في الثانية الواحدة.
<b>NULL</b>	Null pointer (macro)	يمثل مؤشر فارغ

##### Types

الأنواع:

<b>clock_t</b>	Clock type (type)	يمثل نوع بيانات لوحدات الزمن التي يتم إرجاعها بواسطة دالة <code>clock</code>
<b>size_t</b>	Unsigned integral type (type)	يمثل نوع بيانات صحيح غير سالب.
<b>time_t</b>	Time type (type)	يمثل نوع بيانات لنقطة زمنية (timestamp)
<b>struct tm</b>	Time structure (type)	يمثل هيكل بيانات لتخزين معلومات التاريخ والوقت.

## دوال مكتبة time.h المستخدمة في هذا الكود :

- **time(0)** : تحصل على الوقت الحالي من نظام التشغيل.
- **ctime(&T)** : تحول الوقت الحالي إلى سلسلة نصية قابلة للطباعة بتنسيق محدد مسبقاً.
- **gmtime(&T)** : تحول الوقت الحالي إلى (UTC - هيكل بيانات) tm يمثل التاريخ والوقت بتوقيت UTC.
- **asctime(gmTD)** : تحول (UTC - هيكل بيانات) tm إلى سلسلة نصية قابلة للطباعة بتنسيق محدد مسبقاً.

شرح كود يظهر استخدام بعض وظائف مكتبة time.h في C++

```
#pragma warning(disable : 4996)

#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    time_t T = time(0);

    char* DT = ctime(&T);

    cout << "Local Time and Date Now is : " << DT << endl;

    tm* gmTime = gmtime(&T);

    DT = asctime(gmTime);

    cout << "UTC Time and Date now is : " << DT << endl;

    return 0;
}
```

## الشرح:

```
#pragma warning(disable : 4996)
```

شرح: يستخدم هذا السطر لإيقاف تحذير محدد من المترجم. في هذه الحالة، يتم إيقاف التحذير رقم 4996، والذي يتعلق بوظيفة **ctime**.

```
#include <iostream>
```

شرح: يقوم هذا السطر بتضمين مكتبة **iostream** في الكود التي تستخدم لعمليات الإدخال والإخراج، والذي يسمح باستخدام دالة **cout** لطباعة الرسائل.

```
#include <ctime>
```

شرح: يقوم هذا السطر بتضمين مكتبة **ctime** في الكود ، والذي يسمح باستخدام وظائف مكتبة **time.h** التي تحتوي على وظائف للتعامل مع الوقت والتاريخ.

```
int main()
{
```

#### وظائف معالجة الوقت:

نوع البيانات `time_t` : لتخزين الوقت الحالي.

```
    time_t T = time(0);
```

شرح: يقوم هذا السطر باستدعاء دالة `time(0)` التي تحصل على الوقت الحالي وتخزينه في متغير من نوع `time_t` يسمى `T`.

#### وظائف التحويل:

: `ctime(time_t)` الدالة

```
    char* TD = ctime(&T);
```

شرح: يقوم هذا السطر باستدعاء دالة `ctime(&T)` التي تحول الوقت المخزن في `T` إلى سلسلة نصية قابلة للطباعة وتخزينها في متغير `TD` من نوع `char*`.

```
    cout << "\n\tLocal Time and Date is : " << TD << endl;
```

شرح: يقوم هذا السطر بطباعة السلسلة النصية `TD` التي تحتوي على التاريخ والوقت المحلي باستخدام `cout`.

#### وظائف التحويل:

: `gmtime(time_t)` الدالة

```
    tm* gmTD = gmtime(&T);
```

شرح: يقوم هذا السطر باستدعاء دالة `gmtime(&T)` التي تحول الوقت المخزن في `T` إلى (`tm` - هيكل بيانات) يمثل التاريخ والوقت بتوقيت `UTC` وتخزينه في متغير `gmTD` من نوع `*tm`.

#### وظائف التحويل:

: `asctime(timeptr)` الدالة

```
    TD = asctime(gmTD);
```

شرح: يقوم هذا السطر باستدعاء دالة `asctime(gmTD)` التي تحول (`tm` - هيكل بيانات) إلى سلسلة نصية قابلة للطباعة وتخزينها في متغير `TD`.

تقوم الدالة بتحويل هذه المعلومات إلى سلسلة نصية قابلة للطباعة تتبع التنسيق التالي:

Www Mmm dd hh:mm:ss yyyy

حيث:

`Www`: يمثل اسم اليوم (مثل Sun أو Mon).

`Mmm`: يمثل اسم الشهر (مثل Feb أو Jan).

`dd`: يمثل يوم الشهر (من 1 إلى 31).

`hh`: يمثل الساعة (من 00 إلى 23).

`mm`: يمثل الدقيقة (من 00 إلى 59).

`ss`: يمثل الثواني (من 00 إلى 59).

`yyyy`: يمثل السنة + 1900.

```
    cout << "\n\tUTC Time and Date is : " << TD << endl;
```

شرح: يقوم هذا السطر بطباعة السلسلة النصية `TD` التي تحتوي على التاريخ والوقت بتوقيت `UTC` باستخدام `cout`.

```
    return 0;
}
```



```
Microsoft Visual Studio Debug Console
Local Time and Date Now is : Mon Feb 26 02:54:55 2024
UTC Time and Date now is : Mon Feb 26 00:54:55 2024
```



# Lesson #60 :

## Date & Time Structure

### <ctime> (time.h)

#### Functions

##### Time manipulation

معالجة الوقت:

<b>clock()</b>	Clock Program (function)	تحصل على عدد وحدات الزمن التي انقضت منذ بدء تشغيل البرنامج.
<b>difftime()</b>	Return difference between two times (function)	تحسب الفرق بين نقطتين زمنيتين من نوع <code>time_t</code> وتعيده بالثواني.
<b>mktime()</b>	Convert tm structure to time_t (function)	تحول هيكل بيانات <code>tm</code> إلى نقطة زمنية من نوع <code>time_t</code>
<b>time()</b>	Get current time (function)	تحصل على الوقت الحالي كنقطة زمنية من نوع <code>time_t</code>

##### Conversion

التحويل:

<b>asctime()</b>	Convert tm structure to string (function)	تحول هيكل بيانات <code>tm</code> إلى سلسلة نصية قابلة للطباعة تمثل التاريخ والوقت بتنسيق محدد مسبقاً.
<b>ctime()</b>	Convert time_t value to string (function)	تحول نقطة زمنية من نوع <code>time_t</code> إلى سلسلة نصية قابلة للطباعة تمثل التاريخ والوقت بتنسيق محدد مسبقاً
<b>gmtime()</b>	Convert time_t to tm as UTC time (function)	تحول نقطة زمنية من نوع <code>time_t</code> إلى هيكل بيانات <code>tm</code> يمثل التاريخ والوقت بتوقيت UTC
<b>localtime()</b>	Convert time_t to tm as local time (function)	تحول نقطة زمنية من نوع <code>time_t</code> إلى هيكل بيانات <code>tm</code> يمثل التاريخ والوقت بالتاريخ المحلي.
<b>strftime()</b>	Format time as string (function) (str, maxsize, format, timeptr)	تنسيق التاريخ والوقت في سلسلة نصية وفقاً للتنسيق المحدد.

##### Macro constants

ثوابت الماكرو:

<b>CLOCKS_PER_SEC</b>	Clock ticks per second (macro)	يمثل عدد وحدات الزمن في الثانية الواحدة.
<b>NULL</b>	Null pointer (macro)	يمثل مؤشر فارغ

##### Types

الأنواع:

<b>clock_t</b>	Clock type (type)	يمثل نوع بيانات لوحدات الزمن التي يتم إرجاعها بواسطة دالة <code>clock</code>
<b>size_t</b>	Unsigned integral type (type)	يمثل نوع بيانات صحيح غير سالب.
<b>time_t</b>	Time type (type)	يمثل نوع بيانات لنقطة زمنية (timestamp)
<b>struct tm</b>	Time structure (type)	يمثل هيكل بيانات لتخزين معلومات التاريخ والوقت.

## دوال مكتبة time.h المستخدمة في هذا الكود :

- **time\_t** : هو نوع بيانات يُستخدم لتخزين نقاط زمنية. نقطة زمنية هي عدد الثواني التي انقضت منذ 1 يناير 1970.
- **time(0)** : تحصل على الوقت الحالي من نظام التشغيل.
- **localtime(&t)** : تحول الوقت الحالي إلى سلسلة نصية قابلة للطباعة بتنسيق محدد مسبقاً.

## شرح كود يُظهر استخدام بعض وظائف مكتبة time.h في C++ :

```
#pragma warning(disable : 4996)
#include <iostream>
#include <ctime>
using namespace std;
/*
struct tm
{
    int tm_sec; // seconds after the minute - [0, 60] including leap second
    int tm_min; // minutes after the hour - [0, 59]
    int tm_hour; // hours since midnight - [0, 23]
    int tm_mday; // day of the month - [1, 31]
    int tm_mon; // months since January - [0, 11]
    int tm_year; // years since 1900
    int tm_wday; // days since Sunday - [0, 6]
    int tm_yday; // days since January 1 - [0, 365]
    int tm_isdst; // daylight savings time flag
};
*/
int main()
{
    time_t t = time(0);

    tm* now = localtime(&t);

    cout << "Time Now : " << now->tm_hour << ":" << now->tm_min << ":" << now->tm_sec << endl;
    cout << "Date Today: " << now->tm_mday << " / " << now->tm_mon + 1 << " / "
        << now->tm_year + 1900 << endl;
    cout << "\n-----\n";
    cout << "Year : " << now->tm_year + 1900 << endl;
    cout << "Month : " << now->tm_mon + 1 << endl;
    cout << "Day : " << now->tm_mday << endl;
    cout << "Hour : " << now->tm_hour << endl;
    cout << "Min : " << now->tm_min << endl;
    cout << "Second: " << now->tm_sec << endl;
    cout << "Week Day (Days since sunday) : " << now->tm_wday << endl;
    cout << "Year Day (Days since Jan 1st) : " << now->tm_yday << endl;
    cout << "Hours of daylight savings time: " << now->tm_isdst << endl;

    return 0;
}
```

### الشرح:

**#pragma warning(disable : 4996)**

شرح: يستخدم هذا السطر لإيقاف تحذير محدد من المترجم. في هذه الحالة، يتم إيقاف التحذير رقم 4996، والذي يتعلق بوظيفة **ctime**.

**#include <iostream>**

شرح: يقوم هذا السطر بتضمين مكتبة **iostream** في الكود التي تُستخدم لعمليات الإدخال والإخراج، والذي يسمح باستخدام دالة **cout** لطباعة الرسائل.

**#include <ctime>**

شرح: يقوم هذا السطر بتضمين مكتبة **ctime** في الكود ، والذي يسمح باستخدام وظائف مكتبة **time.h** التي تحتوي على وظائف للتعامل مع الوقت والتاريخ.

```
int main()
{
```

### وظائف معالجة الوقت:

نوع البيانات **time\_t** : لتخزين الوقت الحالي ( وهو إجمالي عدد الثواني التي انقضت منذ 1 يناير 1970 ).

```
    time_t t = time(0);
```

شرح: يقوم هذا السطر باستدعاء دالة **time(0)** التي تحصل على الوقت الحالي وتخزينه في متغير من نوع **time\_t** يسمى **t**.

### وظائف التحويل:

الدالة **localtime(&t)**

يُستخدم هذا السطر لتحويل الوقت المخزن في **t** إلى (**structure - هيكل بيانات**) **tm** وتخزينه في المتغير **now**.

```
    tm* now = localtime(&t);
```

. **tm** : هو مؤشر إلى (**structure - هيكل بيانات**) **tm\***

. **structure (tm)** : يُستخدم لتخزين معلومات التاريخ والوقت.

يحتوي **tm** على معلومات حول التاريخ والوقت، مثل:

: سنة (من 1900) **tm\_year**

: شهر (من 0 إلى 11) **tm\_mon**

: يوم الشهر (من 1 إلى 31) **tm\_mday**

: ساعة (من 0 إلى 23) **tm\_hour**

: دقيقة (من 0 إلى 59) **tm\_min**

: ثانية (من 0 إلى 59) **tm\_sec**

: يوم من الأسبوع (من 0 إلى 6) **tm\_wday**

: يوم من السنة (من 0 إلى 365) **tm\_yday**

: ما إذا كان الوقت هو توقيت صيفي (DST) (0 أو 1) **tm\_isdst**

**now** : هو اسم المتغير الذي سيتم تخزين (**structure - هيكل بيانات**) **tm** فيه.

(**localtime(&t)**) : هي دالة تحول الوقت الحالي المخزن في المتغير **t** من نوع **time\_t** إلى (**structure - هيكل بيانات**) **tm** بالتوقيت المحلي وتخزينه في المتغير **now**.

القيم التي يتم تخزينها في (**structure - هيكل بيانات**) **tm** تتوافق مع إعدادات الوقت المحلي لجهاز التشغيل، بما في ذلك التوقيت الصيفي إن كان معمولاً به.

```
    return 0;
}
```

```
Microsoft Visual Studio Debug Console

Time Now : 20:59:23
Date Today: 26 / 2 / 2024

-----
Year : 2024
Month : 2
Day : 26
Hour : 20
Min : 59
Second: 23
Week Day (Days since sunday) : 1
Year Day (Days since Jan 1st) : 56
Hours of daylight savings time: 0
```