תכנות מכוון עצמים ו- C++
יחידה 04
constructors, destructor

קרן כליף

ביחידה זו נלמד:

- בנאי (constructor)
- בנאי ב"מ (empty/default constructor)
- מפרק (destructor)
- בנאי העתקה (copy constructor)
- R-Value ו- move constructor
- מילות המפתח default ו- delete
- בנאי ככלי להמרה
- explicit c’tor

קונסטרקטור

בנאי / c’tor / constructor

בנאי (constructor, בקיצור c’tor)

- כאשר נוצר אובייקט הוא עובר בשיטה שנקראית בנאי (constructor) אשר מאתחלת את נתוניו
- בכל מחלקה שאנחנו כותבים יש c’tor שאנחנו מקבלים במתנה מהקומפיילר אשר מאתחל את תכונות האובייקט עם זבל

```
class MyClass
{
public:

};

void main()
{
    MyClass c1;
    MyClass* c2;
    c2 = new MyClass;
    delete c2;
}
```

פה יש מעבר ב- c’tor

רק יצירת מצביע

יצירת האובייקט,
ולכן מעבר ב- c’tor

c’tor דריסתו

- ניתן לדרוס את ה- c’tor שאנחנו מקבלים במתנה ובכך לבצע פעולה שלנו עם יצירת האובייקט
- c’tor הוא שיטה במחלקה עם 2 מאפיינים:
  - שמו כשם המחלקה
  - אין לציין עבורו ערך מוחזר

```
class MyClass
{
public:
    MyClass()
    {
        cout << "In the c'tor of MyClass\n";
    }
};
void main()
{
    MyClass c1;
    cout << "-----------\n";
    MyClass* c2;
    cout << "-----------\n";
    c2 = new MyClass;
    cout << "-----------\n";
    delete c2;
}
```

```
In the c'tor of MyClass
-----------
-----------
In the c'tor of MyClass
-----------
```

c’tor תפקידו

- מאחר וה- c’tor נקרא עם יצירת האובייקט באופן מיידי, תפקידו לאתחל את ערכי האובייקט
- כלומר, לא נרצה שיווצר לנו אובייקט אשר תכונותיו עם ערכי זבל
- בבנאי שראינו, שאינו מקבל פרמטרים, נהוג לאפס את שדותיו של האובייקט

העמסה c'tor

- מאחר והבנאי הוא שיטה, ניתן להעמיס אותו
- נוכל לייצר c'tor אשר יקבל ערכים מהמשתמש
  - שם נוסף לבנאי שאינו מקבל פרמטרים: empty/default c'tor

```
#ifndef __CLOCK_H
#define __CLOCK_H

class Clock
{
public:
    Clock();
    Clock(int h, int m);

    void show() const;

private:
    int hours, minutes;
};
#endif //__CLOCK_H
```

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

#include "clock.h"

Clock::Clock()
{
    hours = minutes = 0;
}
Clock::Clock(int h, int m)
{
    hours = h;
    minutes = m;
}
void Clock::show() const {.}
```

```
00:00
21:30
```

```
#include "clock.h"

void main()
{
    Clock c1, c2(21, 30);

    c1.show();
    cout << endl;

    c2.show();
    cout << endl;
}
```

C‘tor Delegation

- מאפשר קריאה מבנאי אחד לבנאי אחר, ובכך אנחנו חוסכים את שורות הקוד המאתחלות את שדה האובייקט ומרכזים את כל האתחולים למקום אחד יחיד

```
#ifndef __CLOCK_H
#define __CLOCK_H

class Clock
{
public:
   Clock();
   Clock(int h, int m);

   void show() const;

private:
   int hours, minutes;
};
#endif //__CLOCK_H
```

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

#include "clock.h"

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

Clock::Clock(int h, int m)
{
    hours = h;
    minutes = m;
}
void Clock::show() const {.}
```

```
#include "clock.h"

void main()
{
    Clock c1, c2(21, 30);

    c1.show();
    cout << endl;

    c2.show();
    cout << endl;
}
```

```
00:00
21:30
```

ביטול ה- defualt c’tor שהתקבל במתנה

- ברגע שאנחנו מגדירים c’tor כלשהו, הקומפיילר לוקח ה- default c’tor שהוא נתן לנו במתנה
- התוצאה: לא ניתן לייצר אובייקטים ללא פרמטרים
- עדיין נוכל להגדיר אותו בעצמנו, כמו בדוגמאות הקודמות

```
#ifndef __CLOCK_H
#define __CLOCK_H

class Clock
{
public:
    Clock(int h, int m);

    void show() const;

private:
    int hours, minutes;
};

#endif // __CLOCK_H
```

```
#include "clock.h"

void main()
{
    Clock c1;
    Clock c2(21, 30);
}
```

```
error C2512: 'Clock' : no appropriate
default constructor available
```

C++11

מילת המפתח default

- ראינו שברגע שכתבנו c'tor בעצמנו, כבר אין default c'tor במתנה, ואם אנחנו רוצים אותו עלינו להגדיר אותו בעצמנו
- ניתן להגדירו ללא גוף, ולהגיד לקומפיילר שיספק את מימוש ברירת המחדל שלו

```
class Clock
{
public:
    Clock() = default;
    Clock(int h, int m)
    {
        hours = h;
        minutes = m;
    }
    void show() const  {...}

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

```
void main()
{
    Clock c1, c2(21, 30);

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

לא יתקמפל עד שיוגדר default c'tor

```
0-858993460:0-858993460
21:30
```

ערכי default ב- c’tor

- מאחר וה- c’tor הוא שיטה, ניתן לתת ערכי default לפרמטרים שהוא מקבל

```
#ifndef __CLOCK_H
#define __CLOCK_H

class Clock
{
public:
    Clock(int h=0, int m=0);

    void show() const;

private:
    int hours, minutes;
};
#endif //__CLOCK_H
```

3 c’tor במחיר של אחד ☺
אחד מהם הוא default c’tor

```
#include "clock.h"

void main()
{
    Clock c1, c2(10), c3(11, 30);

    c1.show();
    cout << endl;

    c2.show();
    cout << endl;

    c3.show();
    cout << endl;
}
```

```
00:00
10:00
11:30
```

C++11

מתן ערכי ב"מ לשדות בהגדרת המחלקה

- החל מ- C++11 ניתן לאתחל את התכונות בגוף המחלקה, ולא רק בקונסטרקטור

```
#ifndef __CLOCK_H
#define __CLOCK_H

class Clock
{
public:
    Clock() = default;
    Clock(int h, int m);

    void show() const;

private:
    int hours=10, minutes;
};

#endif // __CLOCK_H
```

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

#include "clock.h"

void main()
{
    Clock c1;

    c1.show();
    cout << endl;
}
```

```
10:0-858993460
```

האם תמיד נרצה בנאי שלא מקבל פרמטרים?

- כאשר יש בנאי שלא מקבל פרמטרים מקובל שהוא יאפס את כל השדות
- לא תמיד נרצה שיהיה לנו בנאי המאפס את כל ערכי השדות שכן אז לא תהייה משמעות לאובייקט:
  - למשל אובייקט "תאריך": האם יש משמעות לתאריך 0.0.0??
  - למשל אובייקט "שחקן כדורסל": האם יש משמעות לאובייקט שגובהו 0.0 שמו "" ותאריך לידתו 0.0.0?
  - למשל עבור אובייקט "שעון", דווקא כן מקובל ששעון מאופס הוא 00:00

יצירת מערך של אובייקטים

- כאשר יוצרים מערך של אובייקטים, הקומפיילר יוצר כל אובייקט דרך מעבר ב- default c'tor
- במקרה ואין default c'tor נקבל שגיאת קומפילציה

```
class MyClass
{
    int num1, num2;
public:
    MyClass(int n1, int n2)
    {
        num1 = n1;
        num2 = n2;

        cout << "In c'tor -> num1=" << num1 << " num2=" << num2 << endl;
    }
};
```

```
void main()
{
    MyClass arr[2];
}
```

abc no default constructor exists for class "MyClass"

C2512 'MyClass': no appropriate default constructor available

C++11

אתחול מערך אובייקטים

```
class MyClass
{
    int num1, num2;
public:
    MyClass(int n1, int n2)
    {
        num1 = n1;
        num2 = n2;

        cout << "In c'tor -> num1=" << num1 << " num2=" << num2 << endl;
    }
};
```

```
void main()
{
    //MyClass arr1[2];
    MyClass arr2[2] = { {3, 4}, {7, 8} };
}
```

```
In c'tor -> num1=3 num2=4
In c'tor -> num1=7 num2=8
```

ואם לא רוצים לספק default c’tor?

- אמרנו שלא עבור כל מחלקה נרצה לספק default c’tor מאחר ואין משמעות לוגית לאובייקט ללא איתחול (למשל "תאריך", "שחקן כדורסל" וכד')

- במקרה כזה נגדיר מערך של מצביעים, ונקצה כל איבר רק לאחר קבלת נתונים

דוגמא

```
Enter day, month, year: 31 7 2017
Enter day, month, year: 19 2 2017
Enter day, month, year: 3 3 2016
31/7/2017
19/2/2017
3/3/2016
```

```
class Date
{
private:
    int day, month, year;

public:
    Date(int d, int m, int y);

    void show() const;
};

Date::Date(int d, int m, int y) {
    day = d;
    month = m;
    year = y;
}

void Date::show() const {
    cout << day << "/" << month << "/" << year << " ";
}
```

```
void main()
{
    Date* arr[3];

    for (int i = 0; i < 3; i++)
    {
        int day, month, year;

        cout << "Enter day, month, year: ";
        cin >> day >> month >> year;
        arr[i] = new Date(day, month, year);
    }

    for (int i = 0; i < 3; i++)
    {
        arr[i]->show();
        cout << endl;
    }

    for (int i = 0; i < 3; i++)
        delete arr[i];
}
```

מערך של מצביעים

הקצאת כל איבר במערך

מאחר וכל איבר הוא מצביע,
נפנה לשיטות עם ->

לא לשכוח לשחרר את האיברים,
מאחר והוקצו דינאמית

דיסטרקטור

מפרק / d’tor / destructor

מפרק (destructor, d’tor)

- כאשר אובייקט מת (עם סיום הפונקציה או התוכנית) יש מעבר בשיטה הנקראית destructor
- שיטה זו קיימת בכל מחלקה והיא עושה כלום
- ניתן לדרוס שיטה זו עם מימוש שלנו
- d’tor הוא שיטה במחלקה עם 3 מאפיינים:
  1. לפני שם השיטה יש את הסימן ~
  2. שמה כשם המחלקה
  3. אין לציין עבורה ערך מוחזר

מעבר ב- destructor דוגמה

```
class Stam
{
    int num;
public:
    Stam(int n)
    {
        num = n;
        cout << "In c'tor -> num=" << num << endl;
    }

    ~Stam()
    {
        cout << "In d'tor -> num=" << num << endl;
    }
};

void foo(Stam s)
{
    cout << "In foo\n";
}

void goo(Stam& s)
{
    cout << "In goo\n";
}
```

destructor

הריסת הפרמטר עם היציאה מהפונקציה

```
In c'tor -> num=5
---------------
In foo
In d'tor -> num=5
---------------
In goo
---------------
---------------
In c'tor -> num=8
---------------
In d'tor -> num=8
---------------
In d'tor -> num=5
```

```
void main()
{
    Stam s1(5);
    cout << "-----------\n";
    foo(s1);
    cout << "-----------\n";
    goo(s1);
    cout << "-----------\n";
    Stam* s2;
    cout << "-----------\n";
    s2 = new Stam(8);
    cout << "-----------\n";
    delete s2;
    cout << "-----------\n";
}
```

הצורך ב- destructor

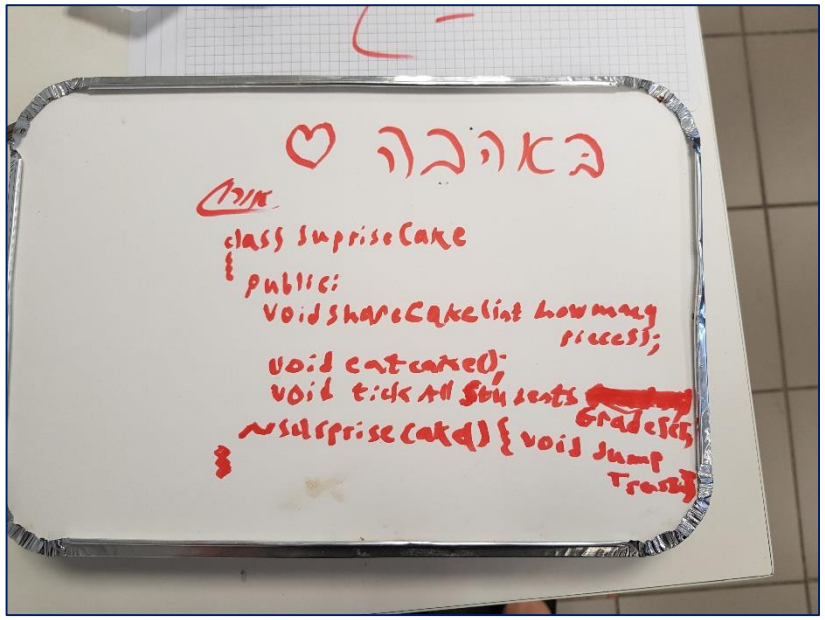
- יתכן ובמחלקה יהיו תכונות שיוקצו דינאמית
- ה- destructor הוא המקום בו נשחרר זכרון זה

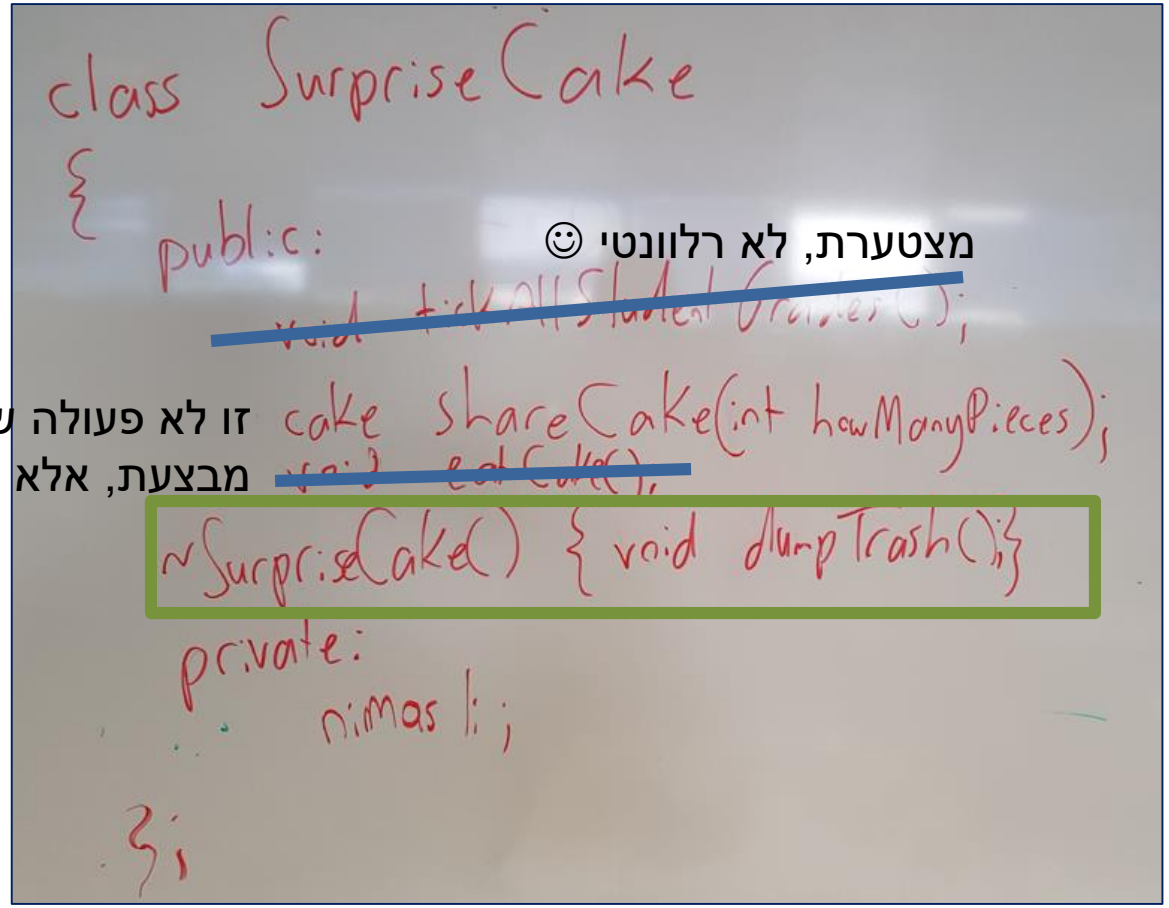
```
#ifndef __PERSON_H
#define __PERSON_H

#include <string.h>

class Person
{
    char* name;
    int id;

public:
    Person(const char* n, int i)
    {
        name = new char[strlen(n)+1];
        strcpy(name, n);
        id = i;
    }

    ~Person()
    {
        delete[]name;
    }
};

#endif // __PERSON_H
```

הקצאה דינאמית של תכונה

שחרור התכונה שהוקצתה דינאמית

הצורך ב- destructor





מצטערת, לא רלוונטי ☺

זו לא פעולה שהעוגה מבצעת, אלא Person

קופי קונסטרקטור

בנאי העתקה / copy constructor / copy c’tor

בנאי העתקה (copy c'tor)

- copy c'tor הוא מקרה פרטי של בנאי שהפרמטר שהוא מקבל הוא אובייקט אחר מאותו הטיפוס
- מטרתו לייצר אובייקט נוסף זהה לאובייקט שהתקבל כפרמטר
- הקומפיילר מספק לנו copy c'tor במתנה אשר מבצע "העתקה רדודה": מעתיק תכונה-תכונה

```
void main()
{
    Clock c1(11, 30);
    Clock c2(c1);

    cout << "c1: ";
    c1.show();
    cout << "\nc2: ";
    c2.show();

    c1.setHour(13);
    cout << "\nAfter change:\nc1: ";
    c1.show();
    cout << "\nc2: ";
    c2.show();
    cout << endl;

}
```

יצירת אובייקט דרך copy c'tor

```
c1: 11:30
c2: 11:30
After change:
c1: 13:30
c2: 11:30
```

ניתן לראות כי נוצר אובייקט חדש עם ערכים הזהים לאובייקט המקורי ברגע היצירה. 2 אובייקטים אלו כעת בלתי תלויים, ושינוי באחד לא משפיע על השני

copy c'tor דריסתו

- גם את ה- copy c'tor ניתן לדרוס ולממש מחדש
- במימוש נעתיק את ערכי התכונות מהפרמטר שהתקבל לאובייקט הנוצר

```
class Clock
{
public:
    Clock(int h=0, int m=0);
    Clock(const Clock& other);

private:
    int hours, minutes;
};

Clock::Clock(const Clock& other)
{
    hours = other.hours;
    minutes = other.minutes;
}
```

נשים לב:

1. הפרמטר המתקבל הוא by ref: אין צורך להעביר העתק של הפרמטר מטעמי יעילות (בהמשך נראה סיבה נוספת)
2. הפרמטר המתקבל הוא const: כדי להצהיר שהשיטה לא משנה את הפרמטר שהתקבל

השמת הערכים של other בתוך האובייקט שנוצר עכשיו

copy c'tor הבעייתיות

```
class Person
{
    char* name;
    int id;

public:
    Person(const char* n, int i) {...}

    Person(const Person& other)
    {
        name = other.name;
        id = other.id;
    }

    ~Person()  {...}
};


void main()
{
    Person p1("gogo", 111);
    Person p2(p1);
}
```

המימוש המתקבל במתנה

המחרוזת "gogo"
נמצאת בכתובת 1000

p1	
id:	111
name:	1000

p2	
id:	111
name:	1000

p2 הוא העתק של p1, ובפרט מכיל העתק של הכתובת שבתכונה name

p1 הולך למשרד הפנים ומשנה את שמו הנמצא בכתובת 1000. השינוי משפיע גם על p2...

מתי חייבים לממש copy c’tor?

- ראינו שאנחנו מקבלים copy c’tor במתנה, אך כאשר יש במחלקה הקצאות דינאמיות, נצטרך לממש אותו בעצמנו
- אחרת תהייה הבעיה של העתקה רדודה, כלומר, 2 מצביעים מכילים את אותה הכתובת, ואז יש תלות בין האובייקטים

כאשר יש תכונה שמקצים אותה דינאמית, נממש copy c’tor כדי למנוע את ההעתקה הרדודה.

כאשר מממשים copy c’tor כנראה צריך גם לממש את ה- d’tor, לשחרור ההקצאה

השמה של אובייקטים הינה על אותו עיקרון. כרגע, אם יש הקצאות דינאמיות במחלקה, נמנע מלבצע השמה בין אובייקטים.
הסבר מפורט כאשר נלמד על העמסת אופרטורים..

מימוש תקין של ה- copy c'tor

```
class Person
{
    char* name;
    int id;

public:
    Person(const char* n, int i) {...}

    Person(const Person& other)
    {
        name = new char[strlen(other.name)+1];
        strcpy(name, other.name);
        id = other.id;
    }

    ~Person()  {...}
};

void main()
{
    Person p1("gogo", 111);
    Person p2(p1);
}
```

המחרוזת "gogo" נמצאת בכתובת 1000

המחרוזת "gogo" נמצאת גם בכתובת 2000

p1	
id:	111
name:	1000

p2	
id:	111
name:	2000

המימוש שלנו ל- copy c'tor

p2 הוא העתק של p1, אבל מכיל <u>העתק</u> של התוכן שבתכונה name

p1 הולך למשרד הפנים ומשנה את שמו הנמצא בכתובת 1000. השינוי הפעם אינו משפיע על p2...

מעבר ב- copy c’tor

- עוברים ב- copy c’tor בכל פעם כאשר נוצר העתק של אובייקט:
  1. ביצירת אובייקט עם נתונים של אובייקט אחר

```
int main()
{
     Person p1("gogo", 111);
     Person p2(p1);
}
```

  2. כאשר מעבירים אובייקט by value לפונקציה או לשיטה

```
void foo(Person p);
```

  3. כאשר מחזירים אובייקט by value מפונקציה

```
Person moo1();
Person& moo2();
Person* moo3();
```

פה אין מעבר ב- copy c’tor כי אלו התייחסויות לאובייקטים קיימים

דוגמאות למעברים ב- copy c'tor

```
class MyClass
{
public:
    MyClass()                  { cout << "In c'tor\n"; }
    MyClass(const MyClass& )   { cout << "In copy c'tor\n"; }
    ~MyClass()                 { cout << "In d'tor\n"; }
};
void foo(MyClass c)
{
    cout << "In foo\n";
}
void goo(const MyClass& c)
{
    cout << "In goo\n";
}
MyClass moo()
{
    cout << "In moo\n";
    MyClass c;
    return c;
}
MyClass koo()
{
    cout << "In koo\n";
    return MyClass();
}
```

```
void main()
{
    MyClass c1;
    cout << "---------\n";
    foo(c1);
    cout << "---------\n";
    goo(c1);
    cout << "---------\n";
    MyClass c2 = moo();
    cout << "---------\n";
    MyClass c3 = koo();
    cout << "---------\n";
}
```

```
In c'tor
---------
In copy c'tor
In foo
In d'tor
---------
In goo
---------
In moo
In c'tor
In copy c'tor
In d'tor
---------
In koo
In c'tor
---------
In d'tor
In d'tor
In d'tor
```

במקרה של יצירת אובייקט בשורת ההחזרה, הקומפיילר מבצע אופטימיזציה ואינו מייצר אובייקטים מיותרים

מדוע ה- copy c’tor חייב לקבל את הפרמטר by ref

```
MyClass(const MyClass& )
```

- כאשר מעבירים אובייקט לפונקציה by value מועבר העתק שלו
- ההעתק נוצר ע"י מעבר ב- copy c’tor
- כדי שה- copy c’tor יקבל העתק של הפרמטר הוא יצטרך לייצר אותו דרך מעבר ב- copy c’tor
- וכך נוצר loop אינסופי...

כתיבת ה- copy c’tor ב- private

- יתכן ונרצה למנוע מעבר ב- copy c’tor
  - למשל: למנוע שיבוט בני-אדם
- במקרה כזה נצהיר על ה- copy c’tor ב- private ולא נממש
- התוצאה: שגיאת קומפילציה כאשר יש ניסיון לייצר העתק
- אם לא נגדיר אותו ב- private יהיה את ה- copy c’tor שניתן במתנה

```
class MyClass
{
private:
   MyClass(const MyClass&);

public:
   MyClass()  {...}
   ~MyClass() {...}
};

int main()
{
   MyClass c1;
   MyClass c2(c1);
}
```

"MyClass::MyClass(const MyClass &)" (declared at line 7) is inaccessible

'MyClass::MyClass': cannot access private member declared in class 'MyClass'

C++11

מילת המפתח delete

- ראינו שכדי לחסום שימוש ב- copy c'tor הגדרנו אותו ב- private
- מילת המפתח delete חוסכת זאת מאיתנו ובעצם חוסמת את השימוש במתודה

```
class MyClass
{
public:
    MyClass()  {...}
    ~MyClass() {...}

    MyClass(const MyClass&) = delete;
};
```

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

'MyClass::MyClass(const MyClass &)': attempting to reference a deleted function

Move C’tor

R-Value לעומת L-Value

המושגים L-Value ו- R-Value

- הקומפיילר משתמש רבות במושגים L-Value ו- R-Value
  - L-Value הינו משתנה שיש לו שם וניתן לגשת אליו ישירות
  - R-Value הינו משתנה זמני שאין לו שם
    - יש לו כתובת בזיכרון אך אין משמעות לשנות את ערכו מאחר והאובייקט תיכף ימות

```
int foo()        {return 5;}
void goo(int x) {cout << x << endl;}

int main()
{
    int x = 3 + 4;
    goo(foo());
}
```

- הקומפיילר יודע לזהות מקרה בו פונקציה מקבלת כפרמטר אובייקט זמני (שתיכף ימות)!

זיהוי R-Value דוגמה (1)

```
class Person
{
    char* name;
public:
    Person(const char* n)
    {
        name = new char[strlen(n) + 1];
        strcpy(name, n);
        cout << "In Person::Person name is " << name
                << " at address " << (void*)name << "\n";
    }

    Person(const Person& other) {…}

    ~Person()
    {
        cout << "In Person::~Person ";
        if (name != nullptr)
                cout << "delete " << name << " ";
        cout << "at address " << (void*)name << "\n";
        delete[]name;
    }
};
```

זיהוי R-Value דוגמה (2)

```
void goo(const Person& p)
{
     cout << "In goo& p.name=" << p.name << " at address "
          << (void*)(p.name) << "\n";
}

void goo(const Person&& p)
{
     cout << "In goo&& p.name=" << p.name << " at address "
          << (void*)(p.name) << "\n";
}

void main()
{
    Person p1("gogo");
    cout << "--------------\n";
    goo(p1);
    cout << "--------------\n";
    goo(Person("momo"));
    cout << "--------------\n";
}
```

```
In Person::Person name is gogo at address 0011E948
--------------
In goo& p.name=gogo at address 0011E948
--------------
In Person::Person name is momo at address 0011EBE8
In goo&& p.name=momo at address 0011EBE8
In Person::~Person delete momo at address 0011EBE8
--------------
In Person::~Person delete gogo at address 0011E948
```

C++11

move c’tor

- ב- copy c’tor ישנה בעיה במקרה בו הוא מעתיק אובייקט זמני שתיכף ימות:
  - העתקות שהוא מבצע מיותרות

- הרעיון של move c’tor הוא בהינתן שמייצרים אובייקט כהעתק מאובייקט זמני, אז לא משכפלים את הערכים, אלא "גונבים" אותם ע"י השתלטות על כתובותיהם

מימוש move c’tor

המימוש של ה- move c’tor יותר יעיל
מאחר ואינו מבצע הקצאות ושחרורי זכרון

- ה- copy c’tor נראה כך:

```
Person(const Person& other)
{
    name = new char[strlen(other.name)+1];
    strcpy(name, other.name);
    id = other.id;
}
```

- ה- move c’tor נראה כך:

```
Person(Person&& other)
{
    name = other.name;
    other.name = nullptr;

    id = other.id;

    cout << "In Person::Person(move) name is " << name
        << " at address " << (void*)name << "\n";
}
```

"גניבת" שדה הכתובת
– לא העתקה

כדי שלא תהיה
תעופה בשחרור

move c’tor דוגמת שימוש

```
Person foo(const char* name)
{
    cout << "In foo\n";

    Person a(name);
    return a;
}

Person koo(const char* name)
{
    cout << "In koo\n";
    return Person(name);
}

void main()
{
    Person p1 = foo("fofo");
    cout << "-----------\n";
    Person p2 = koo("koko");
    cout << "-----------\n";
}
```

שוב רואים שצורת כתיבה זו
של יצירת האובייקט בשורת
ההחזרה יותר יעילה

```
In foo
In Person::Person name is fofo at address 0066EEC8
In Person::Person(move) name is fofo at address 0066EEC8
In Person::~Person at address 00000000
--------------
In koo
In Person::Person name is koko at address 0066F168
--------------
In Person::~Person delete koko at address 0066F168
In Person::~Person delete fofo at address 0066EEC8
```

המרות / casting

Casting אוטומטי

Forced Casting

יצירת אובייקט זמני

casting אוטומטי

```
class MyClass
{
private:
   int num;
public:
    MyClass(int n)
    {
        num = n;
        cout << "In c'tor num=" << num << endl;
    }
    int getNum() const { return num; }
};

void foo(MyClass c)
{
    cout << "In foo: c.num=" << c.getNum() <
}

void main()
{
    MyClass c(4);
    foo(c);
    foo(7);
}
```

```
In c'tor num=4
In foo: c.num=4
In c'tor num=7
In foo: c.num=7
```

מאחר והפונקציה foo אמורה לקבל משתנה מטיפוס MyClass, הקומפיילר בודק האם בהינתן משתנה מטיפוס int, ניתן ליצר אובייקט מטיפוס MyClass. מאחר וקיים למחלקה בנאי המקבל int ניתן לייצר אובייקט זמני מהטיפוס המבוקש

מעבר ב- copy c'tor ליצירת העתק הפרמטר

מעבר ב- c'tor המקבל int ליצירת האובייקט

casting אוטומטי (2)

```
class MyClass
{
private:
   int num;
public:
    MyClass(int n)
    {
        num = n;
        cout << "In c'tor num=" << num << endl;
    }
    int getNum() const { return num; }
};

void foo(MyClass c)
{
    cout << "In foo: c.num=" << c.getNum() << endl;
}

void main()
{
    foo('a');
    foo(91.8);
}
```

```
In c'tor num=97
In foo: c.num=97
In c'tor num=91
In foo: c.num=91
```

הקומפיילר לא מוצא c’tor מתאים, ולכן מנסה להמיר
את טיפוס הפרמטר ל- int, מאחר והצליח פונה לבנאי

casting אוטומטי באמצעות בנאי סיכום

- כאשר מנסים לשלוח לפונקציה המצפה לקבל טיפוס Z, פרמטר מטיפוס Y, הקומפיילר בודק האם ניתן לבצע המרה לטיפוס המבוקש Z
- כדי לבצע casting מטיפוס Y ל- Z, יש ליצר אובייקט זמני מהטיפוס Z, דרך מעבר ב- c'tor מתאים המקבל Y
- אם לא ניתן לבצע את ה- casting, כלומר לא קיים בנאי מתאים, מתקבלת שגיאת קומפילציה

צורות נוספות ל- casting

בכל המקרים של casting נוצר אובייקט זמני, שימות עם סיום השורה

```
class MyClass
{
private:
    int num;
public:
    MyClass(int n)
    {
        num = n;
        cout << "In c'tor num=" << num << endl;
    }
    ~MyClass() {cout << "In d'tor num=" << num << endl;}
    int getNum() const { return num; }
};

void foo(MyClass c)  {cout << "In foo: c.num=" << c.getNum() << endl;}

void main()
{
    MyClass c(3);
    cout << "--------------\n";
    foo(c);
    cout << "--------------\n";
    foo(4);
    cout << "--------------\n";
    foo(MyClass(5));
    cout << "--------------\n";
    foo((MyClass)6);
    cout << "----------
}
```

מעבר ב- c’tor

קריאה לפונקציה בצורה הרגילה, יצירת העתק

automatic casting: יצירת אובייקט זמני

יצירת אובייקט זמני והעתקתו לפונקציה

forced casting: יצירת אובייקט זמני

```
In c'tor num=3
--------------
In foo: c.num=3
In d'tor num=3
--------------
In c'tor num=4
In foo: c.num=4
In d'tor num=4
--------------
In c'tor num=5
In foo: c.num=5
In d'tor num=5
In d'tor num=5
--------------
In c'tor num=6
In foo: c.num=6
In d'tor num=6
In d'tor num=6
--------------
In d'tor num=3
```

הגבלות על השימוש ב- forced casting

```
class MyClass
{
private:
    int num1, num2;
public:
    MyClass(int n1, int n2)
    {
        num1 = n1;
        num2 = n2;
    }
};

void foo(MyClass c) { /*...*/ }

int main()
{
    foo(MyClass(5, 6));
    foo((MyClass)7, 8);
}
```

הפעם למחלקה יש שני שדות ולכן שני שדות לקונסטרקטור

יצירת אובייקט זמני

לא ניתן להשתמש ב- forced casting כאשר הקונסטרקטור מקבל יותר מפרמטר אחד. הקומפיילר חושב שיש לקרוא לפונקציה foo שמקבלת MyClass ו- int.

casting אוטומטי מוגבל במספר הקפיצות האוטומטיות

```
class One
{
public:
    One(int n)          { cout << "Creating One\n"; }
};

class Two
{
public:
    Two(const One& o)   { cout << "Creating Two\n"; }
};

class Three
{
public:
    Three(const Two& t) { cout << "Creating Three\n"; }
};
```

סיכום הפרמטרים המתקבלים בכל קונסטרקטור:

int → One → Two → Three

casting אוטומטי מוגבל במספר הקפיצות האוטומטיות

```
int main()
{
    One o1(7);
    cout << "1 -------------\n";
    Two t1(o1);
    cout << "2 -------------\n";
    Two t2(8);
    cout << "3 -------------\n";
    Three th1(t2);
    cout << "4 -------------\n";
    Three th2(o1);
    cout << "5 -------------\n";
    Three th3(9);
    cout << "6 -------------\n";
}
```

צריכות להתבצע 3 המרות,
ולכן לא עובר קומפילציה

```
Creating One
1 -------------
Creating Two
2 -------------
Creating One
Creating Two
3 -------------
Creating Three
4 -------------
Creating Two
Creating Three
5 -------------
```

no instance of constructor "Three::Three" matches the argument list

'Three::Three(Three &&)': cannot convert argument 1 from 'int' to 'const Two &'

סיכום הפרמטרים המתקבלים בכל קונסטרקטור:

int → One → Two → Three

Explicit Constructor

(1) explicit c’tor

```
class MyClass
{
private:
   int x;
public:
    MyClass(int num)
    {
        x = num;
        cout << "In c'tor x=" << x << endl;
    }
};
void foo(MyClass c)
{
     cout << "In foo\n";

}
void main()
{
    MyClass c1(97);
    MyClass c2 = 98;

    foo(c1);
    foo(99);
}
```

```
In c'tor x=97
In c'tor x=98
In foo
In c'tor x=99
In foo
```

קריאה מפורשת ל- c’tor (explicit)

קריאה לא מפורשת ל- c’tor (implicit)

קריאה מפורשת עם
אובייקט מתאים למתודה

מתבצעת המרה (explicit)
מ- int ל- MyClass

(2) explicit c'tor

```
class MyClass
{
private:
   int x;
public:
   explicit MyClass(int num)
    {
        x = num;
        cout << "In c'tor x=" << x << endl;

    }
};

void foo(MyClass c) {...}

void main()
{
    MyClass c1(97);
    MyClass c2 = 98;

    foo(c1);
    foo(99);
}
```

כאשר מציינים שבנאי הוא explicit משמע
לא תתבצע המרה אוטומטית לטיפוס

no suitable constructor exists to convert from "int" to "MyClass"

מתי נרצה להשתמש ב- explicit?

```
class Person
{
    char name[20];
public:
    Person(const char* n)
    {
        strcpy(name, n);
        cout << "Creating " << name << endl;
    }

    Person(const Person& other)
    {
        strcpy(name, other.name);
        cout << "Copying " << name << endl;
    }

    ~Person()
    {
        cout << "Killing " << name << endl;
    }
};
```

```
int main()
{
    Person p("gogo");
    cout << "----------\n";
    p = "momo";
    cout << "----------\n";
}
```

```
Creating gogo
----------
Creating momo
Killing momo
----------
Killing momo
```

הכוונה כנראה הייתה לשים את הערך momo בשדה השם (במקום הפעלה של מתודה setName), אבל התוצר הוא יצירה של אובייקט זמני, השמתו ל- p ולבסוף הריגתו. מאוד לא יעיל!

אם הקונסטרקטור היה explicit שורת השמה זו לא הייתה עוברת קומפילציה, כי יש יצירת אובייקט באופן שאינו מפורש.

לכן יש השמים explicit על קונסטרקטור כדי שלא יווצרו בתמימות אובייקטים ללא בקרה

סיכום 3 המתנות

default c'tor

- בנאי שלא עושה כלום, מאפשר לייצר אובייקט ללא פרמטרים
- נלקח עם מימושו של c'tor כלשהו

default c'tor

- לא מבצע כלום
- נרצה לדרוס אותו כאשר יש הקצאות דינאמיות ביצירת האובייקט

copy c'tor

- מבצעה העתקה רדודה של השדות
- נרצה לדרוס אותו כאשר יש הקצאות דינאמיות ביצירת האובייקט כדי למנוע הצבעה כפולה

ביחידה זו למדנו:

- בנאי (constructor)
- בנאי ב"מ (empty/default constructor)
- מפרק (destructor)
- בנאי העתקה (copy constructor)
- מילות המפתח default ו- delete
- בנאי ככלי להמרה
- explicit c’tor