C++: מורה נבוכים

std::pair [C++11]

STL Container שמחזיק 2 ערכים, שיכולים להיות מסוג שונה זה מזה.
בד"כ נועד להחזרת יותר מערך אחד מפונקציה או כדי לשמור זוגות key-value ב-containers.

std::set [C++98]

STL Container שמחזיק ערכים ייחודיים (unique) באופן ממוין.

unpacking [C++17] מ-std::pair

ניתן לבצע unpacking מ-`std::pair`, בדומה ל-Python, בצורה הבאה:

```
std::pair<int, std::string> person(1, "Alice");
auto [id, name] = person; // id = 1, name = "Alice"
```

שימוש ב-std::vector [C++98] במקום std::list

`std::vector` הוא מערך דינאמי רציף בזיכרון שמציע element access מהיר יותר, ויעיל יותר בהיבטי זיכרון, ולכן בד"כ יהיה עדיף על פני `std::list`, שממומשת כרשימה מקושרת דו-כיוונית (ולכן יתרונה היחיד הוא בהכנסה ובהוצאה).

std::find [C++11]

פונקציית STL שמבצעת חיפוש ב-range (בד"כ container), ומחזירה iterator למיקום של האלמנט.

std::begin, std::end [C++11]

בדומה ל-`.begin()` ול-`.end()`, שממומשות עבור containers, `std::begin` ו-`std::end` גם הן מחזירות iterator-ים להתחלה או לאיבר-אחרי-אחרון של range מסויים, אלא שהן פונקציות כלליות, שעובדות גם על containers וגם על מערכים רגילים (C-like).

std::function [C++11]

סוג של wrapper עבור עצמים שהם callable, כדוגמת פונקציות רגילות ופונקציות lambda. סוג של pointer שיכול לשמור, להעתיק ולקרוא לעצמים האלה. שימושי בעיקר עבור callbacks והעברת פונקציות כפרמטר.

```
// Function that takes a std::function as a parameter
void executeFunction(const std::function<void(int)>& func, int
                     value) {
    func(value); // Invoke the passed function
}

int main() {
    // Using std::function to wrap a lambda
    std::function<void(int)> print = [](int x) { std::cout << x; };

    // Passing the std::function to another function
    executeFunction(print, 10);
}
```

Move Semantics [C++11]

מנגנון שמאפשר "הזזת" משאב ממשתנה אחד לאחר. המנגנון חוסך העתקות רבות, ולכן יעיל יותר מבחינת ביצועים. ממומש כברירת-מחדל ב-STL וב-Smart Pointers.

noexcept [C++11]

keyword שמסמן פונקציה ככזו שלא זורקת exceptions. מסייע לקומפיילר באופטימיזציות ולכן משפר ביצועים.

```
void safeFunction() noexcept {}
```

final [C++11]

keyword בעל 2 שימושים: מניעת ירושה ממחלקה, מניעת override לפונקציה וירטואלית.

```
class FinalClass final { // Cannot be inherited
   // Class implementation
};

class Base {
public:
   // Cannot be overridden
   virtual void display() final { std::cout << "Base display\n"; }
};
```

override [C++11]

keyword שמסמן פונקציה ש"רוכבת" על פונקציה וירטואלית. שימושי כדי לתפוס שגיאות בזמן קומפילציה (הקומפיילר יזהה שלא הוגדרה פונקציה "רוכבת" על הפונקציה הוירטואלית שהוגדרה).

```
class Base {
public:
    virtual void show() { std::cout << "Base show\n"; }
    virtual void display() { std::cout << "Base display\n"; }
};

class Derived : public Base {
public:
    // Correctly overrides
    void show() override { std::cout << "Derived show\n"; }
    // Incorrect example: will cause a compile error
    void show(int x) override { std::cout << "Derived show\n"; }
};
```

<=> [C++20]

זהו spaceship operator, שמוגדר עבור מחלקה, ונועד כדי להשוות בין 2 מופעים (instances) של מחלקה כלשהיא, ולהחזיר האם מופע a גדול (greater), קטן (less) או שווה (equal) למופע b. יכול להיות מוגדר באופן דיפולטיבי או ידני (custom). היתרון שלו הוא שבעזרת ההגדרה שלו בלבד, הקומפיילר יודע לג'נרט את כל 6 האופרטורים האחרים: `<`, `>`, `==`, `!=`, `<=`, `>=`.

```
class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}
```

```
    // Custom spaceship operator implementation
    auto operator<=>(const Point& other) const {
        if (x != other.x) { return x <=> other.x; }
        return y <=> other.y;
    }
};

int main() {
    Point p1(3, 4);
    Point p2(5, 6);
    Point p3(3, 4);

    // Using the custom comparison operators
    if (p1 < p2) { std::cout << "p1 is less than p2\n"; }
    if (p1 > p2) { std::cout << "p1 is greater than p2\n"; }
    if (p1 == p3) { std::cout << "p1 is equal to p3\n"; }
    if (p1 != p2) { std::cout << "p1 is not equal to p2\n"; }
    if (p1 <= p2) { std::cout << "p1 is less than or equal to p2\n";
                    }
    if (p2 >= p1) { std::cout << "p2 is greater than or equal to
                      p1\n"; }
    return 0;
}
```

Smart Pointers [C++11]

מצביעים שמטרתם לנהל את הזיכרון המוקצה באופן מיטבי כדי למנוע זליגות ושגיאות.
`std::unique_ptr<type>`- מצביע למשאב אחד מסוג (type) מסויים באופן בלעדי. לא ניתן להעתיק אותו, אלא רק להזיז (move) אותו. המשאב נמחק כאשר המצביע נמחק.
`std::shared_ptr<type>`- מצביע למשאב אחד מסוג (type) מסויים באופן שאינו בלעדי, כלומר ניתן להעתקה. המשאב נמחק כאשר המצביע האחרון אליו נמחק.

std::make_shared [C++11] ,std::make_unique [C++14]

פונקציות מובנות שמטרתן לייצר משאב (הקצאה דינאמית) ולהחזיר מצביע חכם אליו.

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
std::shared_ptr<int> ptr = std::make_shared<int>(100);
```

inline [C++98]

מאפשר להצהיר ולהגדיר משתנה או פונקציה ולהשתמש בהם במספר קבצי `.cpp` אחרים ללא בעיות linking (multiple definition).

```
// header.h
#pragma once
inline int add(int a, int b) { return a + b; }

// file1.cpp
#include "header.h"
int main() {
    add(2, 3);
    return 0;
```

```
}

// file2.cpp
#include "header.h"
int main() {
    add(5, 7);
    return 0;
}
```

extern [C++98]

מאפשר להצהיר על משתנה או פונקציה שמוגדרים בקובץ אחר, כדי להשתמש בהם בקובץ הנוכחי.
מאפשר שימוש במשתנים גלובאליים תוך מניעת שגיאות linking (multiple definition).

```
// file.cpp
void display() { std::cout << "Extern" << std::endl;}

// main.cpp
extern void display(); // Declaration of function

int main() {
    display(); // Outputs "Extern"
    return 0;
}
```

[C++11] איתחול משתני מחלקה בעת הגדרתם

ניתן לאתחל משתנים של מחלקה מסויימת בזמן הגדרת המחלקה. מונע את הצורך באיתחול ב-constructor.

```
class MyClass {
public:
    int a = 5; // In-class member initializer
    double b = 3.14; // In-class member initializer
    // Constructor
    MyClass(int x) : a(x) { } // 'b' is initialized by default value
};

int main() {
    MyClass obj(10);
    std::cout << "a: " << obj.a << std::endl; // Outputs: a: 10
    std::cout << "b: " << obj.b << std::endl; // Outputs: b: 3.14
    return 0;
}
```

[C++17] איתחול משתנה סטטי של מחלקה בעת ההגדרה

בעזרת ה-keyword `static`, ניתן לאתחל משתנה סטטי (static) של מחלקה עם הגדרתו, ואין צורך לעשות זאת בקובץ `.cpp` נפרד, מחוץ למחלקה.

```
class MyClass {
public:
    // Inline static member variable initialization
```

```
    inline static int count = 0; // Initialization directly in the
                                 // class definition
    MyClass() { ++count;}
};

int main() {
    MyClass obj1;
    MyClass obj2;
    std::cout << MyClass::count << std::endl; // Outputs: 2
    return 0;
}
```

איתחול אחיד [C++11]

ניתן לאתחל משתנים מכל סוג בצורה אחידה (מבחינת syntax): {}. יותר נוח וגם מונע שגיאות כדוגמת narrowing conversion.

```
int c{5};
double d{4.5};
int e(3.14); // e is initialized to 3, narrowing conversion occurs
int f{3.14}; // Error: narrowing conversion from double to int
```

explicit [C++98]

זהו keyword שמונע המרה מרומזת (Implicit conversion) בקריאה ל-constructor עם ארגומנט יחיד.

```
class MyClass {
public:
    // Constructor marked as explicit
    explicit MyClass(int value) {}
};

int main() {
    MyClass obj1(10); // OK: Direct initialization
    MyClass obj2(10.5); // Error: No implicit conversion from double
    return 0;
}
```

const [C++98]

שימוש ב-keyword זה בפונקציה של מחלקה מבטיח שהפונקציה לא משנה משתנים של המחלקה.

```
class MyClass {
private:
    int value;
public:
    MyClass(int v) : value(v) {}
    // Const method- this method won't modify the object
    int getValue() const { return value; }
   // Non-const method- this method modifies the object
    void setValue(int v) { value = v; }
};
```

```
int main() {
    MyClass obj(42);
    std::cout << "Value: " << obj.getValue() << std::endl; // OK
    obj.setValue(100); // OK

    const MyClass constObj(84);
    std::cout << constObj.getValue() << std::endl; // OK
    constObj.setValue(200); // Error: Can't call non-const method on
                            // const object
    return 0;
}
```

default [C++11]

מאפשר להגדיר constructor בצורת ברירת-המחדל שלו. טוב בעיקר לקריאות (ברור מה הכותב רצה לעשות), ולמקרים בהם נדרש לממש constructor, ורוצים שהוא יהיה דיפולטיבי.

```
class Counter {
public:
    int count;
    Counter() = default; // Compiler-generated default constructor
    Counter(int c) : count(c) {} // Parameterized constructor
};

int main() {
    Counter c1; // Default constructor (count is uninitialized)
    Counter c2(10); // Parameterized constructor (count is
                    // initialized to 10)
return 0;
}
```

Concepts [C++20]

מנגנון שנועד לאכוף פרמטרים לפונקציות תבנית (templated functions). שימושי בהגברת הקריאות של הקוד (כי ברור מה הפונקציה מצפה לקבל), ובתפיסת שגיאות בזמן קומפילציה.
יש להשתמש ב-keyword `concept`.
ניתן להגדיר concepts ידניים (custom), וניתן לשלב concepts.

```
// Define a concept
template<typename T>
concept Integral = std::is_integral_v<T>;

template<Integral T>
T add(T a, T b) {return a + b; }

int main() {
    add(5, 10); // OK: int
    add(5.5, 10.1); // Compile error: double not Integral
    return 0;
}
```

Lambda Function [C++11]

פונקציות lambda (פונקציות אנונימיות) הן פונקציות "קלות משקל" שמוגדרות "על הדרך". מדובר בפונקציות רגילות, אך עקב גודלן, יש להן syntax נוח יותר, שהופך את הקוד לקריא יותר ותורם גם לפונקציונאליות.

syntax: `[capture](parameters) -> return_type { body }`.

```
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    // Lambda to print each number
    auto print = [](int n) { std::cout << n << " "; };

    // This will output: 1 2 3 4 5
    std::for_each(numbers.begin(), numbers.end(), print);
    return 0;
}
```

std::optional, std::nullopt [C++17]

`std::optional` מציין אובייקט שמכיל או לא-מכיל ערך. זה שימושי בשביל להגביר קריאות של קוד (זה ברור שפונקציה שמחזירה `std::optional` עלולה לא להחזיר ערך מסיבה כלשהיא), למנוע שימוש בערכים כמו `-1` כדי לציין שגיאה או אין-ערך, ואף למנוע null dereferencing (מאחר שהקוד לא יתקמפל אם לא בודקים את הערך לפני שקוראים אותו).

`std::nullopt` מציין אובייקט שלא-מכיל ערך.

```
std::optional<int> getValue(bool returnValue) {
    if (returnValue) {
        return 42; // Returns an integer wrapped in std::optional
    }
    return std::nullopt; // Returns an empty optional
}

int main() {
    auto value = getValue(true);
    if (value) { // Check if value is present
        std::cout << *value << std::endl; // Outputs: 42
    }
    auto emptyValue = getValue(false);
    if (!emptyValue) { // Check if value is absent
        std::cout << "No value" << std::endl; // Outputs: No value
    }
    return 0;
}
```

auto [C++11]

keyword שמשמש לזיהוי אוטומטי של סוג משתנה בזמן האיתחול שלו. משפר קריאות וגמישות של קוד.

`auto` לא משמר `const`.

`auto&` משמר `const`.

```
auto x = 10; // Deduces int
auto y = 3.14; // Deduces double
auto z = "Hello"; // Deduces const char*
```

```
const int a = 10;
auto b = a; // b is non-const int
auto& c = a; // c is const int&
```

using [C++11]

מספק syntax נוח יותר שמחליף את `typedef`.

```
typedef int* IntPointer;
using IntPointer = int*; // Equivalent
```

enum class [C++11]

גירסה משודרגת ל-`enum` המקורי. היתרונות של `enum class`: גישה לערך דרך שם ה-`enum` כדי למנוע התנגשות שמות משתנים (משתנה אחר באותו שם), לא מבצע המרה של הערכים ל-`int` (מה שעלול לגרום לשגיאות בקוד).

```
enum class Color { Red, Green, Blue };
enum Direction { North, South, East, West }; // Traditional enum
Color c = Color::Red; // No conflicts
Direction d = Color::Red; // Error: type mismatch (good)
int x = South; // Implicitly converts to int (bad)
```

using enum [C++20]

feature שמטרתו להקל על השימוש ב-`enum class`, ע"י הבאת המשתנים של ה-`enum` לתוך ה-scope הנוכחי.

```
enum class Color { Red, Green, Blue };

int main() {
    using enum Color; // Brings Red, Green, Blue into scope
    Color color = Red; // No need for Color:: prefix
    return 0;
}
```

constinit [C++20]

keyword שמטרתו לבצע איתחול למשתנים בזמן קומפילציה, ולא בזמן ריצה (איתחול בזמן ריצה, או dynamic initialization, הוא נושא בעל בעיות פוטנציאליות משלו).

```
constinit int x = 42; // Guaranteed compile-time initialization

int getValue() {
    return 42;
}
constinit int y = getValue(); // Error: getValue() is not constexpr
```

consteval [C++20] ,constexpr [C++11]

`constexpr` מאפשר לציין משתנים או פונקציות שיכולות (אבל לא חייבות) להיות מחושבות (evaluated) בזמן קומפילציה. מטרתו לשפר ביצועים.

`consteval` מציין פונקציה שחייבת להיות מחושבת (evaluated) בזמן קומפילציה. זה מאפשר לוודא ערכים קבועים בחלקים קריטיים של הקוד ומונע שגיאות זמן-ריצה.

```
consteval int getConstEval() { return 42; }
constexpr int getConstExpr(int x) { return x * x; }
```

```
int main() {
    constexpr int a = getConstEval(); // OK
    int b = getConstEval(); // Error: consteval function must be
                            // evaluated at compile time
    constexpr int c = getConstExpr(5); // OK, evaluated at compile
                                       // time
    int d = getConstExpr(5); // OK, evaluated at runtime
    return 0;
}
```

std::string_view [C++17]

משתנה "קל משקל" ויעיל, שהוא reference ל-std::string ללא יכולת לנהל את התוכן שלה. שימושי מאוד למקרים בהם נדרשת קריאה מ-std::string בלבד, ללא יכולת שינוי (נפוץ מאוד כפרמטר לפונקציה, לדוגמה).
יש לשים לב שאם התוכן של std::string משתנה, לא ניתן לגשת יותר ל-std::string_view שלה (undefined behavior), מאחר ש-std::string_view עדיין יצביע (refer) לבתים הקודמים בזיכרון. במקרה זה יש להגדיר std::string_view חדש.

```
// Function that takes a string_view
void printMessage(std::string_view message) {
    std::cout << message << std::endl; // Prints the message
}

int main() {
    std::string str = "Hello, World!";
    std::string_view sv = str; // sv references the data in str
    printMessage(sv); // Prints: Hello, World!

    // Modify str
    str = "Goodbye, World!"; // str now points to new memory
    printMessage(sv); // sv is now dangling and may reference
                      // invalid memory
    return 0;
}
```

int array[] [C++98] לעומת int* array

ייצוג בזיכרון: int array[] הוא מסוג array שגודלו ידוע בזמן קומפילציה ולכן הוא משוריין (allocated) מראש. int* array הוא מסוג pointer ל-integer שעשוי להיות בודד, או איבר ראשון של מערך integers. לא משוריין (allocated) זיכרון מראש, אלא דינאמית.

```
int array[5]; // An array of 5 integers, automatically allocated
int* array = new int[5]; // An array of 5 integers, dynamically
allocated
```

מידע על הגודל: גודלו של int array[] ידוע תמיד, וניתן לבצע sizeof(array) כדי לקבלו. גודלו של int* array אינו ידוע. הוא תמיד יהיה 4 או 8 בתים (כגודל sizeof(int*)).

```
int size = sizeof(array) / sizeof(array[0]); // Size of the array
// No way to determine the size of the dynamically allocated array
// without tracking it
```

איתחול: ניתן לאתחל int array[] בזמן הגדרתו. לא ניתן לאתחל int* array ללא שיריון (allocation) זיכרון תחילה.

```
int array[] = {1, 2, 3, 4, 5}; // Initializes with values
int* array = new int[5]{1, 2, 3, 4, 5}; // Dynamically allocated and
initialized
```

שיחרור זיכרון: הזיכרון של `int array[]` משתחרר באופן אוטומטי כשהוא יוצא מחוץ ל-scope.
הזיכרון של `int* array` חייב להיות משוחרר באופן ידני ע"י `delete[]`.

נקודה חשובה: בהעברת `int array[]` לפונקציה, השימוש בו בתוך הפונקציה הוא כ- `int* array`.

```
void printArray(int array[], int size) { // array[] decays to int*
    for (int i = 0; i < size; ++i) {
        // Accessing elements using pointer notation
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;
}
```

raw string literals [C++11]

צורה מיוחדת של שימוש ב-string literals, עם features שמקילים על הכותב ומגבירים קריאות, כמו לדוגמה תמיכה בירידת שורה, וביטול הצורך ב-escaping.

```
const char* rawString = R"(A "raw" string with a backslash: \)";
const char* multiLineString = R"(Line 1
                                 Line 2
                                 Line 3)";
const char* jsonString = R"({"name": "John", "age": 30})";
```

digit separator [C++14]

feature שמאפשר להוסיף הפרדה בין ספרות, להגברת הקריאות, ללא שינוי בערך המשתנה.

```
int main() {
    int million = 1'000'000; // Readable format for one million
    long long largeNumber = 1'234'567'890; // Readable format for a
                                           // large number
    return 0;
}
```