

תכנות מכוון עצמים ו- ++C

יחידה 03

enum, הכלת אובייקטים, const, inline

קרן כליף

# ביחידה זו נלמד:

- enum
- מחלקה המכילה מחלקה אחרת
- בעיית ה- include הכפולים
- const
  - פרמטר const
  - שיטת const
  - ערךמשתנה const
  - מוחזר const
- constexpr
- מתודות inline

# enum

---

# enum

- הגדרת טיפוס חדש שיכיל ערך מספרי מתוך קבוצה מוגדרת מראש
- כלומר, הגדרת אוסף קבועים בעלי קשר לוגי

- למשל: ימות השבוע, אוסף צבעים, ערכים בוליאנים וכד'

- דוגמה:

```
enum eColor {RED, YELLOW, BLUE};  
↔  
const int RED = 0;  
const int YELLOW = 1;  
const int BLUE = 2;
```

- זוהי הגדרה של 3 קבועים, הראשון מקבל באופן אוטומטי את ערך 0, זה שאחריו את ערך 1 וכו'.

- כעת נוכל להגדיר בתוכנית משתנים מטיפוס eColor ולתת להם את הערכים RED/YELLOW/BLUE

# enum שימוש בערכיו

```
enum eColor {RED, YELLOW, BLUE};
```

```
int main()
```

```
{
```

```
↔ eColor c1 = RED;
```

```
eColor c2 = eColor::RED;
```

```
cout << c1 << endl;
```

```
cout << c2 << endl;
```

```
}
```

יותר קריא להשתמש בשם המלא של ה-enum, כך יותר ברור לקורא הקוד להבין מהו ההקשר של הקבוע

0

0

# enum מתן ערך שאינו בקבוצה

- אם ניתן למשתנה מטיפוס ה-enum ערך מספרי שאינו הוגדר בקבוצת הערכים שלו, נראה את הערך המספרי (לא נקבל שגיאה)
- קליטה לתוך enum צריכה להתבצע לתוך int ואז לבצע המרה

```
enum eColor {RED, YELLOW, BLUE};
```

```
int main()
{
    eColor c1 = RED;
    cout << "c1=" << c1 << endl;

    eColor c2 = (eColor)5;
    cout << "c2=" << c2 << endl;

    eColor c3;
    //cin >> c3;
    int colorNum;
    cin >> colorNum;
    c3 = (eColor)colorNum;
}
```

c1=0  
c2=5

הקומפיילר לא מבצע casting מ- int לטיפוס החדש, לכן צריך לעשות המרה מפורשת

# enum טריק לקבלת שמו של הקבוע

- ניתן להגדיר מערך גלובלי של מחרוזות עם שמות ה-enum בהתאמה לערכיהם

```
enum eColor { RED, YELLOW, BLUE };  
const char* colors[] = { "Red", "Yellow", "Blue" };  
  
int main()  
{  
    cout << "Selected color is " << colors[eColor::RED] << endl;  
}
```

המרה מ-enum ל-int  
מבוצעת באופן אוטומטי

```
Selected color is Red
```





# enum מתן ערכים שונים

- ניתן לכל ערך באוסף לתת ערך שאינו עוקב לערך שלפניו, ע"י השמה:

מאחר ולא נתנו ל- Red ערך, ערכו יהיה ערך עוקב לקבוע שלפניו

```
enum eColor { White = 10, Black = 20, Red,  
              Yellow = 40, Blue = 50 };
```

```
int main()  
{  
    eColor c1 = (eColor)1, c2 = (eColor)20,  
             c3 = (eColor)21, c4 = (eColor)45;  
}
```

 c1	1	eColor
 c2	Black (20)	eColor
 c3	Red (21)	eColor
 c4	45	eColor



# Enhanced Enums

```
enum eGrades:char {A, B, C};  
//enum eGrades:char { 'A', 'B', 'C' };  
  
int main()  
{  
    char mathGrade = eGrades::A;  
    cout << "Math grade is " << mathGrade << endl;  
  
    char physicsGrade = eGrades::B + 'A';  
    cout << "Physics grade is " << physicsGrade << endl;  
}
```

למרות שהגדרנו את הטיפוס כ- char, הערכים סדרתיים ומתחילים מ- 0

'A' הוא ערך, בניגוד ל- A שהוא שם משתנה, וב- enum מגדירים שמות של משתנים שהם קבועים

Watch 1		
Name	Value	Type
mathGrade	0 '\0'	char
physicsGrade	66 'B'	char

```
Math grade is  
Physics grade is B
```

# הפתרון Enhanced Enums

```
enum eGrades:char { A='A' , B, C};

int main()
{
    char mathGrade = eGrades::A;
    cout << "Math grade is " << mathGrade << endl;

    char physicsGrade = eGrades::B/* + 'A'*/;
    cout << "Physics grade is " << physicsGrade << endl;
}
```

```
Math grade is A
Physics grade is B
```

# Enum Classes

ניתן להגדיר את ה-enum כ-enum class,  
ואז חלות עליו מספר הגבלות

```
enum class eColor {RED, YELLOW, BLUE };
```

```
int main()
```

```
{
```

```
✗ eColor c1 = YELLOW;
```

```
✓ eColor c2 = eColor::YELLOW;
```

הפניה לערך חייבת להיות  
משויכת לשם הקבוצה

```
✗ int n1 = eColor::YELLOW;
```

```
✓ int n2 = (int)eColor::YELLOW;
```

```
}
```

אין המרה אוטומטית ל- int

# enum שימוש במחלקה

ה- enum מוגדר בתוך המחלקה ולא גלובלית, משום שערכיו קשורים לעולם הבעיה של המחלקה

```
class Clock
{
public:
```

נגדיר את ה- enum ב- public כדי שיהיה נגיש גם מחוץ למחלקה

```
enum eDisplayType {FULL_DAY, HALF_DAY};
```

```
int getMinutes();
```

```
int getHours();
```

```
eDisplayType getDisplayType();
```

```
bool setMinutes(int m=0);
```

```
bool setHours(int h=0);
```

```
void setDisplayType(eDisplayType type);
```

```
void show();
```

```
void tick();
```

```
void addMinutes(int add);
```

```
private:
```

```
int hours, minutes;
```

```
eDisplayType displayType;
```

```
};
```

```
void Clock::show()
{
```

```
int h = hours;
```

```
if (displayType == HALF_DAY)
```

```
h %= 12;
```

```
cout << (h < 10 ? "0" : "")
```

```
<< h << ":"
```

```
<< (minutes < 10 ? "0" : "")
```

```
<< minutes;
```

```
if (displayType == HALF_DAY)
```

```
cout << (hours < 12 ?
```

```
"am" : "pm");
```

ה- private עבר לסוף הקובץ משום שיש בו שימוש ב- enum, שצריך להיות מוגדר מעליו

# enum שימוש במחלקה ה-main

```
void main()
{
    Clock c;

    c.setHours(22);
    c.setMinutes(30);

    c.setDisplayType(Clock::FULL_DAY);
    cout << "The time is (full day): ";
    c.show();
    cout << endl;

    c.setDisplayType(Clock::HALF_DAY);
    cout << "The time is (half day): ";
    c.show();
    cout << endl;
}
```

ה-enum הוא קבוע המוגדר בחלק ה-public במחלקה, לכן ניתן לגשת אליו בשמו המלא מחוץ למחלקה

```
The time is (full day): 22:30
The time is (half day): 10:30pm
```

מחלקה המכילה מחלקה אחרת  
ובעיית ה-include הכפולים

---

# מחלקה המכילה מחלקה

```
#ifndef __OVEN_H
#define __OVEN_H

#include "clock.h"

class Oven
{
private:
    int temperature;
    Clock startTime;
    int minutesToWork;

public:
    void show();
    int getTemperature();
    Clock getStartTime();
    int getMinutesToWork();

    void setTemperature(int t);
    void setStarTime(Clock c);
    void setMinutesToWork(int t);
};

#endif // __OVEN_H
```

```
void Oven::show()
{
    cout << "Temperature: " << temperature <<
        "\nMinutes: " << minutesToWork <<
        "\nStart working at ";
    startTime.show();
    cout << endl;
}
```

# מחלקה המכילה מחלקה שימוש ב- main

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

#include "clock.h"
#include "oven.h"

void main()
{
    Oven o;
    Clock c;

    c.setHours(13);
    c.setMinutes();

    o.setTemperature(180);
    o.setMinutesToWork(50);
    o.setStarTime(c);

    o.show();
}
```

```
Temperature: 180
Minutes: 50
Start working at 13:00
```



# בעיית ה-include הכפולים

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

```
#include "clock.h"
#include "oven.h"
```

במידה והקובץ clock.h לא היה עטוף ב-`ifndef`  
היינו מקבלים שגיאת קומפילציה של redefinition

```
void main()
{
    Oven o;
    Clock c;

    c.setHours(13);
    c.setMinutes();

    o.setTemperature(180);
    o.setMinutesToWork(50);
    o.setStarTime(c);

    o.show();
}
```



C2011

'Clock': 'class' type redefinition

# תזכורת פעולת ה-include

- פעולת ה-include היא פקודת קדם-מעבד (preprocessor) אשר שותלת בקוד במקום כל פקודת include את תוכן הקובץ שאותו כללנו בפקודה

a.h

```
struct A
{
    int x;
};
```

main.cpp

```
#include "a.h"

int main()
{
}
```



```
struct A
{
    int x;
};

int main()
{
}
```

# פעולת include הבעייתיות

- יתכן ונעשה include לקובץ מסוים יותר מפעם אחת:

a.h

```
struct A
{
    int x;
};
```

b.h

```
#include "a.h"

// prototypes
void foo();
```

main.cpp

```
#include "a.h"
#include "b.h"

int main()
{
}
```



```
struct A
{
    int x;
};

struct A
{
    int x;
};
```

```
// prototypes
void foo();

int main()
{
}
```

נקבל שגיאה של  
redefinition מאחר  
והקומפיילר רואה את  
ההצהרה על המבנה שמוגדר  
ב- a.h יותר מפעם אחת

# הפתרון לבעיית ה- include הידור מותנה

- ראינו בעבר את הפקודה `#define` לצורך הגדרת קבוע מסוים
- פקודה זו מוסיפה את הקבוע שהוגדר לטבלת סימולים של התוכנית במידה וטרם הוגדר. במידה וכבר הוגדר דורסת את ערכו.
- ניתן גם לכתוב פקודת `define` ללא ערך, רק כדי להכניס קבוע מסוים לטבלת הסימולים
- ניתן לבדוק האם קבוע מסוים הוגדר בטבלת הסימולים בעזרת הפקודה `#ifdef` או אם לא הוגדר בעזרת הפקודה `#ifndef`
- במידה והתנאי מתקיים, הקופיילר יהדר את קטע הקוד הבא עד אשר יתקל ב-  
`#endif`

# הידור מותנה מימוש

a.h

```
#ifndef __A_H
#define __A_H

struct A
{
    int x;
};

#endif // __A_H
```

b.h

```
#ifndef __B_H
#define __B_H

#include "a.h"

// prototypes
void foo();

#endif // __B_H
```

main.cpp

```
#include "a.h"
#include "b.h"

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



טבלת הסימולים:

```
__A_H
__B_H
```

main.cpp לאחר  
preprocessor

```
struct A
{
    int x;
};

// prototypes
void foo();

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

כעת יש לנו ב-main  
פעם אחת בלבד את  
ההגדרות מכל קובץ

# פעולת ה-include בעיה נוספת

- כאשר יש 2 מבנים אשר כל אחד מגדיר אובייקט מטיפוס המבנה השני, מתקבלת שגיאת קומפילציה שקשה להבינה:

```
#ifndef __A_H
#define __A_H
#include "b.h"

struct A
{
    B b;
};
#endif // __A_H
```

```
#ifndef __B_H
#define __B_H
#include "a.h"

struct B
{
    A a;
};
#endif // __B_H
```

הקומפיילר אינו מכיר את הטיפוס A  
ולכן שגיאת הקומפילציה...

```
#include "a.h"
#include "b.h"

int main()
{
}
```

טבלת הסימולים:  
\_\_A\_H  
\_\_B\_H

# הכלה דו-כיוונית הפתרון

- במקרה זה נדאג שלפחות אחד המבנים יכיל רק מצביע למבנה השני, ולא אובייקט
  - כאשר יוצרים אובייקט צריך לבצע `include` לקובץ המגדיר אותו
  - כאשר יש מצביע לאובייקט לא חייבים לבצע `include` לקובץ המכיל אותו, אלא להסתפק בהצהרה שמבנה זה יוגדר בהמשך
- בקובץ `cpp` בו תהיה היצירה של האובייקט נבצע את ה- `include` לקובץ בו מוגדר המבנה

```
#ifndef __A_H
#define __A_H
#include "b.h"

struct A
{
    B b;
};
#endif // __A_H
```

```
#ifndef __B_H
#define __B_H

struct A;

struct B
{
    A* a;
};
#endif // __B_H
```

הצהרה שהמחלקה תוגדר בהמשך

# forward declaration תוצר ה- preprocessor

```
#ifndef __A_H
#define __A_H
#include "b.h"

struct A
{
    B b;
};
#endif // __A_H
```

```
#ifndef __B_H
#define __B_H

struct A;

struct B
{
    A* a;
};
#endif // __B_H
```

```
#include "a.h"
#include "b.h"

int main()
{
}
```

טבלת הסימולים:  
\_\_A\_H  
\_\_B\_H

main.cpp לאחר  
preprocessor

```
struct A;

struct B
{
    A* a;
};

struct A
{
    B b;
};

int main()
{
}
```



# const

---

פרמטרים שהם const

מתודות const

ערך מוחזר const

# העברת פרמטרים by ref

## שדרוג המחלקה Oven

```
#ifndef __OVEN_H
#define __OVEN_H

#include "clock.h"

class Oven
{
private:
    int temperature;
    Clock startTime;
    int minutesToWork;

public:
    void show();
    int getTemperature();
    Clock& getStartTime();
    int getMinutesToWork();

    void setTemperature(int t);
    void setStarTime(Clock& c) {startTime = c;}
    void setMinutesToWork(int t);
};

#endif // __OVEN_H
```

פרמטרים מטיפוס אובייקט תמיד ישלחו by ref מטעמי יעילות, אין שינוי במימוש.

ערך מוחזר המחזיר תכונה של האובייקט שהינה אובייקט בעצמה, גם תחזור by ref מטעמי יעילות, כדי לחסוך את ההעתקה

מטעמי יעילות, כדי לא לשלוח העתק של האובייקט לפונקציה ומהפונקציה, נקבל ונשלח רק reference אליו. מקביל לשליחת מצביע ב- C

# העברת פרמטר by ref הבעייתיות

- ראינו שניתן להעביר אובייקט לשיטה by ref כדי לחסוך את ההעתקה שלו
- הבעיה: האובייקט המקורי חשוף לשינויים בתוך הפונקציה
- הפתרון: הצהרה שהפונקציה אינה משנה את הפרמטר
- סינטקס: שמים את המילה const לפני טיפוס הפרמטר:  
`void foo(const MyClass& c)`
- המשתנה יהיה קבוע בתוך הפונקציה, ולא ניתן יהיה לשנותו

# העברת פרמטר כ- const

```
#ifndef __OVEN_H  
#define __OVEN_H
```

```
#include "clock.h"
```

```
class Oven
```

```
{  
private:  
    int temperature;  
    Clock startTime;  
    int minutesToWork;  
  
public:  
    void show();  
    int getTemperature();  
    Clock& getStartTime();  
    int getMinutesToWork();  
  
    void setTemperature(int t);  
    void setStarTime(const Clock& c) {startTime = c;}  
    void setMinutesToWork(int t);  
};  
#endif // __OVEN_H
```

כאשר מעבירים פרמטר לפונקציה שהוא אובייקט תמיד נעביר אותו by ref מטעמי יעילות, ונציין שהוא const, כדי שיהיה ברור לקורא הקוד שהפרמטר הועבר by ref מטעמי יעילות, ולא על-מנת לשנותו

הגנה על הפרמטר מפני שינויים בתוך הפונקציה: מועבר by ref מטעמי יעילות – לחסוך את ההעתקה, ולכן ישנו ה- const כדי ליידע את הקורא שמשתנה זה אינו ניתן לשינוי בפונקציה

# משתנים/פרמטרים שהם const

- לא ניתן לשנות את ערכיו של משתנה שהוגדר כ- const
- הקומפיילר לא מאפשר להפעיל אף שיטה על משתנה שהוא const:

```
void main()  
{  
    Clock c1;  
    c1.show();  
  
    const Clock c2;  
    c2.show();  
}
```

the object has type qualifiers that are not compatible with the member function "Clock::show"

- הפתרון: הגדרת השיטה כ- const

# שיטות שהן const

```
class Clock
{
public:
    enum eDisplayType {FULL_DAY, HALF_DAY};

    int getMinutes() const;
    int getHours() const;
    eDisplayType getDisplayType() const;

    bool setMinutes(int m=0);
    bool setHours(int h=0);
    void setDisplayType(eDisplayType type);

    void show() const;
    void tick();
    void addMinutes(int add);

private:
    int hours, minutes;
    eDisplayType displayType;
};
```

כאשר שיטה אינה משנה את ערכי  
תכונות האובייקט, נצהיר עליה כ- const

שימו לב: הקומפיילר אינו מתריע על אי הגדרת  
const, אך זהו תכנות נכון, מעין "חוזזה" בין מי  
שכותב את המחלקה למי שמשתמש בה, ולכן  
יש להקפיד על שימוש נכון ב- const

# const הוא חלק מחתימת השיטה

- 2 שיטות בעלות שם זהה ורשימת פרמטרים זהה, יכולות להיבדל אחת מהשניה ב-const (functions overloading):

```
class MyClass
{
public:
    void foo()      {cout << "In foo()\n";}
    void foo() const {cout << "In foo() const\n";}
};
```

- במקרה זה, כאשר יש משתנה רגיל ומשתנה const כל אחד יפנה לשיטה המתאימה

```
void main()
{
    MyClass c1;
    const MyClass c2;

    c1.foo();
    c2.foo();
}
```

```
In foo()
In foo() const
```

- האם ניתן היה לוותר על אחת מ-2 הגרסאות?

```
class MyClass
{
public:
    void foo()          {cout << "In foo()\n";}
    void foo() const {cout << "In foo() const\n";}
};
```

- כן, הגרסא בלי ה-const
- משתנה שהוא const יכול להפעיל אך ורק שיטה שהיא const
- משתנה רגיל יכול להפעיל כל שיטה



# האם הקוד הבא מתקמפל? אם כן, מה הפלט, אחרת מהי השגיאה?

```
class Inner
{
public:
    void foo() {cout << "In Inner::foo\n";}
};

class Outer
{
    Inner i;
public:
    const Inner& getInner() const {return i;}
};

void main()
{
    Outer o;
    o.getInner().foo();
}
```

התיקון יהיה להגדיר את השיטה foo כ- const

הקוד אינו מתקמפל מאחר ו- getInner מחזירה אובייקט שהוא const ולכן ניתן להפעיל עליו רק שיטות שהוגדרו כ- const

Outer o; 'Inner::foo' : cannot convert 'this' pointer from 'const Inner' to 'Inner &'  
o.getInner().foo();

# constexpr

---

- ידוע לנו כי ערך החוזר פונקציה מחושב בזמן ריצה
- במידה וכל הערכים שבשימוש הפונקציה ידועים בזמן קומפילציה, ניתן לגרום לפונקציה לחשב את הערך המוחזר כבר בזמן קומפילציה ובכך לחסוך בביצועים בזמן ריצה

```
constexpr double getCircleArea(int radius)
{
    return 3.14 * radius * radius;
}

void main()
{
    const int RADIUS = 5;
    cout << "circle area with radius=" << RADIUS
          << " is " << getCircleArea(RADIUS) << endl;

    int radius;
    cout << "Enter radius: ";
    cin >> radius;
    cout << "circle area with radius=" << radius
          << " is " << getCircleArea(radius) << endl;
}
```

מאחר ו-RADIUS הוא const, הקומפיילר מחשב את הערך המוחזר מהפונקציה כבר בזמן קומפילציה

מאחר וערכו של radius ידוע רק בזמן ריצה, הפונקציה מתנהגת רגיל ומחושבת בזמן ריצה

# פונקציות constexpr

- כאשר הקומפיילר מתייחס להגדרת ה- `constexpr` הפונקציה הופכת להיות כמו מאקרו
- מתודות המוגדרות כ- `constexpr` אינן יכולות להיות וירטואליות (רלוונטי לפרק של פולימורפיזם)
- ישנה הגבלה ל- `return` יחיד במימוש, אחרת תהיה שגיאת קומפילציה

# כיצד ניתן לדעת שהקומפיילר אכן משתמש בפונקציה כ-constexpr?

```
constexpr double getCircleArea(int radius)
{
    return 3.14 * radius * radius;
}
```

```
void main()
{
```

```
    const int RADIUS = 5;
    cout << "circle area with radius=" << RADIUS
          << " is " << getCircleArea(RADIUS) << endl;
    static_assert(getCircleArea(RADIUS) == (3.14 * 25),
                  "result should be 3.14*5*5");
```

```
    int radius;
    cout << "Enter radius: ";
    cin >> radius;
    cout << "circle area with radius=" << radius
          << " is " << getCircleArea(radius) << endl;
    static_assert(getCircleArea(radius) == (3.14 * 25),
                  "result should be 3.14*5*5");
```

```
}
```

static\_assert משמש לבדיקת ערך הידוע בזמן קומפילציה. במידה והערך אינו ידוע, תוצג שגיאה בזמן קומפילציה.

expression did not evaluate to a constant

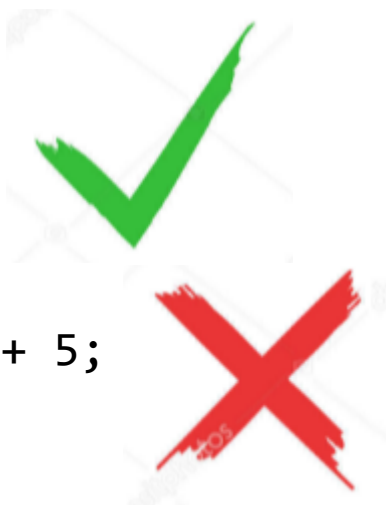
# משתנה constexpr לעומת משתנה const

- ערכו של משתנה const אינו ניתן לעדכון לאחר יצירתו
- ערכו של משתנה constexpr חייב להיות ידוע בזמן קומפילציה, בנוסף לכך שאינו ניתן לעדכון לאחר יצירתו

```
void main()
{
    int x;
    cin >> x;

    const int y1 = x + 5;

    constexpr int y2 = x + 5;
}
```



inline

---

# מתודות inline

- כאשר כותבים מתודה במחלקה, הקפידה אליה מתבצעת בזמן ריצה (זהו המנגנון הרגיל של קריאה לפונקציות)
- קריאות מרובות למתודה יכולות לייצר תקורה בזמן ריצת התוכנית
  - תקורה זו די מינימלית, אבל יש מערכות בהן כל חלקיק שניה משמעותי
- הגדרת המתודה כ- inline תדאג למנוע את הקפידה למתודה ע"י שתילת קוד המתודה במקום המבוקש (הרעיון כמו macro)
- ציון כי מתודה היא inline הינה המלצה בלבד לקומפיילר:
  - הקומפיילר יכול להתעלם ממנה
  - הקומפיילר יכול להחליט על מתודה מסויימת שהיא inline גם אם המתכנת לא הצהיר על כך במפורש



# מתודת inline: דוגמה

```
class Stam
{
private:
    int x;

public:
    inline int getX() const;
};

int Stam::getX() const
{
    return x;
}
```

המימוש של פונקציית inline צריך להיות זמין בזמן הקומפילציה, ולכן ימומש בקובץ ה- h

```
int getX() const {return x;}
```

מתודות שממומשת ב- header הינן עם המלצת inline באופן אוטומטי. פונקציות הכוללות לולאות לא יהיו inline.

# ביחידה זו למדנו:

- enum
- מחלקה המכילה מחלקה אחרת
- בעיית ה- include הכפולים
- העברת פרמטרים by ref
- const
  - פרמטר const
  - שיטת const
  - ערךמשתנה const
  - מוחזר const
- constexpr
- מתודות inline