



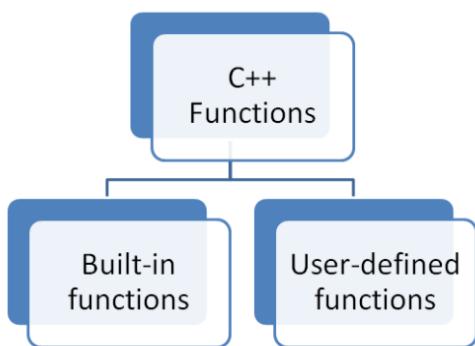
Programming 2

Suliman Alkasasbeh

الدوال (Functions)

❖ شو يعني دالة (Function) ؟

- الدالة عبارة عن **مجموعة أوامر برمجية** بتكتب مرة وحدة وبنستدعيها بأي مكان بالكود بدل ما نكرر نفس الأوامر أكثر من مرة . هاد بساعد **بتقليل حجم الكود، تسهيل قراءته، وإعادة استخدامه.**



في C++ عندنا نوعين أساسيين من الدوال:

- **الدوال الجاهزة → (Predefined Functions)** موجودة أساساً داخل مكتبات C++ ، بنستخدمها مباشرة بدون ما نكتبها.
- **الدوال اللي بكتبها المبرمج → (User-Defined Functions)** هي الدوال انت كمبرمج بتكتبها بنفسك حسب الحاجة و عندنا نوعين منهم ، نوع برجع قيمة **return function** و نوع ما رجع قيمة **void function** .

خلينا نشرح نوع نوع :

◀ **الدوال الجاهزة(Predefined Functions)**

❖ هي الدوال اللي C++ موفرتها تلقائياً داخل مكتباتها، فإنت ما بتحتاج تكتبها، بس بنستدعيها لما تحتاجها.

❖ حتى تقدر تستخدمها، لازم تضيف المكتبة المناسبة باستخدام `#include <cmath>` مثل مكتبة الـ cmath اللي رح نشرحها الان .

- شرح دوال الرياضيات الجاهزة في مكتبة الـ (cmath)

- ❖ C++ فيها مجموعة دوال رياضية جاهزة داخل مكتبة `<cmath>` ، وهي بتسهل علينا عمليات رياضية مثل الجذر التربيعي، القوة، التقريب، إلخ.
- ❖ لازم نضيف المكتبة `<cmath>` حتى نقدر نستخدم هاي الدوال .

→ `sqrt(x)`

هذا الفنكشن بيأخذ رقم (double x) وبحسب الجذر التربيعي له و برجعلي ايه.

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

int main() {
    double x = 2.25 ;
    cout << sqrt(x);
}
```

-طبعا مش شرط يكون الرقم double ممكن int / float لكن يفضل يكون casting تجنبًا للأخطاء ولو ما دخلت double بتعمل c++ للرقم و بتحوله double

- double

الجذر التربيعي ل
2.25
هو
1.5

1.5

→ `pow(x, y)`

هذا الفنكشن بيأخذ رقمين (y, x) وبحسب y^x يعني x مرفوع للقوة y

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

int main() {
    double base = 2 , power = 5;
    cout << pow(base , power);
}
```

- القوه هي ضرب الرقم بنفسه على عدد الأس يعني بالمثال عندنا

- $2^5 = 32$ يعني $(2*2*2*2*2 = 32)$

2⁵ = 32

→ `floor(x)`

التقريب لأسفل (القيمة الصحيحة الأصغر) بتأخذ رقم (double x) ويقتربه لأقرب عدد صحيح أصغر أو يساويه.

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

int main() {
    double num1 = 48.79 , num2 = 74.359 , num3 = -3.45;
    cout << " 48.79 -> " << floor(num1) << endl;
    cout << " 74.359 -> " << floor(num2) << endl;
    cout << " -3.45 -> " << floor(num3) << endl;
}
```

48.79 -> 48
74.359 -> 74
-3.45 -> -4

لاحظ عندنا تم التقريب لأقرب عدد صحيح أصغر

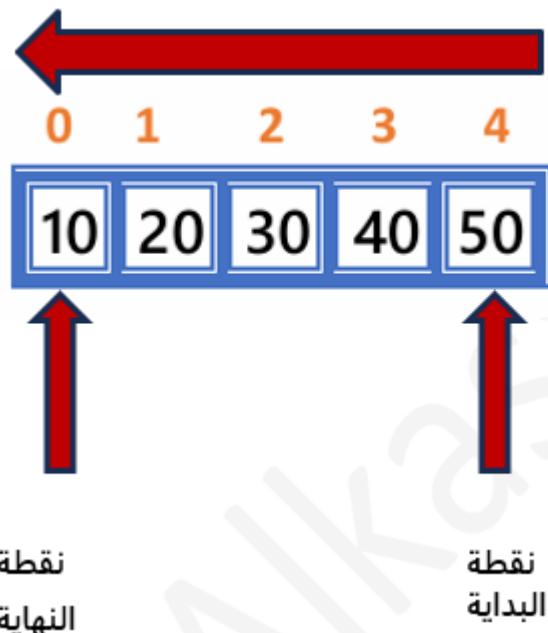
طيب ليش ال num3 تم تقريبا لل -4 ؟

قيمتها -3.45 و بما انها عدد سالب تم تقريبا لل -4

لأنه ال -4 <-3.45

طباعة عناصر المصفوفة بالعكس:

إذا كنا بدنا نطبع جميع عناصر المصفوفة بالعكس لازم نبدأ ال **for** من اكبر قيمة **index**, وتمشي بشكل عكسي لأصغر قيمة، حتى نصل من آخر عنصر ل أول عنصر يعني لازم نستخدم **for loop** عكسيه **(i=size array-1 ; i>=0 ; --i)**.



```
● ○ ●
int main() {
    int Jadara[5] = { 10,20,30,40,50 };
    for (int i = 4; i >= 0; --i)
    {
        cout << Jadara[i] << " ";
    }
}
```

Microsoft Visual Studio Debug X

```
50 40 30 20 10
C:\Users\abdel\source\re
0).
Press any key to close t
```

String

شو يعني String بالـ C++ ؟

الـ **String** هي نوع بيانات بتخزن نصوص (جمل، كلمات) مثلاً:

```
string s = "JADARA";
```

ملاحظة مهمة: الـ **string** نوع أساسي في C++ ، عشان هيك لازم نستدعي مكتبة:

```
#include <string>
```

تعريف وتهيئة المتغيرات : String

String variable

Arrays of characters

```
string str1= "AAAAA AAAAA";
string str2= "";
```

```
char str1[] = "AAAAA AAAAA";
char str2[] = "";
```

الفرق بين char array و string variable

: تخزن جمل طويلة أو قصيرة يقبل الفراغات داخل النص بدون مشاكل تقدر تعمل عمليات مثل دمج، مقارنة، قياس طول... إلخ.

: هون بنعرف مصفوفة من نوع **char** يعني كل حرف رح ينحط بخانة من خانات المصفوفة، **بتقدر تحدد الحجم حسب عدد الحروف**، أو تتركه فاضي (وهو يضبط نفسه حسب النص اللي بتكتبه).

مقارنة string باستخدام compare()

الصيغة:



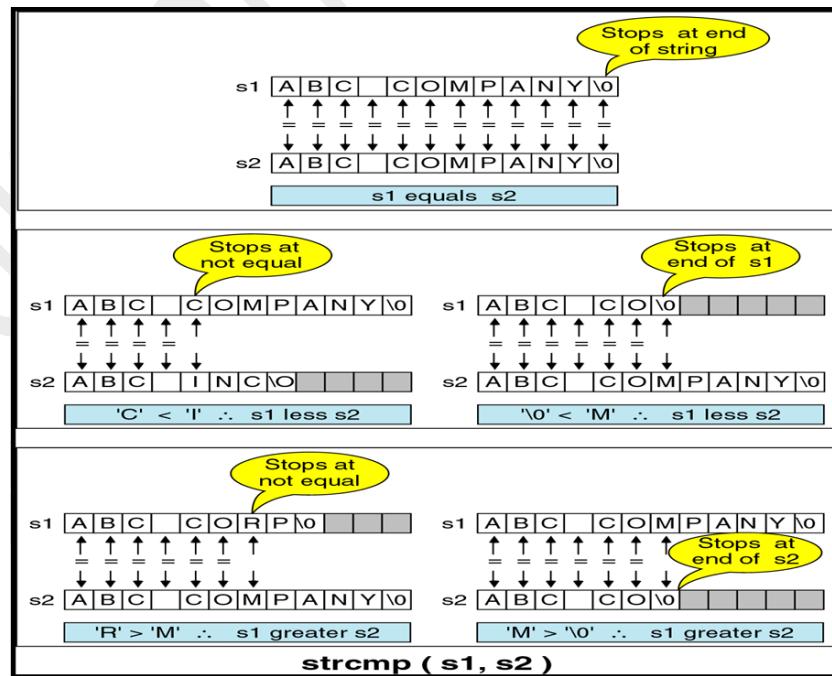
```
src.compare(dest)
```

مثال:



```
string str1 = "str1";
string str2 = "str2";

int result = str1.compare(str2);
```



Pointers

كل متغير بتعرفه في C++ بينجذله مكان في الذاكرة (RAM) ، والمكان هذا إله عنوان، يعني موقعه داخل الذاكرة.

والرمز **&** في C++ يستخدموه عشان يعطيك عنوان المتغير.

مثال:

```
int main()
{
    // declare variables
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;

    // print address of var1
    cout << "Address of var1: " << &var1 << endl;

    // print address of var2
    cout << "Address of var2: " << &var2 << endl;

    // print address of var3
    cout << "Address of var3: " << &var3 << endl;
}
```

: Output

```
C:\Users\abdel\OneDrive\Doc
Address of var1: 0x6ffe4c
Address of var2: 0x6ffe48
Address of var3: 0x6ffe44
```

كل مرة تشغل البرنامج، العنوانين ممكن تتغير، لأنها بتعتمد على مكان تخزين المتغيرات وقت التشغيل.

يعني لو شغلت نفس الكود 3 مرات، ممكن يطبع عنوانين مختلفة كل مرة.
هذا بسبب نظام التشغيل وطريقة إدارة الذاكرة.

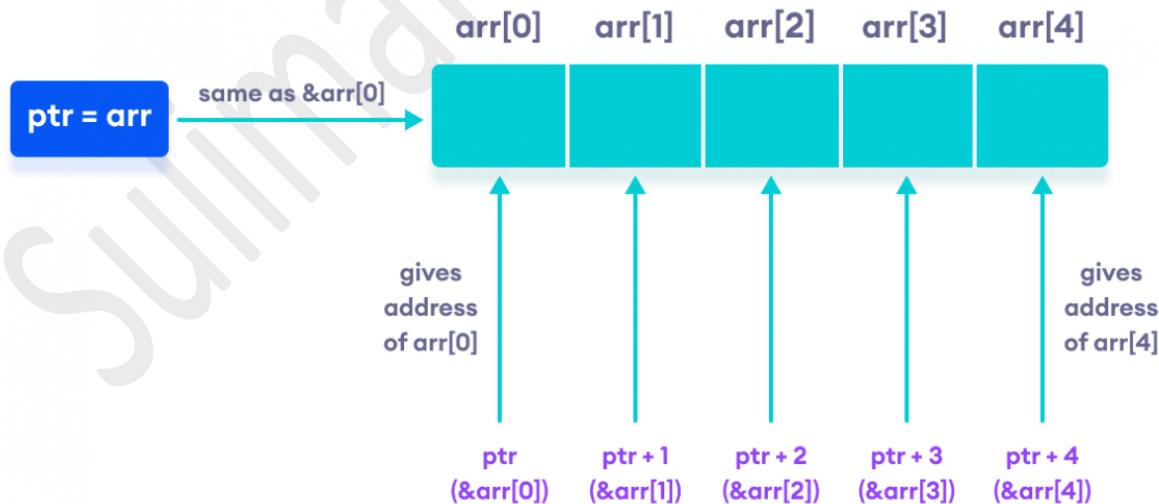
المؤشرات في C++ مش بس بتخزن عنوان متغير واحد، كمان بتقدر تخزن عنوان خلايا المصفوفة.



```
int arr[5];
int *ptr;
ptr = arr;
```

هون السطر `ptr = arr;` معناه إن المؤشر `ptr` صار يخزن عنوان أول عنصر في المصفوفة، أي نفس قيمة `&arr[0]`. وبالتالي، لو كتبنا `ptr[*ptr + 1]` فهي بتعطينا قيمة أول عنصر، و `(ptr + 1)[*ptr]` بتعطينا قيمة العنصر الثاني، وهكذا.

أما بالنسبة للمسافات بين العناوين في الذاكرة، فهي تعتمد على نوع البيانات.
إذا كان المؤشر من نوع `int*`، فالمسافة بين `ptr` و `ptr + 1` هي 4 بايت لأن حجم `int` هو 4 بايت.
بينما لو كان المؤشر من نوع `char*`، المسافة تكون 1 بايت فقط لأن حجم `char` = 1 بايت
الفكرة هون إن المؤشرات بالمصفوفات بتتحرك في الذاكرة حسب حجم النوع اللي بتشير إله، مش رقمياً فقط.



Shallow VS Deep Copy Pointer

ـ Shallow Copy أو "النسخ السطحي" هو لما يكون عندنا مؤشرين أو أكثر من نفس النوع بيشيروا لنفس الموقع في الذاكرة.

يعني بدل ما نعمل نسخة جديدة من البيانات، بننسخ العنوان فقط، فكل المؤشرات بتدل على نفس البيانات الأصلية.

```
● ● ●
int *first;
int *second;

first = new int[10];
```

هون عرّفنا مؤشرين من نوع int: first و second، وخصصنا للمؤشر first مصفوفة ديناميكية من 10 عناصر باستخدام new، فصار يشير لأول عنصر بالمصفوفة.

```
first → [10 | 36 | 89 | 29 | 47 | 64 | 28 | 92 | 37 | 73]
```

بعدين بنعمل:

```
● ● ●
second = first;
```

بهذا السطر ما انعملت نسخة جديدة من المصفوفة، إنما صار المؤشر الثاني (second) يشير لنفس موقع الذاكرة اللي المؤشر الأول (first) بيشير له. يعني الاثنين صاروا بيشاوروا على نفس البيانات بالضبط، ومهما عدلنا على واحد منهم التغيير رح يبيّن على الثاني كمان، لأنهم مربوطين بنفس العنوان في الذاكرة.

```
first → [10 | 36 | 89 | 29 | 47 | 64 | 28 | 92 | 37 | 73]
second → [10 | 36 | 89 | 29 | 47 | 64 | 28 | 92 | 37 | 73]
```

OOP vs Struct

شو هي OOP وليش بنسخدمها؟

- طريقة لبناء البرامج عن طريق تجميع البيانات (Attributes) والدوال (Methods) اللي بتعامل معها داخل "كائنات" **class** مبنية وفق "قوالب" اسمها **objects**

< ليش نستخدم ال OOP ؟

- بتنظم الكود وتسهل قراءته.

- بتقلل التكرار (Code Reuse)

- بتحمي البيانات (Encapsulation)

- بتسهل تعديل وتوسيع البرنامج

قبل ال **oop** كان في مفهوم او طريقة برمجة اسمها **struct** عشان نجمع أكثر من متغير تحت اسم واحد

: struct خلينا نشوف بناء ال

- بتكون عندنا أولاً من ال Keyword بالبداية وهي **struct**.

- ثانياً **StructName** و قواعد تسميته مثل قواعد تسمية ال **variables**

- ثالثاً الأقواس و لازم تسكتهم ب **(;)** semicolon



```
1 struct StructName {
2     dataType variable1;
3     dataType variable2;
4     ...
5 };
```

Class Methods

لما نعرف دوال (methods) داخل الكلاس، بنقدر نكتبها بطرفيتين:

> Class Methods – Inside

هون بنكتب تعريف الدالة داخل الكلاس نفس المثال السابق
هاي الطريقة سهلة ومناسبة لما تكون
الدالة قصيرة

```

4 class Car {
5 public:
6     string brand;
7     int year;
8
9     void start() {
10        cout << brand << " is starting..." << endl
11    }
12 };

```

هون حطينا الفنكشن start داخل
الكلاس ، بنسخدمها عن طريق ال
ObjectName.MethodName()
(Car1 . year)

Example

> Class Methods - Outside

هون نعلن عن الدالة جوّا الكلاس، بس نعرفها برا الكلاس باستخدام الرمز (::) .

تتذكروا ببرمجة 1 لما أخذنا مفهوم ال prototype؟ كنا نكتب بس إعلان للدالة بالبداية،
وبعدين نكتب تعريفها بعد main؟

```

● ● ●
1 class Car {
2 public:
3     string brand;
4     int year;
5
6     void start();
7 };
8
9 void Car::start() {
10
11     cout << brand << " is starting...\n" << endl;
12 }

```

نفس الفكرة هون

جوّا الكلاس بس كتبنا إعلان للدالة
وبرأ الكلاس كتبنا التعريف و شو وظيفة
الفنكشن
بنستخدم (:: ClassName) حتى نوضح
انه الدالة تابعة للكلاس

بنستخدم هالطريقة لما تكون الدالة
طويلة أو الكلاس كبير.

إحنا قبل شوي شفنا الـ Constructor العادي (بدون معاملات) اللي كان يعطي **قيمة**
افتراضية داخل الكود.

بس أحياناً بدنـا نقدر نمرر قيمة مختلفة لكل كائن جديد نعمله.

هون بيجي دور الـ Constructor هو **Parameterized Constructor** هو عيشان نقدر نمرر قيمة مختلفة للـ **كائن** وقت إنشاؤه (parameters).

عندنا للـ **Parameterized** طريقتين لكتابـه :

الأولى داخل الكلاس يعني نكتب تعريف الـ Constructor جوا نفس الكلاس مباشرة

مثال :

```
class Car {           // The class
public:             // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
Car(string x, string y, int z) { // Constructor with parameters
    brand = x;
    model = y;
    year = z;
}
```

```
int main() {
    // Create Car objects and call the
    // constructor with different values
    Car carObj1("BMW", "X5", 1999);
    Car carObj2("Ford", "Mustang", 1969);

    // Print values
    cout << carObj1.brand << " " <<
    carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " <<
    carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```

class Car { ... };

عرفنا كلاس اسمـه Car يحتوي ثلاثة خصائص (brand, model, year).

Car (string x, string y, int z)

هذا هو الـ Constructor، بياخذ 3 قيم: نوع السيارة، موديلها، وسنـتها.

brand = x; model = y; year = z;
داخل الـ Constructor بنخزن القيم الممـرة في خصائص الكلاـس.

في main() أنشأنا كـائنـين:

carObj2("Ford", "Mustang", 1969) و carObj1("BMW", "X5", 1999)

كل كـائن أخذ قـيم مـختلفـة، وتم تخـزينـها تلقـائـاً عن طـريق الـ

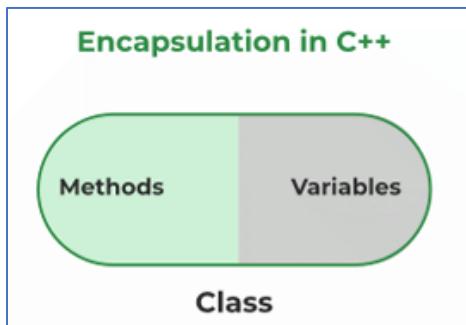
Constructor بعدـين طبعـنا الـ قـيم، وشفـنا كل سيـارة مـطبـوعـة بمـواصفـاتـها الخـاصـة.

Encapsulation

في البرمجة الكائنية(OOP) ، في مبدأ مهم اسمه Encapsulation أو التغليف هي طريقة بنغلف فيها البيانات والدوال مع بعض داخل وحدة واحدة اسمها "class" ، بحيث تكون البيانات محمية وما حدا يقدر الوصول إليها إلا بطرق معينة ومحددة

تعريف الـ Encapsulation

- هي عملية جمع (تغليف) البيانات (Data Members) والدوال (Methods) اللي بتعامل معها داخل الـ (Class)



طيب احنا ليش بنستخدم الـ Encapsulation فيه سببين رئيسين:

- **حماية البيانات Data Protection :**

البيانات جوا الكلس محمية، وما حدا بيقدر يغيرها إلا بطرق محددة / (Get / Set)

وهذا بمنع التعديل الخاطئ أو الوصول الغير مصرح به.

- **إخفاء التفاصيل Information Hiding :**

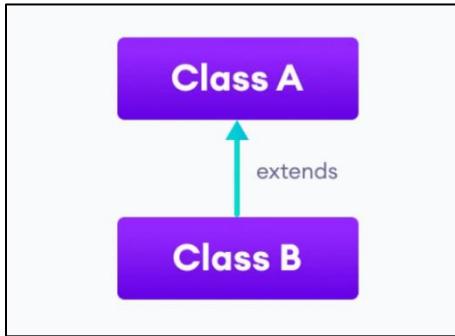
المستخدم ما بشوف كيف الكلس شغال داخلياً،

هو بس بتعامل مع واجهة بسيطة (Public Methods)

يعني تقدر تغيّر الكود الداخلي بدون ما يتأثر المستخدم الخارجي.

أنواع الوراثة في C++

Class B : public A



الوراثة المفردة (Single Inheritance)
يعني عندي **كلس واحد** (B) **يرث من** **كلس واحد** (A)

مثال :

```

#include <iostream>
using namespace std;

class A {
public:
    int x = 10;
    void printMessage()
    {
        cout << "Hello from class A \n";
    }
};

class B: public A {
public:
    int y = 20;
};
  
```

```

int main()
{
    B b;
    cout << "y = " << b.y << "\n";
    cout << "x = " << b.x << "\n";
    b.printMessage();

    return 0;
}
  
```

- **الكلس A هو الكلس الأب**، فيه متغير **x** ودالة **printMessage()**
- **الكلس B هو الابن**، ورث من A باستخدام **public**
- لما أنشأنا كائن **b** من الكلس B، قدرنا نوصل:
 - **للمتغير y من B**
 - **للمتغير x من A**
 - **وللدالة PrintMessage() من A**

```

y = 20
x = 10
Hello from class A
  
```

Output