תכנות מכוון עצמים ו- C++
יחידה 05
init line, contained class, this, const pointer, friend, static

קרן כליף

ביחידה זו נלמד:

- שורת אתחול (init line / init list)
- מעבר ב- c’tor עבור אובייקטים מוכלים
- המצביע this
- מצביע שהוא const
- static:
  - משתנים סטטיים במחלקה
  - שיטות סטטיות
- friend:
  - פונקציית friend
  - מחלקת friend
- תכונות mutable
- מרחבי שמות (namespace)
- ranged based for loop
- auto
- string literals

שורת אתחול (Init Line / Init List)

תזכורת c’tor delegation

- ראינו שקונסטרקטור אחד יכול לקרוא לאחר:

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

#include "clock.h"

Clock::Clock() : Clock(0,0)
{
}

Clock::Clock(int h, int m)
{
    hours = h;
    minutes = m;
}
```

כאשר בקונסטרקטור כותבים פעולות אחרי הנקודותיים, זה נקרא "שורת אתחול"

הקוד בשורת האתחול מתבצע לפני הקוד שבגוף הקונסטרקטור

שורת אתחול (init line / init list)

- ראינו שאת תכונות האובייקט מאתחלים בגוף ה- c'tor

```
Clock::Clock(int h, int m)
{
   hours = h;
   minutes = m;
}
```

- ניתן לאתחל את שדות האובייקט לפני הכניסה לגוף ה- c'tor, בשורת האתחול:

```
Clock::Clock(int h, int m) : hours(h), minutes(m)
{
}
```

- שורת האתחול מבוצעת לפני הפקודות שבגוף ה- c'tor

שורת האתחול יתרון

- עד כה היינו צריכים לתת לפרמטר שם שונה משם התכונה, אחרת הקומפיילר לא היה יודע מתי אנחנו מתכוונים לתכונה ומתי לפרמטר:

```
Clock::Clock(int hours, int minutes)
{
    hours = hours;
    minutes = minutes;
}
```

במקרה זה הקומפיילר מתיחס ל- hours הפרמטר מאחר והוא מוגדר יותר קרוב

- בשימוש ב- init line ניתן לקרוא לפרמטרים עם שם משמעותי, כי הקומפיילר יודע להבחין בתפקיד המשתנה עפ"י מיקומו:

```
Clock::Clock(int hours, int minutes) : hours(hours), minutes(minutes)
{
    //hours = hours;
    //minutes = minutes;
}
```

מחוץ לסוגריים: תכונה

בתוך הסוגריים: פרמטר

אתחול תכונות const ו- ref

- תכונות אלו יכולות לקבל ערך פעם אחת בלבד, עם יצירת האובייקט, כלומר ב- c’tor
- תכונות אלו חובה לאתחל בשורת האתחול!

דוגמה – המחלקות Street ו- Building

- לכל בניין יש מספר קומות, שהוא ערך קבוע
  - ← מאותחל בקונסטרקטור
- לכל בניין יש רפרנס לרחוב בו הוא נמצא
  - ← מאותחל בקונסטרקטור
- לכל בניין יש מספר דיירים וכן לכל רחוב
  - ← עדכון מספר הדיירים בבניין יגרור עדכון מספר הדיירים ברחוב

street.h

```
#ifndef __STREET_H
#define __STREET_H

class Street
{
private:
    char streetName[20];
    char city[20];
    int  numOfTenants;

public:
    Street(const char* streetName, const char* city);

    void setStreetName(const char* s)    { strcpy(streetName, s); }
    void setCity(const char* c)          { strcpy(city, c); }
    void updateNumOfTenants(int num)     { numOfTenants += num; }

    const char* getStreetName() const { return street; }
    const char* getCity()       const { return city; }
    int getNumOfTenants()       const { return numOfTenants; }

    void show() const;
};
#endif // __STREET_H
```

street.cpp

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

#include "street.h"


Street::Street(const char* streetName, const char* city)
{
    numOfTenants = 0;
    setCity(city);
    setStreetName(streetName);
}

void Street::show() const
{
    cout << streetName << ".st " << city
                        << ", " << numOfTenants << " tenants\n";
}
```

building.h

```
#ifndef __BUILDING_H
#define __BUILDING_H

class Street;
#include "street.h"

class Building
{
    Street& theStreet;
    const int numOfFloors;
    int numOfTenants;

public:
    Building(/*const*/ Street& street, int floors);

    const Street& getStreet()       const { return theStreet; }
    int           getNumOfTenants() const { return numOfTenants; }

    void updateNumOfTenants(int num);

    void show() const;
};
#endif // __BUILDING_H
```

מאחר ורק שומרים רפרנס למשתנה מסוג זה, ואין פניה לתכונות ולשיטות, ניתן להסתפק ב- forward declaration

מאחר ושומרים רפרנס לפרמטר זה, והרפרנס אינו const, גם הפרמטר אינו יכול להיות const

building.cpp

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

#include "building.h"
#include "street.h"

Building::Building(/*const*/ Street& street, int floors)
        : theStreet(street), numOfFloors(floors), numOfTenants(0)
{
    //numOfFloors = floors;
}

void Building::updateNumOfTenants(int num)
 {
    numOfTenants += num;
    theStreet.updateNumOfTenants(num);
}

void Building::show() const
{
    cout << "Building with " << numOfFloors << " floors, "
                << numOfTenants << " tenants, at ";
    theStreet.show();
}
```

תכונה שהיא רפרנס יש לאתחול בשורת האתחול

תכונה שהיא const יש לאתחול בשורת האתחול

ניסיון אתחול תכונה שהיא const שלא בשורת האתחול יגרור שגיאת קומפילציה

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

#include "street.h"
#include "building.h"

int main()
{
    Street s("Dizengof", "TA");
    Building b1(s, 5);
    Building b2(s, 6);

    b1.show();
    b2.show();
    s.show();
    cout << endl;

    b1.updateNumOfTenants(12);
    s.setStreetName("Dizengoff");

    b1.show();
    b2.show();
    s.show();
}
```

```
Building with 5 floors, 0 tenants at
Dizengof.st Tel-Aviv, 0 tenants
Building with 6 floors,0 tenants at
Dizengof.st Tel-Aviv, 0 tenants
Dizengof.st Tel-Aviv, 0 tenants

Building with 5 floors, 12 tenants at
Dizengoff.st Tel-Aviv, 12 tenants
Building with 6 floors, 0 tenants at
Dizengoff.st Tel-Aviv, 12 tenants
Dizengoff.st Tel-Aviv, 12 tenants
```

Street s	char street[20]	Dizengof	1000
	char city[20]	TA	1020
	int numOfTenants	0	1040
Building b1	Street& theStreet		
	const int numOfFloors	5	1044
	int numOfTenants	0	1048
Building b2	Street& theStreet		
	const int numOfFloors	6	1052
	int numOfTenants	0	1056

שורת אתחול דגשים תחביריים

- אם מפרידים בין המימושים לבין הגדרת המחלקה, אזי כתיבת שורת האתחול תהייה רק במימוש, ולא בהגדרה
- שורת האתחול היא כלי לשימוש בבנאים בלבד

חוקי קונסטרקטורים בהכלת מחלקות

מחלקה המכילה מחלקה אחרת

- כאשר מחלקה מכילה מחלקה אחרת, האובייקט המוכל נולד לפני האובייקט המכיל
- סדר הריסת האובייקטים הפוך לסדר היצירה:
  - קודם נהרס האובייקט המכיל ורק אז האובייקט המוכל

- מבחינת תחביר במחלקה המכילה:
  - אתחול מתבצע ב- c’tor, לכן יש מעבר ב- c’tor של האובייקט המוכל לפני כניסה לגוף ה- c’tor של האובייקט המכיל
    - אתחול זה מבוצע ב- init line, המבוצע לפני הכניסה לגוף ה- c’tor
  - במקרה ולא אתחלנו במפורש בשורת האתחול את האובייקט המוכל:
    - יהיה ניסיון לאתחלו דרך ה- default c’tor
    - במידה ואינו קיים תתקבל שגיאת קומפילציה

הכלת מחלקות דוגמא 1

```
class Inner
{
     int x;
public:
    Inner(int x) : x(x) { cout << "Inner::Inner x=" << x << endl; }
    ~Inner()            { cout << "Inner::~Inner x=" << x << endl; }
};
class Outer
{
    Inner i;
public:
    Outer() : i(4)          { cout << "Outer::Outer\n"; }
    Outer(int x) : i(x)     { cout << "Outer::Outer(int)\n"; }
    ~Outer()                { cout << "Outer::~Outer\n"; }
};


void main()
{
    Outer o1;
    cout << "------------\n";
    Outer o2(5);
    cout << "------------\n";
}
```

אוביקט מוכל

הפעלת ה- c'tor של האובייקט המוכל בשורת האתחול

```
Inner::Inner x=4
Outer::Outer
------------
Inner::Inner x=5
Outer::Outer(int)
------------
Outer::~Outer
Inner::~Inner x=5
Outer::~Outer
Inner::~Inner x=4
```

הכלת מחלקות דוגמא 2

'Inner' : no appropriate default constructor available

```
class Inner
{
     int x;
public:
    Inner(int x) : x(x) { cout << "Inner::Inner x=" << x << endl; }
    ~Inner()            { cout << "Inner::~Inner x=" << x << endl; }
};
class Outer
{
    Inner i;
public:
    Outer() /*: i(4) */ { cout << "Outer::Outer\n"; }
    Outer(int x) : i(x) { cout << "Outer::Outer(int)\n"; }
    ~Outer()            { cout << "Outer::~Outer\n"; }
};
```

מאחר ואין פניה מפורשת לבנאי של האובייקט המוכל, הקומפיילר מחפש בו default c'tor. מאחר ולא קיים מתקבלת שגיאת קומפילציה

הכלת מחלקות דוגמא 3

מתן ערך deafult הופך בנאי זה להיות גם default c'tor

לפני הכניסה לגוף ה- c'tor תתבצע כניסה ל- default c'tor של אובייקט המוכל

```
class Inner
{
    int x;
public:
    Inner(int x=3) : x(x) { cout << "Inner::Inner x=" << x << endl; }
    ~Inner()              { cout << "Inner::~Inner x=" << x << endl; }
};
class Outer
{
    Inner i;
public:
    Outer() /*: i(4) */ { cout << "Outer::Outer\n"; }
    Outer(int x) : i(x) { cout << "Outer::Outer(int)\n"; }
    ~Outer()            { cout << "Outer::~Outer\n"; }
};

void main()
{
    Outer o1;
    cout << "------------\n";
    Outer o2(5);
    cout << "------------\n";
}
```

```
Inner::Inner x=3
Outer::Outer
------------
Inner::Inner x=5
Outer::Outer(int)
------------
Outer::~Outer
Inner::~Inner x=5
Outer::~Outer
Inner::~Inner x=3
```

סדר האתחול בשורת האתחול

```
class Inner
{
    int x;
public:
    Inner(int x) : x(x) { cout << "Inner::Inner x=" << x << endl; }
    ~Inner() { cout << "Inner::~Inner x=" << x << endl; }
};
class Outer
{
    Inner i1;
    Inner i2;
public:
    Outer() : i2(13), i1(7) { cout << "Outer::Outer\n"; }
    ~Outer() { cout << "Outer::~Outer\n"; }
};
void main()
{
    Outer o;
    cout << "------------\n";
}
```

אמנם בשורת האתחול אנו
מאתחלים קודם את i2 אבל
בפועל יאותחל קודם i1

```
Inner::Inner x=7
Inner::Inner x=13
Outer::Outer
------------
Outer::~Outer
Inner::~Inner x=13
Inner::~Inner x=7
```

המעבר ב- copy c’tor של האובייקט המוכל

- ה- copy c’tor שמקבלים במתנה מהקומפיילר מפעיל את ה- copy c’tor של האובייקט המוכל לפני כניסה לגוף ה- c’tor של האובייקט המכיל

- במידה ואנחנו דורסים את ה- copy c’tor יש לזכור להפעיל c’tor כלשהו של האובייקט המוכל

מעבר ב- copy c'tor של האובייקט המוכל דוגמה

```
class Inner
{
public:
    Inner()                    { cout << "Inner::Inner\n"; }
    Inner(const Inner& other) { cout << "Inner::Inner(copy)\n"; }
    ~Inner()                   { cout << "Inner::~Inner\n"; }
};

class Outer
{
    Inner i;
public:
    Outer()  { cout << "Outer::Outer\n"; }
    ~Outer() { cout << "Outer::~Outer\n"; }
};

void main()
{
    Outer o1;
    cout << "------------\n";
    Outer o2(o1);
    cout << "------------\n";
}
```

מראה שה- copy c'tor
שמקבלים במתנה עבור Outer,
מפעיל את ה- copy c'tor של
האובייקט המוכל Inner

```
Inner::Inner
Outer::Outer
------------
Inner::Inner(copy)
------------
Outer::~Outer
Inner::~Inner
Outer::~Outer
Inner::~Inner
```

יצירת אובייקט
דרך copy c'tor

דריסת ה- copy c'tor

```
class Inner
{
public:
    Inner()                          { cout << "Inner::Inner\n"; }
    Inner(const Inner& other)        { cout << "Inner::Inner(copy)\n"; }
    ~Inner()                         { cout << "Inner::~Inner\n"; }
};
class Outer
{
    Inner i;
public:
    Outer()  { cout << "Outer::Outer\n"; }
    Outer(const Outer&) { cout << "Outer::Outer(copy)\n"; }
    ~Outer() { cout << "Outer::~Outer\n"; }
};
void main()
{
    Outer o1;
    cout << "------------\n";
    Outer o2(o1);
    cout << "------------\n";
}
```

מעבר ב- default c'tor
של האובייקט המוכל

```
Inner::Inner
Outer::Outer
------------
Inner::Inner
Outer::Outer(copy)
------------
Outer::~Outer
Inner::~Inner
Outer::~Outer
Inner::~Inner
```

דריסת ה- copy c'tor גרסה משופרת

```
class Inner
{
    int x;
public:
    Inner()                    { cout << "Inner::Inner\n"; }
    Inner(const Inner& other) { cout << "Inner::Inner(copy)\n"; }
    ~Inner()                   { cout << "Inner::~Inner\n"; }
};
class Outer
{
    Inner i;
public:
    Outer()  { cout << "Outer::Outer\n"; }
    Outer(const Outer& other) : i(other.i) { cout << "Outer::Outer(copy)
    ~Outer() { cout << "Outer::~Outer\n"; }
};
void main()
{
    Outer o1;
    cout << "------------\n";
    Outer o2(o1);
    cout << "------------\n";
}
```

הפעלה של ה- copy c'tor
של האובייקט המוכל

```
Inner::Inner
Outer::Outer
------------
Inner::Inner(copy)
Outer::Outer(copy)
------------
Outer::~Outer
Inner::~Inner
Outer::~Outer
Inner::~Inner
```

המצביע this

שימוש ב- this למתן שמות משמעותיים לפרמטרים

```
void Clock::setHour(int hours)
{
    hours = hours;
}
```

פניה לפרמטר בשני צידי ההשמה

```
void Clock::setHour(int hours)
{
    this->hours = hours;
}
```

פניה לתכונה

פניה לפרמטר

במקרה של פרמטר עם שם זהה לתכונה, לפרמטר יש עדיפות בתוך השיטה, ולכן צריך להקפיד לקרוא לתכונה דרך המצביע this

- ואפשר גם ב- c'tor:

```
Clock::Clock(int hours, int minutes)
{
    this->hours = hours;
    this->minutes = minutes;
}
```

המצביע this

- כאשר אנחנו פונים לאוביקט שהוגדר ב- main יש לו שם, ובאמצעות "." או "->" אנו פונים לתכונותיו ולפעולותיו
- כאשר אנחנו בקוד בשיטה בתוך המחלקה, שם האובייקט אינו ידוע לשיטה
- תיתכן בעיה כאשר השיטה תרצה להתייחס לאובייקט בכללותו, ולא רק לאחת מתכונותיו, מאחר ואינה יודעת את שמו
  - למשל אם תרצה לשלוח את אוביקט לשיטה אחרת
- <u>הפתרון:</u> שימוש במצביע **this**, שזו מילה שמורה בשפה שמשמעותה פניה לאוביקט המפעיל
- בעזרת המצביע this נוכל לתת שמות משמעותיים למשתנים בשיטות

אז מיהו בעצם this?

```
Clock::Clock(int hours, int minutes)
{
    this->hours = hours;
    this->minutes = minutes;
}

void Clock::show() const
{
    cout << (this->hours < 10 ? "0" : "")
         << hours << ":"
         << (minutes < 10 ? "0" : "")
         << minutes;
}
```

ביצירת האובייקט הראשון
this מתייחס ל- c1

ביצירת האובייקט
השני this מתייחס ל- c2

בהפעלה זו this מתייחס ל- c1, וישנו
שימוש ב- this למרות שלא חייבים

```
void main()
{
    Clock c1(10, 15);
    Clock c2(20, 30);

    c1.show();
    c2.show();
```

לסיכום:
this מתייחס לאובייקט שהפעיל את המתודה

דוגמת שימוש ב- this כייצוג האובייקט: קבוצת כדורסל ומאמן

```
#ifndef __TEAM_H
#define __TEAM_H

class Coach;

class Team
{
private:
    char name[20];
    Coach* theCoach;

public:
    Team(const char* name);

    const char* getName() const {return name;}
    Coach* getCoach()           {return theCoach;}

    void setCoach(Coach* newCoach);

    void show() const;
};
#endif // __TEAM_H
```

– Forward declaration
למניעת include דו-כיווני

team.h

coach.h

```
#ifndef __COACH_H
#define __COACH_H

class Team;

class Coach
{
private:
    char name[20];
    Team* theTeam;

public:
    Coach(const char* name, Team* newTeam = nullptr);

    const char* getName() const {return name;}
    void show() const;
    void setTeam(Team* newTeam);
};
#endif // __COACH_H
```

team.cpp

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

#include <string.h>

#include "coach.h"
#include "team.h"

Team::Team(const char* name)
{
    strcpy(this->name, name);
    theCoach = nullptr;
}

void Team::show() const
{
    cout << name;

    if (theCoach == nullptr)
        cout << " doesn't have a coach";
    else
        cout << "'s coach is " << theCoach->getName();
    cout << endl;
}
```

```cpp
void Team::setCoach(Coach* newCoach)
{
    ...
}
```

coach.cpp

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

#include <string.h>

#include "coach.h"
#include "team.h"

Coach::Coach(const char* name, Team* newTeam)
{
    strcpy(this->name, name);
    setTeam(newTeam);
}

void Coach::show() const
{
    cout << name << " ";

    if (theTeam == nullptr)
        cout << "doesn't coach any team now";
    else
        cout << "coaches the team " << theTeam->getName();
    cout << endl;
}
```

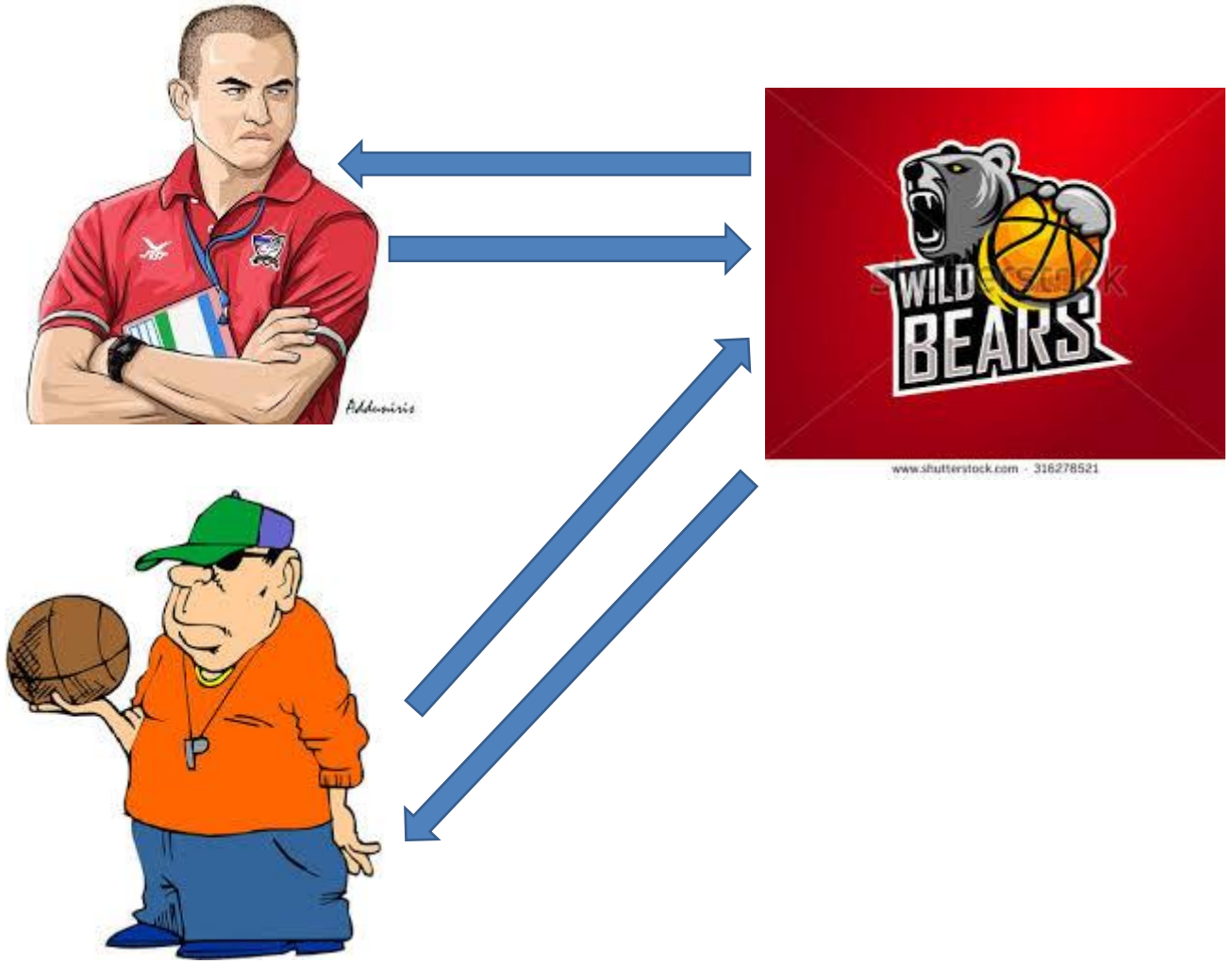
```
void Coach::setTeam(Team* newTeam)
{
    ...
}
```

מה קורה במקרים הבאים?

- כאשר מעדכנים לקבוצה מאמן חדש:
  - במידה והיה קיים לקבוצה מאמן אחר, יש לדאוג "לפטר" אותו
    - ← עדכון שהקבוצה שעכשיו הוא מאמן היא nullptr
  - שיוך המאמן החדש לקבוצה
  - יידוע המאמן החדש שהוא משויך לקבוצה
- כאשר מעדכנים למאמן קבוצה חדשה:
  - יש לעדכן את שדה הקבוצה במאמן
  - יש לעדכן את הקבוצה החדשה שזהו המאמן שלה
- נשים לב שתהליך זה הוא אינסופי ויש לדאוג בקוד לסיים אותו

כאשר למאמן אין קבוצה ולקבוצה אין מאמן דוגמה

כאשר למאמן אין קבוצה ולקבוצה יש מאמן דוגמה



team.cpp

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

#include <string.h>

#include "coach.h"
#include "team.h"

Team::Team(const char* name)
{
    …
}

void Team::show() const
{
    …
}
```

האפשרויות בפרמטר newCoach:

1. nullptr כאינדיקציה לכך שאין מאמן
2. מצביע למאמן הנוכחי
3. מצביע למאמן חדש

```
void Team::setCoach(Coach* newCoach)
{
    if (theCoach != newCoach)
    {
        if (theCoach != nullptr)
           theCoach->setTeam(nullptr);

        theCoach = newCoach;
        if (theCoach != nullptr)
           theCoach->setTeam(this);
    }
}
```

נבדוק שהמאמן החדש אינו המאמן הנוכחי (בדיקת כתובות), כדי למנוע את הלולאה האינסופית

coach.cpp

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

#include <string.h>

#include "coach.h"
#include "team.h"

Coach::Coach(const char* name, Team* newTeam)
{
    ...
}

void Coach::show() const
{
    ...
}
```

```
void Coach::setTeam(Team* newTeam)
{
    if (theTeam != newTeam)
    {
        theTeam = newTeam;
        if (theTeam != nullptr)
            theTeam->setCoach(this);
    }
}
```

נבדוק שהקבוצה החדשה אינה הקבוצה הנוכחית (בדיקת כתובות), כדי למנוע את הלולאה האינסופית

עדכון הקבוצה החדשה ושיוך המאמן עבור הקבוצה

main.cpp

```
#include "team.h"
#include "coach.h"

void main()
{
    Team t1("MACCABI");
    Coach c1("GOGO");
    Coach c2("MOMO");

    t1.show();
    c1.show();
    c2.show();

    t1.setCoach(&c1);
    t1.show();
    c1.show();
    c2.show();

    c2.setTeam(&t1);
    t1.show();
    c1.show();
    c2.show();
}
```

```
MACCABI doesn't have a coach
GOGO doesn't coach any team now
MOMO doesn't coach any team now

MACCABI's coach is GOGO
GOGO coaches the team MACCABI
MOMO doesn't coach any team now

MACCABI's coach is MOMO
GOGO doesn't coach any team now
MOMO coaches the team MACCABI
```

מצביעי const

תזכורת: מצביעים שהם const

```
void main()
{
    int x = 4, y;
    const int* p1 = &x;
    int* const p2 = &x;

  ✓ x = 5;
  ✗ *p1 = 6;
  ✓ p1 = &y;

  ✓ *p2 = 7;
  ✗ p2 = &y;
}
```

int: x	7	1000
int: y	???	1004
const int*: p1	1004	1008
int* const: p2	1000	1012

הזיכרון של ה- main

const משמאל לטיפוס מגן על תוכן ההצבעה ואינו מאפשר לשנות את התוכן אליו מצביעים

const מימין לטיפוס מגן על הכתובת ואינו מאפשר להכיל כתובת של משתנה אחר לאחר האתחול

מצביע const במחלקה

```
class Nanny
{
    char name[20];
public:
    Nanny(const char* name) { setName(name); }
    void setName(const char* name) { strcpy(this->name, name); }
};

class Wall
{
    char color[20];
public:
    Wall(const char* color) { setColor(color); }
    void setColor(const char* color) { strcpy(this->color, color); }
};

class Kindergarten
{
    const Nanny* theNanny;
    Wall* const theWall;
public:
    ...
};
```

ניתן לשנות ולהצביע למטפלת אחרת, אבל הגן אינו יכול לשנות את ערכי תכונות המטפלת

לא ניתן להצביע לקיר אחר, אך הגן כן יכול לשנות את ערכי תכונות הקיר

מצביע const במחלקה (המשך)

```
class Kindergarten
{
    const Nanny* theNanny;
    Wall* const theWall;
public:
    Kindergarten(const Nanny* theNanny, Wall* const theWall)
                 : theWall(theWall)
    {
        this->theNanny = theNanny;
    }

    void testChangePointers(Nanny* newNanny, Wall* newWall)
    {
        theNanny = newNanny;
        theWall = newWall;
    }

    void testChangeValues(const char* newNannyName,
                          const char* newWallColor)
    {
        theNanny->setName(newNannyName);
        theWall->setColor(newWallColor);
    }
};
```

תכונה מטיפוס כתובת שלא ניתן לשנות את הצבעתה חייבת להיות מאותחלת בשורת האתחול

לא ניתן לשנות ולהצביע לקיר אחר (לא ניתן להחליף פיזית את הקיר של הגן)

לא ניתן לשנות את נתוני הגננת (ניתן להפעיל עליה רק שיטות const)

דוגמאות סיכום חשובות

הכלת מחלקות

תכונה שהיא מצביע const

מתי נחזיק תכונה כמצביע ופרמטר כ- by ref

מתודות שמחזירות ref

העמסת מתודה כ- const

איברי מערך דינאמי אינם אובייקטים מוכלים

דוגמה כתובת הסטודנט ובית הספר

address.h

```
#ifndef __ADDRESS_H
#define __ADDRESS_H

class Address
{
private:
    char street[20];
    int houseNumber;
    char city[20];

public:
    Address(const char* street, int houseNumber, const char* city);

    void setStreet(const char* street) {strcpy(this->street, street);}
    void setCity(const char* city)     {strcpy(this->city, city);}
    void setNumber(int houseNumber)    {this->houseNumber=houseNumber;}

    const char* getStreet() const {return street;}
    const char* getCity()   const {return city;}
    int getNumber()         const {return houseNumber;}

    void show() const;
};
#endif // __ADDRESS_H
```

address.cpp

```
Address::Address(const char* street, int houseNumber, const char* city)
{
    setNumber(houseNumber);
    setCity(city);
    setStreet(street);
}

void Address::show() const
{
    cout << houseNumber << " " << street << ".st " << city << endl;
}
```

student.h

לכל סטודנט יהיה מצביע לבית-הספר והעתק של כתובת הבית שלו: שינוי בנתוני ביה"ס ב- main ישפיע גם על הסטודנט, בעוד ששינוי אובייקט הכתובת, לא ישנה את נתוני אובייקט הסטודנט, מאחר והוא מחזיק העתק. שינוי כתובת הבית של סטודנט תתאפשר אך ורק דרך פניה לסטודנט.

```
#ifndef __STUDENT_H
#define __STUDENT_H

#include "address.h"

class Student
{
private:
    char*           name;
    const Address*  schoolAddress;
    Address         homeAddress;
public:
    Student(const char* name, const Address& school, const Address& home);
    Student(const Student& other);
    ~Student();

    const Address& getHomeAddress() const {return homeAddress;}
    Address&       getHomeAddress()       {return homeAddress;}

    void show() const;
};
#endif // __STUDENT_H
```

החזקת מצביע, ולא העתק. לא ניתן לשנות את תוכן הצבעה!

כדי לאפשר קבלת האובייקט המוכל לשינויים

student.cpp

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

#include "student.h"

Student::Student(const char* name, const Address& school, const Address& home)
        : homeAddress(home)
{
    this->name = new char[strlen(name)+1];
    strcpy(this->name, name);
    schoolAddress = &school;
}

Student::Student(const Student& other)
        : homeAddress(other.homeAddress)
{
    name = new char[strlen(other.name)+1];
    strcpy(name, other.name);
    schoolAddress = other.schoolAddress;
}

Student::~Student()
{
    delete []name;
}
```

העתק, לכן מעבר ב- copy c'tor

רק מצביע

העתק, לכן מעבר ב- copy c'tor

רק מצביע

מעבר ב- d'tor של Street עבור homeAddress

```
void Student::show() const
{
    cout << name << " lives at ";
    homeAddress.show();
    cout << "and studies at ";
    schoolAddress->show();
}
```

אובייקט ולכן הפעלת שיטה ע"י נקודה

מצביע ולכן הפעלת שיטה ע"י חץ

main.cpp

12 Bavli, Jerusalem

12 Toledano, Holon

10 Yerushalmi, T.A.

12 Toledano, Holon

```
void main()
{
    Address school("Herzog", 10, "Tel-Aviv");
    Address home("Bavli", 12, "Tel-Aviv");

    Student s1("momo", school, home);
    home.setStreet("Toledano");
    home.setCity("Holon");

    Student s2("gogo", school, home);

    school.setStreet("Yerushalmi");


    s1.getHomeAddress().setCity("Jerusalem");
    cout << "------------------------\n";
}
```

דוגמה פוליגון

point.h

```
#ifndef __POINT_H
#define __POINT_H

#include <iostream>
using namespace std;

class Point
{
    int x, y;
public:
    Point(int x = 0, int y = 0) : x(x), y(y)
    {
        cout << "In Point::Point ";
        show();
        cout << endl;
    }
    Point(const Point& other) : x(other.x), y(other.y)
                { cout << "In Point::Point(copy)\n"; }
    ~Point()
    {
        cout << "In Point::~Point ";
        show();
        cout << endl;
    }

    void show() const {cout << "(" << x << ", " << y << ") ";}
};
#endif // __POINT_H
```

polygon.h

```
#ifndef __POLYGON_H
#define __POLYGON_H

#include "point.h"

class Polygon
{
    int numOfPoints;
    Point* allPoints;
public:
    Polygon(int numOfPoints)
            : numOfPoints(numOfPoints)
    {
        allPoints = new Point[numOfPoints];
    }

    Polygon(const Polygon& other) : numOfPoints(other.numOfPoints)
    {
        allPoints = new Point[other.numOfPoints];
        for (int i = 0; i < numOfPoints; i++)
           allPoints[i] = other.allPoints[i];
    }

    void setPoint(int index, const Point& p) {allPoints[index] = p;}
    ...
};
#endif // __POLYGON_H
```

```
~Polygon()
{
    cout << "In Polygon::~Polygon\n";
    delete[]allPoints;
}

void show() const
{
    cout << "The polygon has " << numOfPoints << " points:\n";
    for (int i = 0; i < numOfPoints; i++)
        allPoints[i].show();
    cout << endl;
}
```

מעבר numOfPoints
פעמים ב-d'tor של Point

מעבר numOfPoints פעמים ב-
default c'tor של Point

שימוש באופרטור השמה!
נרחיב בהמשך

main.cpp

```
#include "polygon.h"

void main()
{
    Polygon triangle(3);

    triangle.setPoint(0, Point(1, 1));
    triangle.setPoint(1, Point(2, 8));
    triangle.setPoint(2, Point(5, 4));

    triangle.show();
}
```

נשים לב שאין מעבר ב- copy c'tor כי אובייקטי ה- Point כבר נוצרו בעת יצירת המערך.
ישנו מעבר באופרטור השמה, עליו נדבר בהמשך

```
In Point::Point (0, 0)
In Point::Point (0, 0)
In Point::Point (0, 0)
In Point::Point (1, 1)
In Point::~Point (1, 1)
In Point::Point (2, 8)
In Point::~Point (2, 8)
In Point::Point (5, 4)
In Point::~Point (5, 4)
The polygon has 3 points:
(1, 1) (2, 8) (5, 4)
In Polygon::~Polygon
In Point::~Point (5, 4)
In Point::~Point (2, 8)
In Point::~Point (1, 1)
```

const על מערך מצביעים

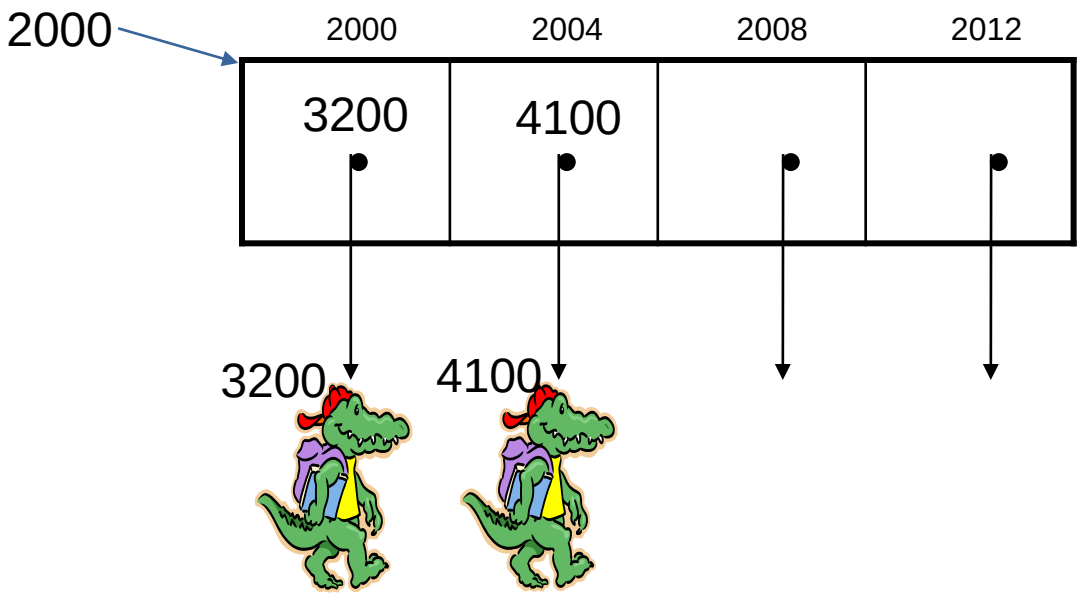
const על מערך מצביעים

- כאשר מחלקה מחזיקה מערך מצביעים, נרצה מתודת get למערך שתגן על חלקיו השונים:

על כתובת ההתחלה (שלא ניתן יהיה להצביע למערך אחר)

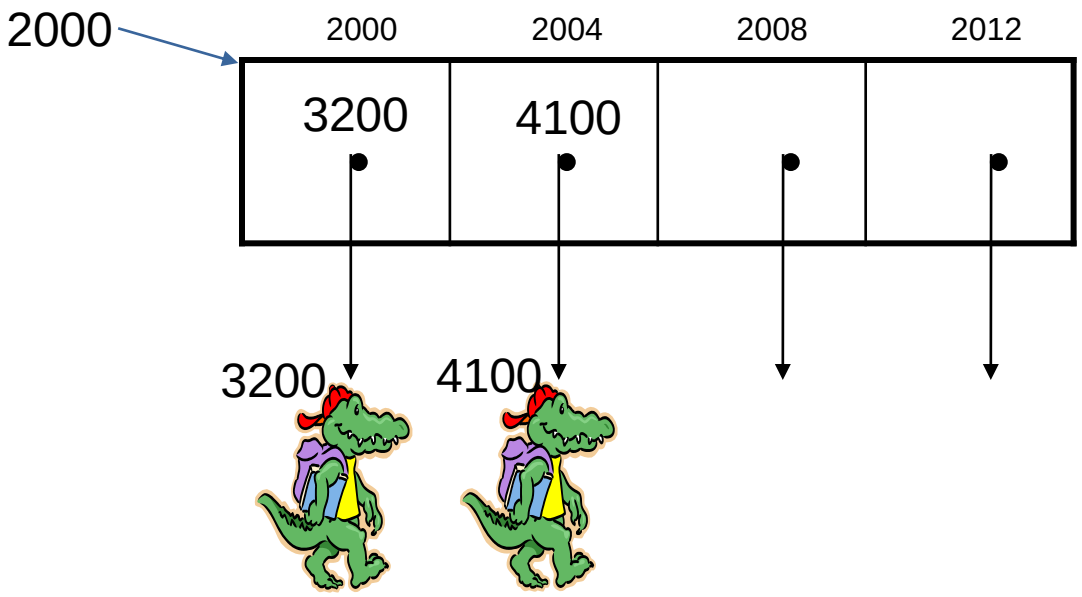
על המצביעים לאובייקטים (שלא ניתן יהיה לשנות את ההצבעות לאובייקטים)

על האובייקטים (שלא ניתן יהיה לשנות את ערכי האובייקטים)



const על מערך מצביעים

הגדרת המערך: `Student** allStudents`



על כתובת ההתחלה (שלא ניתן יהיה להצביע למערך אחר)

`Student**const`

על המצביעים לאובייקטים (שלא ניתן יהיה לשנות את ההצבעות לאובייקטים)

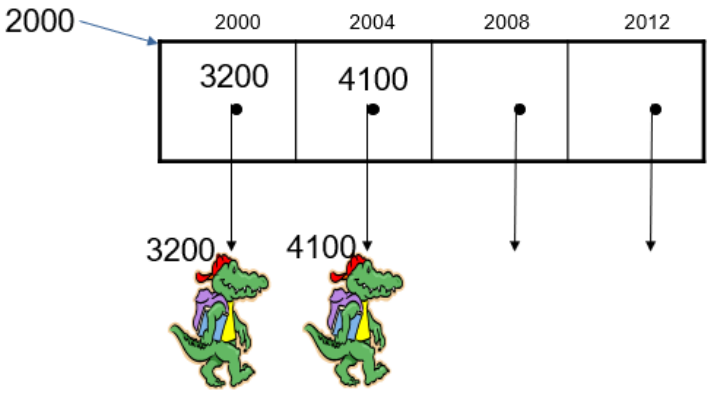
`Student*const*`

על האובייקטים (שלא ניתן יהיה לשנות את ערכי האובייקטים)

`const Student**`

או: `Student const**`

```
class College
{
    Student** allStudents;
    int numOfStudents;
    int maxStudents;

public:
    College(int max) : maxStudents(max), numOfStudents(0)
    {
        allStudents = new Student*[maxStudents];
    }

    int getNumOfStudents() const { return numOfStudents; }
    bool addStudent(Student& newStudent) {...}

          Student** getAllStudents1()  const  { return allStudents; }

    Student const** getAllStudents2()  const  { return allStudents; }
    const Student**

    Student *const* getAllStudents3()  const  { return allStudents; }

    Student**const  getAllStudents4()  const  { return allStudents; }
};
```



במקרה זה הערך המוחזר גם יכול להיכתב כך:
כלומר, ה- const משמאל לכוכביות

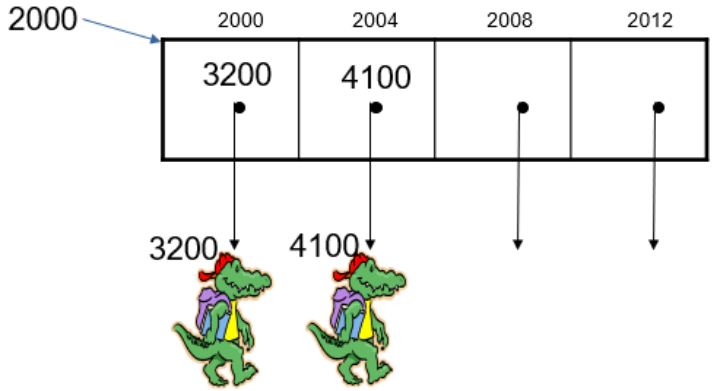
const על מערך מצביעים ה- main

- דוגמה לשימוש במתודה:

```
Student** getAllStudents1()  const  { return allStudents; }
```

ה- const מימין בחתימה על המתודה אינו נאכף, שכן ניתן לשנות את כל 3 חלקי המערך (כתובת התחלה, כתובת האובייקט ותוכן האובייקט)

```
int main()
{
    Student s1("gogo");
    Student s2("momo");

    College c(5);
    c.addStudent(s1);
    c.addStudent(s2);

    Student** arr1 = c.getAllStudents1();
    for (int i = 0; i < c.getNumOfStudents(); i++)
    {
      ✓ arr1[i]->setName("koko");
      ✓ arr1[i] = nullptr;
      ✓ arr1 = nullptr;
    }
}
```



const על מערך מצביעים ה- main

- דוגמה לשימוש במתודה:

```
Student const** getAllStudents2()  const  { return allStudents; }
```

או:

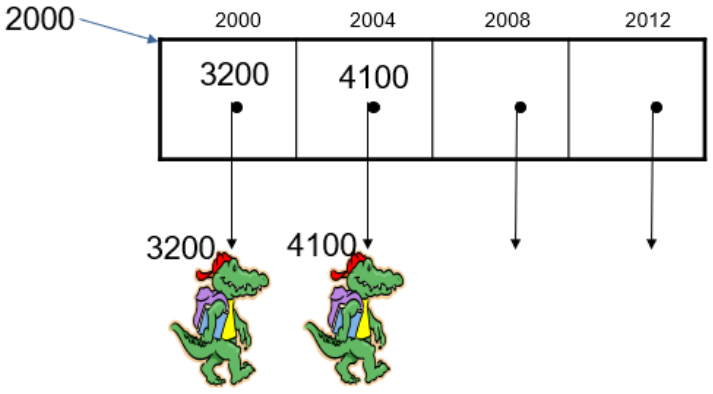
```
const Student**
```

ה- const משמאל לכוכביות
מגן על תוכן האובייקט בלבד

```
int main()
{
    ...

    const Student** arr2  = c.getAllStudents2();
```

או:

```
    Student const**

    for (int i = 0; i < c.getNumOfStudents(); i++)
    {
      ✗ arr2[i]->setName("koko");
      ✓ arr2[i] = nullptr;
      ✓ arr2 = nullptr;
    }
}
```



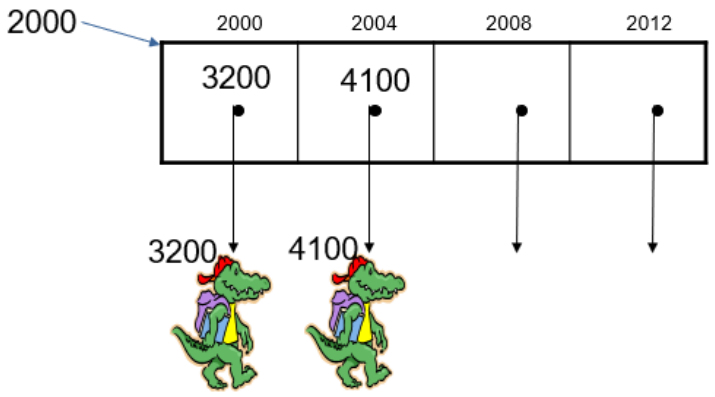
const על מערך מצביעים ה- main

- דוגמה לשימוש במתודה:

```
Student *const* getAllStudents3()  const  { return allStudents; }
```

ה- const בין שתי הכוכביות
מגן על המצביע לאובייקט

```
int main()
{
    ...

    Student*const* arr3 = c.getAllStudents3();
    for (int i = 0; i < c.getNumOfStudents(); i++)
    {
      ✓ arr3[i]->setName("koko");
      ✗ arr3[i] = nullptr;
      ✓ arr3 = nullptr;
    }
}
```



const על מערך מצביעים ה- main

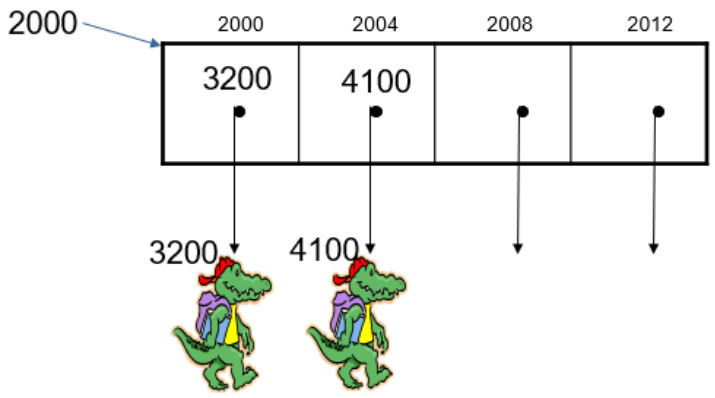
- דוגמה לשימוש במתודה:

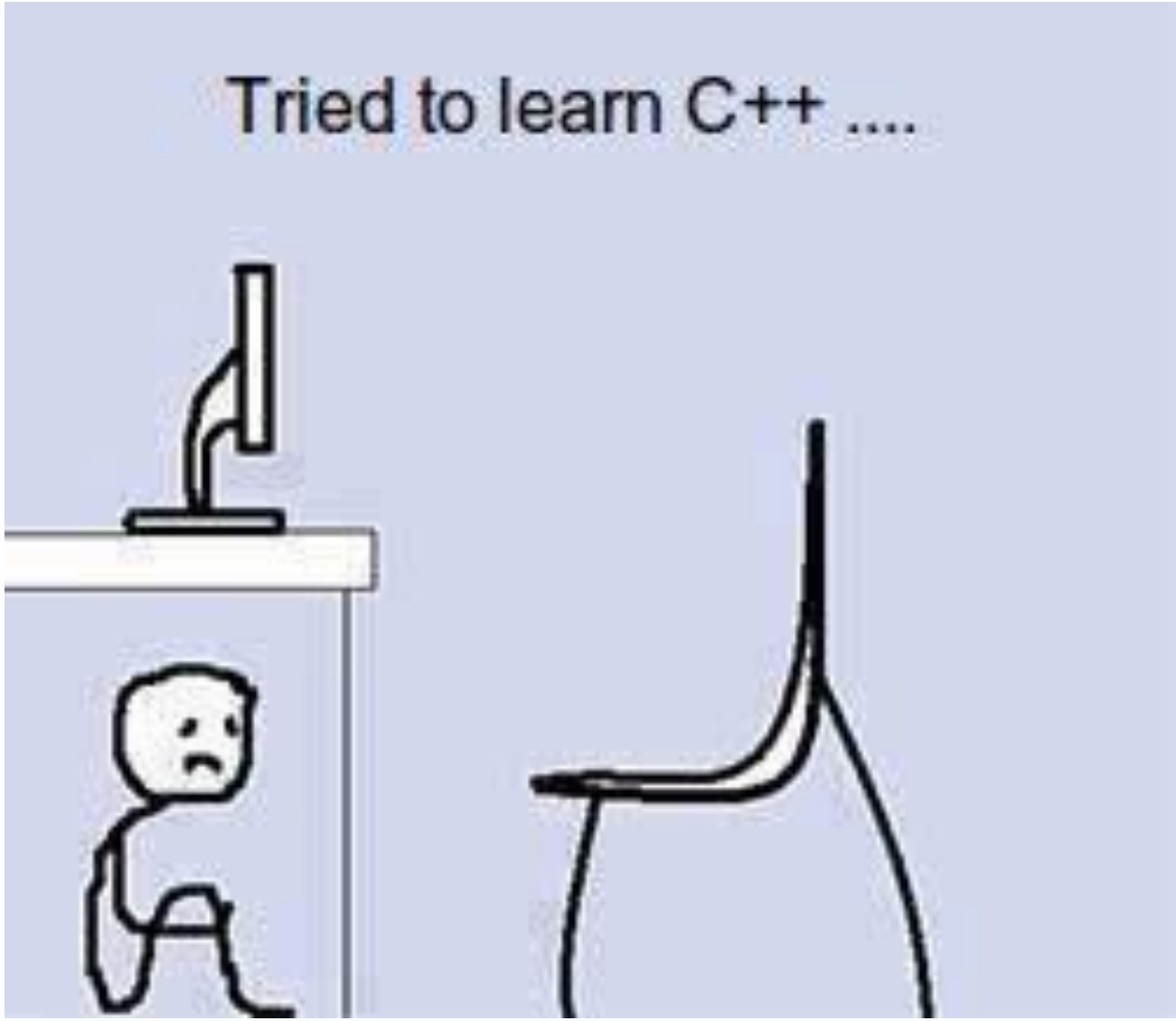
```
Student**const getAllStudents4() const { return allStudents; }
```

ה- const מימין לכוכביות מגן על המצביע למערך

```
int main()
{
    ...

    Student**const arr4 = c.getAllStudents4();
    Student** arr4a = c.getAllStudents4();
    for (int i = 0; i < c.getNumOfStudents(); i++)
    {
      ✓ arr4[i]->setName("koko");
      ✓ arr4[i] = nullptr;
      ✗ arr4 = nullptr;

      ✓ arr4a[i]->setName("koko");
      ✓ arr4a[i] = nullptr;
      ✓ arr4a = nullptr;
    }
}
```

עובר קומפילציה...





https://fbcdn-sphotos-e-a.akamaihd.net/hphotos-ak-ash3/t1/1622691_646133222102226_1217686435_n.jpg

סטטיות

תכונות

הגדרת קבועים במחלקה

שיטות

יצירת מספר סידורי אוטומטי

תכונות סטטיות

תכונת מופע (Instance Attribute)

- עד כה ראינו שתכונה במחלקה משוכפלת עבור כל אובייקט הנוצר מהמחלקה

תכונת מחלקה (תכונה סטטית)

- תכונה שיש עותק אחד שלה עבור כל האובייקטים מהמחלקה
- כל האובייקטים מאותה מחלקה יכולים לקרוא ולשנות תכונה זו
- למשל עבור תכונות שנרצה שערכיהן יהיו זהים לכל האובייקטים
- דוגמאות:
  - מספר האובייקטים שנוצרו ממחלקה מסויימת
  - כל הסטודנטים שנוצרים רוצים לדעת מי הסטודנט המצטיין (זהה לכולם)
- תכונה סטטית קיימת עוד לפני שנוצר אפילו אובייקט אחד מהמחלקה

תכונה סטטית
דוגמה במחלקה Person

```
class Person
{
private:
   static double licenseAge

   char name[20];
   int age;

public:
   Person(const char* name, int age)
   {
      strcpy(this->name, name);
      this->age = age;
   }

   void setLicenseAge(double licenseAge)
   {
      this->licenseAge = licenseAge;
   }

   void show() const
   {
      cout << "Name:  " << name << " (can"
          << (age < licenseAge ? " not" : "")
          << " drive)\n";
   }
}; // class Person

double Person::licenseAge;
```

```
Name:  Gogo (can not drive)
Name:  Momo (can drive)
Name:  Yoyo (can drive)
Changing license age to be 21:
Name:  Gogo (can not drive)
Name:  Momo (can drive)
Name:  Yoyo (can not drive)
```

Person::licenseAge=21

licenseAge =
name="Momo"
age=23

licenseAge =
name="Yoyo"
age=19

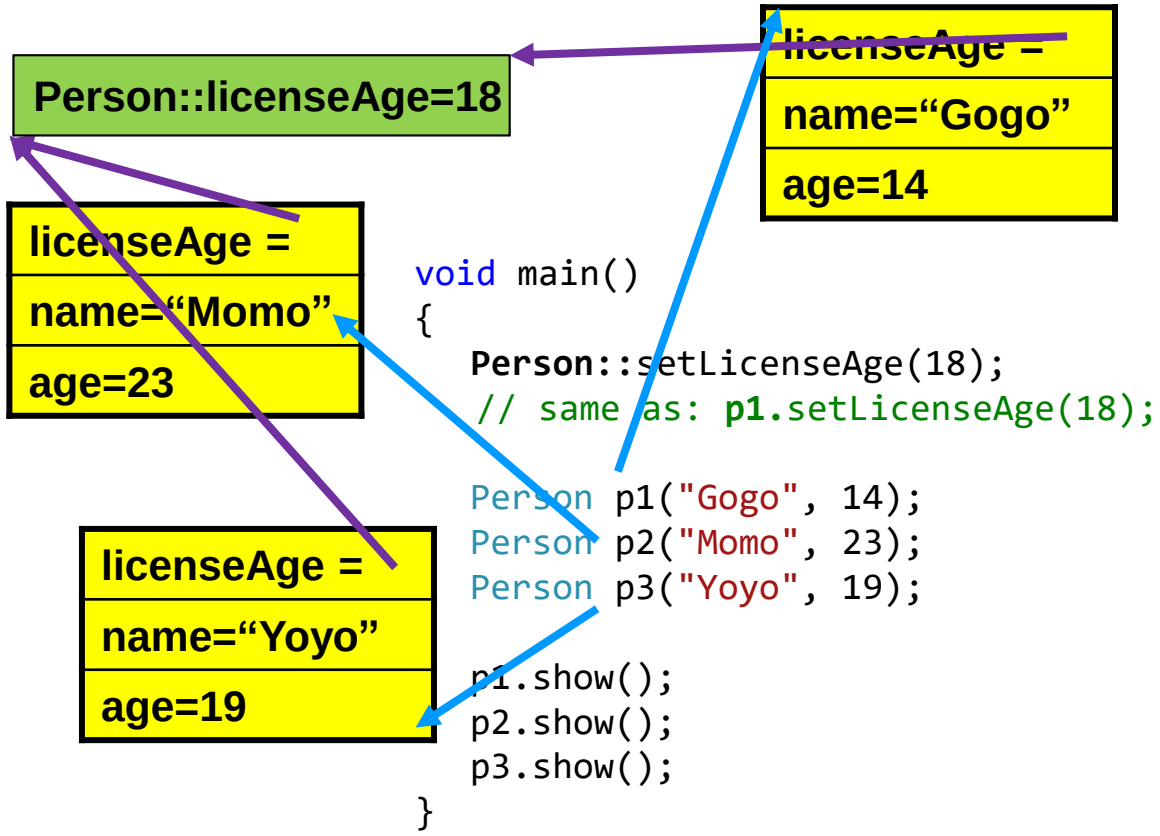
licenseAge =
name="Gogo"
age=14

```
void main()
{
   Person p1("Gogo", 14);
   Person p2("Momo", 23);
   Person p3("Yoyo", 19);
   p1.setLicenseAge(18);
   // same as: p2.setLicenseAge(18);

   p1.show();
   p2.show();
   p3.show();

   cout << "Changing license age to be 21:" << endl;
   p2.setLicenseAge(21);
   // same as: p3.setLicenseAge(21);
   p1.show();
   p2.show();
   p3.show();
}
```

יש להצהיר על המשתנה הסטטי בקובץ cpp ואם אינו מאותחל עם ערך, ערכו ההתחלתי הינו 0, ולא זבל

שיטה סטטית
דוגמה במחלקה Person

```
class Person
{
private:
   static double licenseAge;

   char name[20];
   double age;

public:
   Person(const char* name, double age)
   {
      strcpy(this->name, name);
      this->age = age;
   }

   static void setLicenseAge(double licenseAge)
   {
      this->licenseAge = licenseAge;
      Person::licenseAge = licenseAge;
   }

   void show() const
   {
      cout << "Name:  " << name << " (can"
          << (age < licenseAge ? " not" : "")
          << " drive)\n";
   }
}; // class Person

double Person::licenseAge;
```




```
Name:  Gogo (can not drive)
Name:  Momo (can drive)
Name:  Yoyo (can drive)
```

שיטות סטטיות

- שיטה סטטית היא שיטה הנכתבת בתוך מחלקה, אך אין צורך לייצר אובייקט על מנת להפעיל אותה
- נכתוב שיטה כסטטית במקרה בו אינה מתבססת על נתוניו של אובייקט מסוים, אך קשורה לוגית למחלקה
  - שיטה סטטית יכולה לגשת למשתנים סטטיים, אך לא למשתנים רגילים (משתני מופע), מאחר ואינה מופעלת בהכרח ע"י אוביקט
  - שיטה רגילה יכולה לגשת למשתנים סטטיים
- קריאה לשיטה סטטית מתבצעת באמצעות שם המחלקה או באמצעות אובייקט
  - היתרון: ניתן לקרוא לשיטה עוד לפני שנוצר אפילו אובייקט אחד

- במקרה בו מפרידים את המימוש מההגדרה הציון static יהיה רק בהגדרה

משתנה סטטי כקבוע במחלקה

- יתכן ויהיה במחלקה ערך קבוע לכלל האובייקטים, ולכן נרצה שהוא יהיה חלק מנתוני המחלקה
  - מאחר וקבוע זה משותף לכל האובייקטים, ולא נרצה לשכפל העתק שלו עבור כל אובייקט נגדיר אותו כ- static
  - מאחר והוא קבוע ולא נרצה שישנו את ערכו נגדירו כ- const / constexpr
  - מאחר ולא ניתן לשנות את ערכו ניתן להגדיר קבוע זה ב- public

- מקובל להגדיר קבועים באותיות גדולות (ראו המלצה זו כמחייבת!)

משתנה סטטי כקבוע במחלקה

דוגמה במחלקה Person

```
class Person
{
public:
    static constexpr int ADULT_AGE = 18;

private:
    char name[20];
    int age;

public:
    Person(const char* name, int age)
    {...}

    void show() const
    {
        cout << "Name:  " << name << "\tAge:  " << age
             << " (" << (age < ADULT_AGE ? "child" : "adult") << ")\n";
    }
}; // class Person
```

במידה ועובדים עם גרסת קומפיילר לפני גרסה 11, יש לאתחל את הקבוע מחוץ לגבולות המחלקה, בקובץ ה- CPP

הגדרת משתנה סטטי כקבוע

הגדרת קבוע במחלקה דוגמת גישה ב- main

```
void main()
{
    Person p1("gogo", 21);
    Person p2("momo", 15);

    cout << "A person younger than "
        << Person::ADULT_AGE << " is a child\n";

    p1.show();
    p2.show();
}
```

פניה למשתנה הסטטי תהייה דרך:
ClassName::

מאחר ומשתנה סטטי הוא גם חלק מנתוני כל אובייקט, ניתן לפנות אליו גם דרך אחד האובייקטים

```
A person younger than 18 is a child
Name:   gogo      Age:   21 (adult)
Name:   momo      Age:   15 (child)
```

יצירת תכונת id אוטומטי דוגמה

```
class Person
{
private:
    static int counter;
    char name[20];
    int id;

public:
    Person(const char* name)
    {
        strcpy(this->name, name);
        id = ++counter;
    }

    static int getNumOfPersons()
    {
        return counter;
    }

    void show() const
    {
        cout << "Id:   " << id
        << "\tName:   " << name << endl;
    }
}; // class Person
int Person::counter = 0;
```

```
0 persons have been created
2 persons have been created
Id:   1   Name:   Gogo
Id:   2   Name:   Momo
2 persons have been created
```

```
void main()
{
    cout << Person::getNumOfPersons()
      << " persons have been created\n";

    Person p1("Gogo");
    Person p2("Momo");

    cout << Person::getNumOfPersons()
      << " persons have been created\n";

    p1.show();
    p2.show();

    cout << p2.getNumOfPersons()
      << " persons have been created\n";
}
```

counter =
name =“Gogo”
id =1

Person::counter=2

counter =
name = “Momo”
id = 2

אפשרי לפנות למתודה סטטית עם אובייקט, אך עדיף עם שם המחלקה

אתחול המשתנה הסטטי. מאותחל מחוץ למחלקה ולא ב-c’tor משום שאמור להתבצע פעם אחת בלבד, ולא עבור כל אובייקט. שימו לב: שורת איתחול זו תהייה בקובץ cpp ולא ב- h!!!

אתחול משתנה סטטי

- מאחר ומשתנה סטטי משותף לכל האובייקטים מהמחלקה ונולד פעם אחת בלבד בתחילת התוכנית, לא ניתן לאתחלו בשורת האתחול בקונסטרקטור
  - כי אז למעשה הוא יאותחל מחדש עם יצירת כל אובייקט
  - ואתחול משמעו מתן ערך פעם אחת בלבד (בניגוד להשמה)

- מאחר והמשתנה הסטטי נולד לפני כל אובייקט מהמחלקה, ניתן להשתמש בו כערך ב"מ בקונסטרקטור

אתחול משתנה סטטי דוגמה

```
class Person
{
private:
    static int counter;
    char name[20];
    int id;

public:
    Person(const char* name) : /*counter(0),*/ id(++counter)
    {
        strcpy(this->name, name);
    }

    ...
}; // class Person

int Person::counter = 0;
```

'counter': cannot initialize static class data via constructor

שגיאת קומפילציה עם ניסיון אתחול המשתנה הסטטי בשורת האתחול

אין בעיה להשתמש במשתנה הסטטי בשורת האתחול

friend

פונקציות

מתן חברות למחלקה

```
class Point
{
    int x, y;
public:
    Point(int x = 0, int y = 0) : x(x), y(y) {.

     bool equals(const Point& other) const
     {
         cout << "In Point::equals\n";
         return x == other.x && y == other.y;
     }

     friend bool equalPoints(const Point& p1, const Point& p2)
     {
         cout << "In equalPoints(global)\n";
         return p1.x == p2.x && p1.y == p2.y;
     }
};
void main()
{
    Point p1(10, 11), p2(20, 21);

    cout << "p1 equals p2? " << p1.equals(p2) << endl;
    cout << "p1 equals p2? " << equalPoints(p1, p2) << endl;
}
```

פונקציית friend לעולם לא תהיה const מאחר ואין אובייקט מפעיל

שיטה המשווה בין 2 נקודות, ומאחר וזוהי שיטה של המחלקה, האובייקט הראשון יהיה האובייקט המפעיל, והשני יהיה הפרמטר

מאחר וזוהי פונקציה גלובלית, תקבל את 2 האובייקטים בינהם רוצה להשוות כפרמטרים

```
In Point::Point (10, 11)
In Point::Point (20, 21)
In Point::equals
p1 equals p2? 0
In equalPoints(global)
p1 equals p2? 0
In Point::~Point (20, 21)
In Point::~Point (10, 11)
```

במידה והמימוש נמצא בקובץ ה- cpp המילה friend תכתב רק בהצהרה בקובץ ה- h

פונקציה friend

- פונקציה גלובלית (כמו ב- C) שכתובה בתוך המחלקה ורשאית לגשת לתכונות ה- private של האובייקט
- כתובה בתוך המחלקה מאחר ולוגית קשורה למחלקה
- נציין את המילה השמורה **friend** לפני שם הפונקציה
- במקרה בו מפרידים בין המימוש להגדרה, את המילה **friend** כותבים רק בהגדרה ב- h, ולא גם במימוש

מחלקת friend

- מחלקה יכולה "לתת חברות" למחלקה אחרת, ובכך לאפשר למחלקה האחרת לגשת לתכונות ולשיטות ה- private של המחלקה

- חברות ב- C++ זה כמו בחיים: **חברות נותנים, לא לוקחים**

מתן חברות דוגמה

```
class Point
{
private:
    int x, y;

public:
    Point(int x = 0, int y = 0) : x(x), y(y) {}

    friend class Circle;
};

class Circle
{
    Point center;
    int radius;

public:
    Circle(const Point& p, int radius) : center(p), radius(radius) {}
    void show() const
    {
        cout << "Center at " << center.x << ", " << center.y
                        << ") radius: " << radius << endl;
    }
};
```

המחלקה Point נותנת חברות למחלקה Circle

המחלקה Circle ניגשת ישירות לתכונות של center, וזאת בזכות מתן החברות

במקרה זה נמנע משימוש במתן חברות, מאחר וזה נגד הרעיון של הסתרת הנתונים ב- Object Oriented, וכאן מתן החברות הוא מטעמי עצלנות בלבד

מתן חברות דוגמה טובה

- נרצה לכתוב תוכנה המחזיקה נתוני עובדים בחברה
- העובדים מוגדרים ב- main ולא נרצה לאפשר שכפול שלהם
  - לכן נשים את ה- copy c'tor ב- private או נגדירו כ- delete
- כאשר מוסיפים עובד לחברה, נרצה לשמור שכפול שלו, ולא להחזיק פוינטר לאובייקט שמוגדר ב- main
  - לכן נרצה לאפשר שכפול עובדים..

- הפתרון הוא לשים את ה- copy c'tor של Employee ב- private, ולאפשר רק למחלקה Company לגשת אליו!
  - מבחינה תחבירית יבוצע באמצעות מתן חברות

מתן חברות דוגמה טובה הקוד

```
class Employee
{
private:
    char name[10];
    int salary;

    Employee(const Employee& other) = default;
public:
    Employee(const char* name, int salary)
    {
        strcpy(this->name, name);
        this->salary = salary;
    }
    void show() const
    {
        cout << name << " earns " << salary << endl;
    }

    friend class Company;
};
```

הגדרנו את ה- copy רק בשביל שיהיה תחת הרשאת private ולא public (שזו ההרשאה שיש לו כאשר הוא מתקבל במתנה)

תזכורת: מימוש ברירת מחדל, מאחר ואין הקצאות

מתן חברות למחלקה Company

מתן חברות דוגמה טובה הקוד (המשך)

```
class Company
{
public:
    static constexpr int MAX_EMPLOYEES = 10;
private:
    Employee* allEmployees[MAX_EMPLOYEES];
    int numOfEmployees = 0;
public:
    ~Company()  {…}

    bool addEmployee(const Employee& e)
    {
        if (numOfEmployees == MAX_EMPLOYEES)
           return false;

        allEmployees[numOfEmployees] = new Employee(e);
        allEmployees[numOfEmployees]->salary += 100;
        numOfEmployees++;
        return true;
    }
    void show() const
    {
        cout << "compnay has " << numOfEmployees << " employees:\n";
        for (int i = 0; i < numOfEmployees; i++)
            allEmployees[i]->show();
        cout << endl;
    }
};
```

לא ניתן לשכפל עובדים כי ה- copy ב- private!

יצירת שכפול העובד.
אפשרי רק בגלל מתן החברות!

סתם, רק כדי שתהיה לנו דרך קלה להבחין בין המקור להעתק בעת ההדפסות

```
void main()
{
    Employee e1("gogo", 15000);
    ///Employee e2(e1);

    Company c;
    c.addEmployee(e1);

    e1.show();
    cout << endl;
    c.show();
}
```

```
gogo earns 15000

compnay has 1 employees:
gogo earns 15100
```

תכונות mutable

תכונות mutable

- כאשר אובייקט הוא const:
  - לא ניתן לשנות את ערך תכונותיו
  - ניתן להפעיל עליו רק מתודות שהן const

- ישנן תכונות המשקפות מצב של אובייקט, והן צריכות להשתנות אפילו אם האובייקט הוא const או במתודה const

- הפתרון הוא להגדיר תכונות אלו כ- mutable

תכונות mutable דוגמה

```
class A
{
public:
    int x;
    mutable int y;

    void foo() const
    {
    ✗ x = 8;
    ✓ y = 9;
    }
};

void main()
{
    A a1;
    const A a2;

    ✓ a1.x = 3;
    ✓ a1.y = 4;

    ✗ a2.x = 5;
    ✓ a2.y = 6;
}
```

מתודה const יכולה לשנות רק תכונות שהן mutable

באובייקט const ניתן לשנות רק תכונות שהן mutable

תכונות mutable דוגמת הריגת אדם

```
class Person
{
    char name[20];
    mutable bool isAlive;

public:
    Person(const char* name)
        : isAlive(true)
    {
        strcpy(this->name, name);
    }

    void kill() const
    {
        isAlive = false;
    }

    void print() const
    {
        cout << name << " is "
                << (isAlive ? "alive" : "dead") << endl;
    }
};
```

גם אדם שהוא const יכול למות, לכן המתודה const

המתודה ניגשת לתכונה שהיא mutable ומשנה את ערכה

```
void main()
{
    Person p1("gogo");
    const Person p2("momo");

    p1.print();
    p2.print();

    p1.kill();
    p2.kill();

    p1.print();
    p2.print();
}
```

```
gogo is alive
momo is alive
gogo is dead
momo is dead
```

מרחבי שמות (namespace)

מרחבי שמות (namespace)

- כאשר עובדים על פרויקט גדול, לרוב משתמשים בספריות מוכנות
- יתכן מצב שיהיו מספריות שונות 2 פונקציות בעלות שם זהה המקבלות את אותם נתונים
- תיוצר בעיה של התנגשות בשמות, והקומפיילר לא ידע לאיזה פונקציה לפנות
- <u>הפתרון</u>: שימוש ב- namespace
  - השימוש ב- namespace מאפשר קישור של פונקציה מסוימת לחבילת קוד מסוימת

namespace דוגמה

```
1.  #include <iostream>
2.  using namespace std;

3.  namespace first
4.  {
5.      void foo()
6.      {
7.          cout << "This is the first foo\n";
8.      }
9.  }
10. namespace second
11. {
12.     void foo()
13.     {
14.         cout << "This is the second foo\n";
15.     }
16. }
17. void foo()
18. {
19.     cout << "This is just foo\n ";
20. }

21. int main()
22. {
23.     first::foo();
24.     second::foo();
25.     foo();
26. }
```

```
This is the first foo
This is the second foo
This is just foo
```

- להלן קטע קוד עם 3 פונקציות עם שם זהה
- 2 מימושים נמצאים בתוך namespace שונים
- פונקציה שלא בתוך namespace נמצאת במרחב השמות הגלובלי
- פניה לפונקציה הנמצאת בתוך namespace מחייבת ציון שם ה-namespace שבתוכו היא נמצאת

namespace קיצור אופן השימוש

```
1.  #include <iostream>
2.  using namespace std;

3.  namespace first
4.  {
5.      void foo()
6.      {
7.          cout << "This is the first foo\n";
8.      }
9.  }
10. namespace second
11. {
12.     void foo()
13.     {
14.         cout << "This is the second foo\n";
15.     }
16. }
17. using namespace second;

18. int main()
19. {
20.     first::foo();
21.     second::foo();
22.     foo();
23. }
```

פקודה זו מאפשרת לנו לפנות לפונקציות שתחת namespace זה בלי הקידומת

```
This is the first foo
This is the second foo
This is the second foo
```

namespace קיצור אופן השימוש

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

namespace first
{
   void foo() { cout << "This is the first foo\n"; }
}

namespace second
{
   void foo() { cout << "This is the second foo\n"; }
}

void foo()  {cout << "This is just foo\n"; }

using  namespace  first;
using  namespace  second;

int main()
{
   first::foo();
   second::foo();
   //foo();  // ERROR!
   ::foo();
}
```

במקרה זה נהייה חייבים תמיד לפנות בשם המלא של הפונקציה, אחרת נקבל את השגיאה: ambiguous call to overloaded function מאחר והקומפיילר אינו יודע לאיזו פונקציה לפנות

פניה בשם המלא לפונקציה הנמצאת במרחב השמות הגלובלי

מדוע שמים את using namespace std?

- בתוך namespace זה יש את כל הפקודות הבסיסיות
- בלעדיו נצטרך להוסיף את הקידומת std:: לכל הפונקציות הבסיסיות שבהן נשתמש, אחרת נקבל למשל את השגיאה:

error C2065: 'cout' : undeclared identifier

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

int main()
{
   int x;

   cout << "Enter a number: ";
   cin >> x;
}
```

```
#include <iostream>


int main()
{
   int x;

   std::cout << "Enter a number: ";
   std::cin >> x;
}
```

Ranged Based for-Loop
Auto
Raw String Literals

C++11

Ranged Based For-Loop

- ניתן לרוץ על לולאה ללא אינדקס!
- היתרון: לא צריך משתנה עבור גודל המערך

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

int main()
{
    int arr[] = { 1,2,3,4,5 };

    int size = sizeof(arr) / sizeof(arr[0]);
    for (int i=0 ; i < size ; i++)
        cout << arr[i] << " ";
    cout << endl;

    for (int& x : arr)
        x = x * x;

    for (int x : arr)
        cout << x << " ";
    cout << endl;
}
```

ניתן לקבל ref לאיבר ולשנותו

```
for (int x : arr)
    cout << x << " ";
cout << endl;
```

```
1 2 3 4 5
1 4 9 16 25
```

לולאה זו מתאימה רק למערכים עליהם רוצים לעבור באופן סדרתי על כל האיברים מההתחלה לסוף (אי אפשר למשל לעבור רק על מיקומים זוגיים וכד')

לולאה זו עובדת רק על משתנה שניתן לדעת לגביו את גדול המערך, כלומר לא יעבוד על מערך שהתקבל בפונקציה או על מערך שהוקצה דינאמית

C++11

auto

- ניתן להגדיר משתנים ללא טיפוס ספציפי!

```
int main()
{
    auto a = 5;
    auto* b = &a;
    auto c = &a;
    auto& d = a;
    auto x;

    cout << typeid(a).name() << endl;
    cout << typeid(b).name() << endl;
    cout << typeid(c).name() << endl;
    cout << typeid(d).name() << endl;

    d = 3;
    cout << a << " " << d << endl;

    char ch = 'a';
    c = ch;    // cannot convert from 'char' to 'int *'
}
```

טיפוסו של המשתנה נקבע מיד עם אתחולו
ולכן חייב להיות ידוע בזמן קומפילציה

```
int
int *
int *
int
```

```
3 3
```

typeid היא פונקציה המקבלת משתנה או טיפוס ומחזירה
משתנה מטיפוס type_info, המחזיק מידע על טיפוס.

על משתנה מטיפוס זה ניתן להפעיל את הפונקציה
name() שמחזירה את שם הטיפוס

auto + ranged for loop

```
int main()
{
    const char* daysOfWeek[] = { "Sunday", "Monday", "Tuesday",
                                 "Wednesday", "Thursday", "Friday", "Saturday" };

    for (auto day : daysOfWeek)
        cout << day << " ";
    cout << endl;
}
```

```
Sunday Monday Tuesday Wednesday Thursday Friday Saturday
```

יתרונות וחסרונות לשימוש ב- auto

+ מזכיר שפות מודרניות בהן לא צריך להגדיר ספציפית את סוג הטיפוס
+ מכריח שהמשתנה יהיה מאותחל
– קשה בקריאת הקוד לדעת מה טיפוס המשתנה, לצורך כך יש לחפש את שורת האתחול ולבדוק מהו הטיפוס איתו אותחל המשתנה

- ← אין סיבה להשתמש בו עבור תוכניות פשוטות

+ בפרק של ה- STL נראה כיצד שימוש בו יכול לחסוך בביצועים עקב טעויות נאיביות של מתכנתים מעולים

C++11

Raw String Literals

- כדי להציג למסך תווים מיוחדים (גרשיים, גרש, סלש וכד') יש לשים לפניהם \, מה שלפעמים יכול להיות מעיק, וכן להציג טקסט עם ירידות שורה
- הפתרון הוא Raw String Literals

```
int main()
{
    char* s1 = "\"Hello World\"";
    cout << s1 << endl;

    char* s2 = R"("Hello World")";
    cout << s2 << endl;

    char* s3 = "\"foo()\"";
    cout << s3 << endl;

    char* s4 = R"("foo()")";
    char* s4 = R"##("foo()")##";
    cout << s4 << endl;
}
```

נעטוף את הטקסט שנרצה שיוצג ב- R"(The Text)"

הבעיה: יש את הרצף "(בתוך הטקסט

הפתרון: לעטוף את הסוגריים ברצף כלשהו לבחירתכם (פה בחרתי ##)

```
"Hello World"
"Hello World"
"foo()"
"foo()"
```

ביחידה זו למדנו:

- שורת אתחול (init line)
- מעבר ב- c’tor עבור אובייקטים מוכלים
- המצביע this
- מצביע שהוא const
- static:
  - משתנים סטטיים במחלקה
  - שיטות סטטיות
- friend:
  - פונקציית friend
  - מחלקת friend
- תכונות mutable
- מרחבי שמות (namespace)
- ranged based for loop
- auto
- string literals