תכנות מכוון עצמים ו- C++
יחידה 06
העמסת אופרטורים

קרן כליף

ביחידה זו נלמד:

- מהי העמסת אופרטורים ומוטיבציה
- אופרטור אונארי לעומת אופרטור בינארי
- העמסת אופרטורים:
  - אופרטורים + ו- -
  - אופרטורים כפונקציות friend
  - אופרטור השמה
    - שימוש באופרטור השמה מ- copy c'tor
    - ההבדל בין copy c'tor לאופרטור השמה
  - אופרטור מינוס (- אונארי)
  - אופרטור =+, =-
  - אופרטורים ++ ו- -- (prefix, postfix)
  - אופרטור [ ]
  - אופרטורים לוגיים: ==, >, < , =<, =>
  - אופרטור <<, >>
  - אופרטור casting
  - אופרטור ()
- אופרטור השמה המקבל &&
- std:swap ו- std::move

העמסת אופרטורים מוטיבציה

```
class Point
{
    int x, y;

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

    Point add(const Point& other) const
    {
        return Point(x + other.x, y + other.y);
    }

    void show() const
    {
        cout << "(" << x << ", " << y << ")\n";
    }
};
```

```
void main()
{
    Point p1(5, 6), p2(7, 8);
    Point p3 = p1.add(p2);

    p3.show();
}
```

```
(12, 14)
```

- היינו שמחים להיות מסוגלים לכתוב את הקוד הבא:
  - כלומר, להשתמש באופרטור +

```
void main()
{
    Point p1(5, 6), p2(7, 8);
    Point p3 = p1 + p2;

    p3.show();
}
```

העמסת אופרטורים (Operators Overloading)

- ישנם אובייקטים שהיינו רוצים להפעיל עליהם פעולות חשבון כגון +, -, השמה וכו'
- כאשר אנחנו רוצים לחבר בין 2 נקודות, הקומפיילר לא יכול לנחש את כוונתנו, כלומר יצירת נקודה חדשה שערך ה- x שלה יהיה סכום ערכי ה- x של המחוברים, וכנ"ל עבור y
- אבל ניתן ללמד את הקומפיילר את כוונתנו, ואז נוכל לכתוב פונקציות שהקריאה אליהן תהייה באופן הבא:

```
void main()
{
    Point p1(5, 6), p2(7, 8);
    Point p3 = p1 + p2;

    p3.show();
}
```

p1: אובייקט מפעיל

אופרטור +: השיטה

p2: הפרמטר

```
void main()
{
    Point p1(5, 6), p2(7, 8);
    Point p3 = p1.add(p2);

    p3.show();
}
```

p1: אובייקט מפעיל

add: השיטה

p2: הפרמטר

העמסת אופרטורים דגשים

- לימוד הקומפיילר כיצד לבצע פעולה אינטואיטיבית נקראת "העמסת אופרטורים" (functions overloading)
- נעמיס רק אופרטורים שההגדרה שלהם אינטואיטיבית וברורה!
- דוגמאות:
  - חיבור 2 נקודות
  - חיבור נקודה עם מספר שלם
  - חיבור מלבן עם מלבן
- דוגמאות שיש להגדירן היטב:
  - חיבור 2 עיגולים: מי יהיה המרכז של העיגול החדש?
  - חיבור מספר שלם לשחקן כדורסל: הגדלת מספר הנקודות שקלע?
- דוגמא לא טובה:
  - שחקן כדורסל + שחקן כדורסל

העמסת אופרטורים תחביר

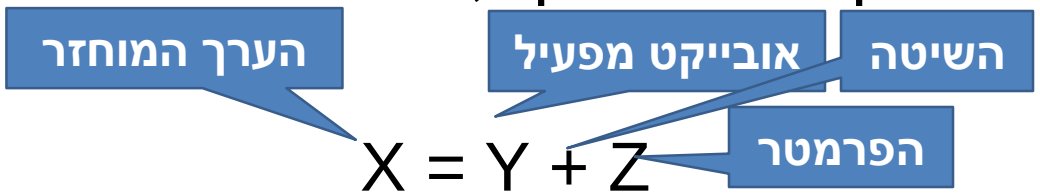
כאשר מעמיסים אופרטור, שם הפונקציה תמיד יתחיל ב- operator, ובשימוש נשתמש רק בסימן

```
class Point
{
    int x, y;

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

    Point add(const Point& other) const
    {
        return Point(x + other.x, y + other.y);
    }

    Point operator+(const Point& other) const
    {
        return Point(x + other.x, y + other.y);
    }

    void show() const
    {
        cout << "(" << x << ", " << y << ")\n";
    }
};
```

שם הפונקציה

העמסת אופרטורים תחביר (2)

- כדי לקבוע מהי חתימת הפונקציה המדוייקת, נבצע תמיד השוואה ל- int:



- בעקבות פעולה זו:
  - הפונקציה מחזירה ערך חדש X, שהוא החיבור בין Y ל- Z, לכן הפונקציה צריכה להחזיר ערך חדש
  - האובייקט המפעיל Y לא השתנה, לכן הפונקציה תהייה const
  - הפרמטר Z לא השתנה, לכן הוא יהיה const ו- by ref

```
Point operator+(const Point& other) const
```

העמסת פונקציות להעמסת אופרטור

```
class Point
{
    int x, y;

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

    Point add(const Point& other) const
    {
       return Point(x + other.x, y + other.y);
    }

    Point operator+(const Point& other) const
    {
       return Point(x + other.x, y + other.y);
    }

    Point operator+(int add) const
    {
       return Point(x + add, y + add);
    }

    void show() const
    {
       cout << "(" << x << ", " << y << ")\n";
    }
};
```

```
void main()
{
    Point p1(5, 6);
    Point p2 = p1 + 3;

    p2.show();
}
```

```
(8, 9)
```

העמסת אופרטור

העמסת פונקציה

דוגמה העמסת האופרטור -

```
class Point
{
    int x, y;

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

    Point operator+(const Point& other) const {return Point(x + other.x, y + other.y);}
    Point operator+(int add)            const {return Point(x + add, y + add);}

    Point operator-(const Point& other) const { return Point(x - other.x, y - other.y); }
    Point operator-(int num)            const { return Point(x - num, y - num); }

    void show() const { cout << "(" << x << ", " << y << ")\n"; }
};

void main()
{
    Point p1(5, 6), p2(7, 8), p3;

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

    p3 = p2 - p1;
    p3.show();
}
```

```
(-2, -2)
(2, 2)
```

כמובן שבחיסור יש משמעות לסדר: מי האובייקט המפעיל ומי הפרמטר

העמסת אופרטור כפונקציית friend מוטיבציה

- לימדנו את הקומפיילר לבצע חיבור/חיסור בין נקודה למספר:
  - הנקודה היא האובייקט המפעיל, והמספר הוא הפרמטר
- ניתן ללמד את הקומפיילר גם לבצע את הפעולה בין מספר לנקודה, והפעם המספר הוא האובייקט המפעיל והנקודה היא הפרמטר..
  - מאחר ואין לנו גישה לקוד של המחלקה int, נממש זאת ע"י כתיבת פונקציה גלובלית:

```
Point operator+(int num, const Point& p)
{
    return Point(p.x + num, p.y + num);
}
void main()
{
    Point p1(5, 6), p2;

    p2 = p1 + 4;
    p2 = 4 + p1;
}
```

במימוש אופרטור באמצעות פונקציה גלובלית, הפרמטר הראשון הוא המשתנה שמשמאל לאופרטור, והפרמטר השני או המשתנה שמימינו

אבל השאיפה היא שבתכנות מונחה עצמים כל דבר יהיה חלק ממחלקה, לכן נרצה להכניס את פונקציה גלובלית זו לתוך המחלקה Point

העמסת אופרטור כפונקציית friend

- פונקציה זו קשורה לוגית ל- Point ולכן נרצה שהקוד שלה יהיה כתוב בתוך המחלקה Point
  - לכן היא תהיה כפונקציית friend במחלקה Point

תזכורת: פונקציית friend היא פונקציה גלובלית הכתובה בתוך מחלקה, יכולה לגשת לתכונות ה- private ואינה יכולה להיות const (אין אובייקט מפעיל)

```
class Point
{
    int x, y;

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

    Point operator+(const Point& other) const {return Point(x + other.x, y + other.y);}
    Point operator+(int add)            const {return Point(x + add, y + add);}

    friend Point operator+(int num, const Point& p) { return Point(p.x + num, p.y + num);}

    void show() const { cout << "(" << x << ", " << y << ")\n"; }
};
```

במידה וממשים פונקציית friend מחוץ לגבולות המחלקה, לא נציין שוב friend ולא נציין שיוך למחלקה:

```
friend Point Point::operator+(int num, const Point& p) {…}
```

העמסת אופרטורים סימטריה בקוד

```
void main()
{
    Point p1(5, 6), p2;

    p2 = p1 + 4;
    p2 = 4 + p1;
}
```

- פעולות החיבור בדוגמא סימטריות, לכן נהוג שגם הקוד יהיה כתוב באותו סגנון
- לכן נכתוב את הפונקציה המחברת נקודה עם שלם גם כ- friend (לא חובה...)

```
class Point
{
    int x, y;

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

    Point operator+(const Point& other) const {return Point(x + other.x, y + other.y);}
    Point operator+(int add)            const {return Point(x + add, y + add);}

    friend Point operator+(int num, const Point& p) { return Point(p.x + num, p.y + num);}
    friend Point operator+(const Point& p, int num) { return num+p;}

    void show() const { cout << "(" << x << ", " << y << ")\n"; }
};
```

חייבים להוריד מימוש זה, אחרת נקבל שגיאת ambiguity: הקומפיילר לא ידע לאיזו גרסא לפנות...

מניעת שכפול: קריאה לפונקציה שמעל

מה קורה קודם?

- בהינתן הביטוי p1+p2+p3, למה הוא שקול?

p1+(p2+p3)

(p1+p2)+p3

בהעמסת אופרטורים הקומפיילר שומר על קדימויות חשבון: כפל וחילוק לפני חיבור וחיסור, וסוגריים לפני הכל

- לצורך הדוגמה, נוסיף הדפסות למימוש האופרטור:

```
Point operator+(const Point& other) const
{
    cout << "Adding to (" << x << ", " << y << ") the point ("
                << other.x << ", " << other.y << ")\n";
    return Point(x + other.x, y + other.y);
}
```

- main לדוגמה:

```
void main()
{
    Point p1(5, 5), p2(6, 6), p3(7, 7);
    p1 + p2 + p3;
}
```

```
Adding to (5, 5) the point (6, 6)
Adding to (11, 11) the point (7, 7)
```

אופרטור השמה

- ביצוע השמה בין שני משתנים רגילים מעתיק את ערך המשתנה שמימין לסימן ההשמה למשתנה שמשמאלו
- עבור אובייקטים ניתן לנו במתנה אופרטור השמה, המעתיק את כל שדות ה- R-Value ל- L-Value

```
int main()
{
    Point p1(5, 6), p2;

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

    p1.setX(8);
    p1.show();
    p2.show();
}
```

```
(5, 6)
(5, 6)
```

```
(8, 6)
(5, 6)
```

לתוך p2 יכנסו הערכים (5, 6)

לשני האובייקטים ערכים זהים כרגע אך הם בלתי תלויים

- כמובן שניתן לדרוס את המימוש שניתן במתנה

אופטור השמה הבעיה במימוש הניתן במתנה

```
class Person
{
    int id;
    char* name;
public:
    Person(int id, const char* name) {...}
    Person(const Person& other)      {...}
    ~Person()                { delete[]name; }

    void show() const
    {
        cout << "Id: " << id
             << ", name: " << name << endl;
    }

    void setName(const char* name)
    {
        delete[] this->name;
        this->name = strdup(name);
    }
};
```

p2	
id:	222
name:	2000

p1	
id:	222
name:	3000

~~"gogo"~~ “koko"

```
int main()
{
    Person p1(111, "momo");
    Person p2(222, "gogo");

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

    p1.setName("koko");
    p1.show();
    p2.show();
}
```

```
Id: 222, name: gogo
Id: 222, name: gogo
```

```
Id: 222, name: koko
Id: 222, name: זבל
```

כאשר יש מצביעים במחלקה, תיתכן הבעיה של ההצבעה הכפולה, כמו הבעיתיות ב- copy c’tor

אופרטור ההשמה שניתן במתנה

```
class Person
{
    int id;
    char* name;
public:
    Person(int id, const char* name) {...}
    Person(const Person& other)      {...}
    ~Person()                { delete[]name; }

    void show() const {...}
    void setName(const char* name) {...}

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

המימוש המתקבל במתנה מבצע העתקה רדודה לכל השדות

- השיטה אינה מחזירה ערך
- הפרמטר הוא const כי אינו משתנה ע"י השיטה
- השיטה אינה const כי משנה את האובייקט המפעיל

אופרטור השמה דריסתו

```
class Person
{
    int id;
    char* name;
public:
    Person(int id, const char* name) {...}
    Person(const Person& other)      {...}
    ~Person()                        { delete[]name; }

    void show() const {...}
    void setName(const char* name) {...}

    void operator=(const Person& other)
    {
        id = other.id;

        delete[]name;
        name = strdup(other.name);
    }
};
```

לא לשכוח באופרטור ההשמה
לשחרר כל שדה שהוקצה דינאמית!

p2	
id:	222
name:	2000

p1	
id:	222
name:	3000

"gogo"

"gogo"

"momo"

```
int main()
{
    Person p1(111, "momo");
    Person p2(222, "gogo");

    p1 = p2;
}
```

הפעם השמות זהים, אבל
במקומות שונים בזיכרון, ולכן
האובייקטים בלתי תלויים

אופרטור השמה בעיה בהשמה עצמית

```
class Person
{
    int id;
    char* name;
public:
    ...

   void operator=(const Person& other)
   {
       if (this != &other)
       {
           id = other.id;

           delete[]name;
           name = strdup(other.name);
       }
   }
};

int main()
{
    Person p1(111, "momo");
    p1 = p1;
}
```

מונע שבמקרה של השמה עצמית לא יועתקו השדות מחדש, ובפרט לא יהיה שחרור זכרון שיגרום לתעופה

פה תהיה תעופה כי כבר אין ממה להעתיק

p1 / other	
id:	111
name:	1000

"momo"

הערך מועתק לעצמו ונותר ללא שינוי

אופרטור השמה תמיכה בהשמה מרובה

- עבור ה- main הבא תתקבל שגיאה:

```
void main()
{
    Point p1(5, 6), p2, p3;

    p3 = p2 = p1;
}
```

מבחינת הקומפיילר מוחזר void,
ואותו לא ניתן להשים לתוך p3

```
error C2679: binary '=' : no operator found which takes a right-hand
operand of type 'void' (or there is no acceptable conversion)
```

אופרטור השמה תמיכה בהשמה מרובה מימוש

```
class Point
{
    int x, y;

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

    const Point& operator=(const Point& other)
    {
        if (this != &other)
        {
            x = other.x;
            y = other.y;
        }
        return *this;
    }
};
```

1. הפונקציה צריכה להחזיר אובייקט מטיפוס Point
2. מחזירה אותו by ref כדי לחסוך את ההעתקה עבור הערך המוחזר
3. נחזיר אותו גם כ- const כדי להגן על האובייקט המוחזר משינויים (כי הוחזר by ref).

החזרה של האובייקט המפעיל, כלומר, זה שביצענו השמה לתוכו

אופרטור השמה מדוע הערך המוחזר הוא const

- ע"פי התקן של השפה, מומלץ לא לעדכן משתנה פעמיים באותה שורה מאחר ולא תמיד התשובה אינטואיטיבית
  - למשל:

```
int main()
{
    int x=0, y;

    y = x++ + ++x;
    cout << y << " " << x << endl;
}
```

```
2 2
```

- מאחר ובאופרטור= האובייקט כבר משתנה ואז אנחנו מחזירים אותו, נרצה לוודא שלא ניתן לבצע עליו עדכון נוסף

אופרטור השמה מדוע הערך המוחזר הוא const דוגמה

```
void resetPoint(Point& p)
{
    p.setX(0);
    p.setY(0);
}

int main()
{
    Point p1(10, 10), p2(20, 20);

    resetPoint(p1 = p2);
}
```

היינו רוצים להמנע מהסנריו הבא בו p1 מתעדכן גם באופרטור ההשמה וגם בפונקציה resetPoint

סיכום מחלקה המכילה הקצאות דינאמיות

- כאשר אחד השדות במחלקה נוצר ע"י הקצאה דינאמית יש לממש את הרביעיה הבאה:
  1. copy c'tor
  2. destructor
  3. אופרטור השמה
  4. move c'tor

- במידה ולא רוצים לממש את ה- copy c'tor או את אופרטור ההשמה נגדיר אותם ב- private או נסמנם ב- `=delete`, כדי לדרוס את אלו הניתנים במתנה, ולייצר שגיאת קומפילציה במקרה בו יהיה ניסיון לעבור דרכם
- את ה- d'tor תמיד חובה לממש!
- <u>שימו לב:</u> במחלקה הכוללת מצביעים בלבד, ולא הקצאות, יש לשים לב מתי באמת יש צורך לממש את הרביעיה

מעבר באופרטור ההשמה של אובייקט מוכל

- בהינתן מחלקה A שיש בה אובייקט מוכל B, אופרטור ההשמה הניתן במתנה של A, מבצע השמה עבור כל שדותיו של A, ובפרט עבור האובייקט המוכל B, ולכן עובר באופרטור ההשמה שלו

```
class B
{
public:
   const B& operator=(const B& other)
   {
        cout << "In B::operator=\n";
        return *this;
   }
};

class A
{
private:
   B b;
};
```

```
void main()
{
   A a1, a2;
   cout << "----------\n";
   a1 = a2;
}
```

```
----------
In B::operator=
```

copy c’tor לעומת אופרטור השמה

- על פניו, נראה כי ה- copy c’tor ואופרטור ההשמה עושים את אותן פעולות
- ההבדל הוא ש- copy c’tor מופעל אך ורק בעת יצירת אובייקט, ואופרטור ההשמה מופעל כאשר האובייקט כבר קיים

```
void main()
{
    Person p1(111, "momo");
    cout << "----------------\n";
    Person p2(p1);
    cout << "----------------\n";
    Person p3 = p1;
    cout << "----------------\n";
    p1 = p2;
}
```

copy c’tor

copy c’tor

אופרטור=

```
----------------
In Person::Person(copy)
----------------
In Person::Person(copy)
----------------
In Person::operator=
```

שימוש באופרטור השמה מה- copy c’tor

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

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

    Person(const Person& other) : name(NULL)
    {
        *this = other;
    }

    ~Person() { delete[]name; }

    const Person& operator=(const Person& other)
    {
        if (this != &other)
        {
            id = other.id;

            delete[]name;
            name = strdup(other.name);
        }
        return *this;
    }
};
```

בגרסאות ישנות של C++ יש צורך לאתחל מצביעים ל- NULL כדי שלא תהיה תעופה בשחרור. החל מגרסה C++11 זה מיותר כי תכונות מאותחלות ל-0 באופן אוטומטי

קריאה לאופרטור ההשמה

ה- copy c’tor ואופרטור ההשמה למעשה עושים את אותו הדבר, רק כל אחד מופעל בזמן שונה. כדי למנוע את שכפול הקוד נקרא לאופרטור ההשמה מה- copy c’tor (האם אפשר ההיפך?)

אופרטור אונארי לעומת אופרטור בינארי

- אופרטור אונארי עובד על משתנה אחד בלבד
  - דוגמאות:
    - אופרטור מינוס: x-
    - אופרטור ++: ++i

- אופרטור בינארי עובד עם 2 משתנים
  - דוגמאות:
    - אופרטור חיבור: x+y
    - אופרטור השמה: x=y

אופרטור מינוס

- נסתכל על הביטוי x-
  - האובייקט המפעיל הוא x
  - השיטה היא האופרטור מינוס
  - האוביקט המפעיל אינו משתנה (← המתודה תהיה const)
  - השיטה אינה מקבלת פרמטרים

- נסתכל על הביטוי y = -x
  - y מקבל את ערכו של x לאחר הפעלת אופרטור –
  - כלומר, בביטוי זה קודם יוערך הביטוי שמימין, ורק אז תוצאתו תכנס לאוביקט שמשמאל ע"י הפעלת אופרטור השמה (← האופרטור צריך להחזיר ערך)

אופרטור מינוס מימוש

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

    Point operator-() const
    {
        cout << "In Point::operator-\n";
        return Point(-x, -y);
    }

    const Point& operator=(const Point& other)
    {
        cout << "In Point::operator=\n";
        if (this != &other)
        {
            x = other.x;
            y = other.y;
        }
        return *this;
    }

    void show() const { cout << "(" << x << ", " << y << ")\n"; }
};
```

p1 נותר ללא שינוי

```
void main()
{
    Point p1(5, 6), p2;

    cout << "----------\n";
    p2 = -p1;
    cout << "----------\n";
    p2.show();
    cout << "----------\n";
    p1.show();
    cout << "----------\n";
}
```

```
------------------
In Point::operator-
In Point::operator=
------------------
(-5, -6)
------------------
(5, 6)
------------------
```

אופרטור =+

- נסתכל על הביטוי y += x
  - y הוא האובייקט המפעיל והוא משתנה בעקבות הפעלת האופרטור (← השיטה לא תהיה const)
  - השיטה היא האופרטור =+
  - הפרמטר הוא x, ואינו משתנה בעקבות הקריאה לשיטה (← הפרמטר יהיה const)
  - כדי לתמוך בהשמה מרובה: z = y+= x :
    - האופרטור =+ צריך להחזיר את y

- עבור אופרטור =- העקרונות זהים

אופרטור += מימוש

```
class Point
{
   int x, y;

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

   const Point& operator+=(const Point& other)
   {
       x += other.x;
       y += other.y;

       return *this;
   }

   const Point& operator+=(int num)
   {
       x += num;
       y += num;

       return *this;
   }

   void show() const { cout << "(" << x << ", " << y << ")\n"; }
};
```

העמסת האופרטור +=

```
void main()
{
    Point p1(5, 6), p2(7, 8), p3;

    cout << "----------------\n";
    p1 += p2;
    p1.show();
    cout << "----------------\n";
    p2 += 4;
    p2.show();
    cout << "----------------\n";
    p3 = p1 += p2;
    p1.show();
    p3.show();
}
```

```
------------------
(12, 14)
------------------
(11, 12)
------------------
In Point::operator=
(23, 26)
(23, 26)
```

אופרטור ++

- את האופרטור ++ נפריד לשני המקרים בו ניתן להשתמש בו:
  - postfix ← x++
  - prefix ← ++x
- בשני המקרים:
  - האובייקט מפעיל הוא x
  - השיטה היא האופרטור++
  - השיטה אינה מקבלת פרמטרים
- חתימת השיטה זהה בשני המקרים ולכן צריך לסמן לקומפיילר לאיזו גרסא לפנות
  - גרסאת ה- postfix תקבל כפרמטר int שלא יהיה בו שימוש
  - זהו הדבר היחידי בשפה מבחינת סינטקס שאינו הגיוני ☺

אופרטור ++ מימושים

```
class Point
{
    int x, y;

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

    const Point& operator++()
    {
        cout << "In operator++(prefix)\n";
        x++;
        y++;
        return *this;
    }

    Point operator++(int)
    {
        cout << "In operator++(postfix)\n";
        Point temp(*this);
        x++;
        y++;
        return temp;
    }

    void show() const { cout << "(" << x << ", " << y << ")\n"; }
};
```

מחזיר by ref ו- const מאחר ומחזיר את האובייקט המפעיל שנשאר חי עם סיום השיטה, ולכן אין בעיה להחזיר אותו by ref

מימוש ה- prefix מקדם את שדות האובייקט ומחזיר את האובייקט המעודכן

עבור מימוש ה- postfix נציין שיתקבל כפרמטר int (שלא ישלח בפועל)

מימוש ה- postfix מייצר אובייקט עם הערכים לפני הקידום ומחזיר אותו, ורק אז מקדם את האובייקט המפעיל

```
return Point(x++, y++);
```

רואים כי גרסת ה- prefix יותר יעילה מאחר ואינה מייצרת אובייקט!

אופרטור ++ שימוש

```
void main()
{
    Point p1(5, 6), p2;

    cout << "-------\n";
    p2 = p1++;
    p1.show();
    p2.show();
    cout << "-------\n";
    p2 = ++p1;
    p1.show();
    p2.show();
}
```

```
------------------
In operator++(postfix)
In Point::operator=
(6, 7)
(5, 6)
------------------
In operator++(prefix)
In Point::operator=
(7, 8)
(7, 8)
```

[סרטון במתנה!]

- הסרטון הבא מציג את האופרטור ++ אשר בתחילה נראה תמים ודומה לשאר האופרטורים, אך הוא מחביא בחובו את אתגר:
  - הוא ממומש גם בגרסת ה- prefix וגם בגרסת ה- postfix
  - חתימתם זהה, וצריך להתגבר על זה איכשהו מבחינה תחבירית

- הדבר ההרבה יותר משמעותי וחשוב בהקשר של אופרטור זה הוא ניתוח היעילות של כל אחד מהמימושים
  - צריך להבין היטב את ניתוח היעילות על-מנת להבין מי מהם יותר יעיל (ובאופן משמעותי) על פני השני.

-

לצפייה בסרטון >>

אופרטורים לוגיים

- כל האופרטורים הלוגיים מחזירים true או false:
  - == =! > => < =<
- חתימת העמסת האופרטורים:
  - מקבלים כפרמטר אובייקט נוסף מאותו טיפוס, שהשיטה אינה משנה, ולכן הפרמטר יועבר כ- const ו- by ref
  - אינה משנה את האובייקט המפעיל, ולכן השיטה תהייה const
  - מחזירה bool

```
bool operator == != > >= < <=(const  Point&  other) const
```

אופרטורים לוגיים דוגמה

```
class Point
{
    int x, y;

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

    bool operator==(const Point& other) const
    {
        return x == other.x && y == other.y;
    }

    bool operator!=(const Point& other) const
    {
        return !(*this == other);
    }
    void show() const {...}
};
```

```
void main()
{
    Point p1(5, 6), p2(7, 8), p3(5, 6);

    cout << (p1 == p2 ? "true" : "false") << endl;
    cout << (p1 != p2 ? "true" : "false") << endl;
    cout << (p1 == p3 ? "true" : "false") << endl;
    cout << (p1 != p3 ? "true" : "false") << endl;
}
```

```
false
true
true
false
```

העמסת האופרטור ostream מוטיבציה

- עד היום כדי להדפיס נתוני אובייקט כתבנו שיטה show והיינו צריכים לקרוא לה

```
void main()
{
    Point p(1, 1);

    cout << "The point is ";
    p.show();
    cout << endl;
}
```

```
The point is: (1, 1)
```

- היינו שמחים אם היה ניתן לבצע את הדבר הבא:

```
void main()
{
    Point p(1, 1);

    cout << "The point is " << p << endl;
}
```

- כלומר, הדפסת אובייקט באמצעות האופרטור <<
- לשם כך נממש את האופרטור <<

העמסת האופרטור ostream דגשים

- האובייקט המפעיל אינו המחלקה שאותה אנו ממשים (למשל Point), אלא אובייקט מהמחלקה ostream
- אין באפשרותנו לשנות את המחלקה ostream ו"ללמד" אותה להדפיס אובייקט מהמחלקה שלנו
- לכן נכתוב פונקציה גלובלית שתקבל כפרמטר גם את האובייקט ostream וגם את האובייקט אותו נרצה להדפיס
- מאחר ופונקציה זו קשורה לוגית למחלקה Point, נרצה שפונקציה זו תהייה כתובה במחלקה
  - כלומר תהיה פונקצית friend

```
void main()
{
    Point p(1, 1);

    cout << "The point is " << p << endl;
}
```

העמסת האופרטור מימוש

```
class Point
{
    int x, y;

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

תזכורת: אמנם הפונקציה לא משנה את האובייקט, אבל מאחר וזוהי פונקציה גלובלית, ולא שיטה, אינה יכולה להיות const

מחזירים ostream& כדי לתמוך בהדפסה מרובה (כמו אופרטור השמה)

פונקציה גלובלית שקשורה למחלקה

הפרמטר ostream מועבר by ref כדי שניתן יהיה להחזירו by ref

```
    friend ostream& operator<<(ostream& os, const Point& p)
    {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};
```

הפרמטר הוא const כי לא משנים את האובייקט שאותו מקבלים

נשים לב שזו לא פקודת cout, שכן משתנה מטיפוס ostream אינו ספציפי לטיפוס הקונסול

[סרטון במתנה!]

- הסרטון הבא מציג כיצד אפשר להדפיס אובייקט באמצעות אופרטור << כמו כל טיפוס בסיסי
- נראה את התחביר המיוחד, נראה על הדרך שימוש בפונקציית friend ונראה כמה זה מגניב!

לצפייה בסרטון <<

העמסת האופרטור istream מוטיבציה

- עד היום כדי לקלוט נתוני אובייקט היה צריך ב- main לכתוב קוד הקורא שדה-שדה
- היינו שמחים אם היה ניתן לבצע את הדבר הבא:

```
void main()
{
    Point p;

    cout << "Enter x and y coordinates: ";
    cin >> p;

    cout << "The point is " << p << endl;
}
```

```
Enter x and y coordinates: 3 4
The point is (3, 4)
```

- כלומר, קליטת נתונים ישירות לאובייקט באמצעות האופרטור <<
- לשם כך נממש את האופרטור <<

העמסת האופרטור istream דגשים

- האובייקט המפעיל אינו המחלקה שאותה אנו ממשים (למשל Point), אלא אובייקט מהמחלקה istream
- אין באפשרותנו לשנות את המחלקה istream ו"ללמד" אותה לקרוא נתוני אובייקט מהמחלקה שלנו
- לכן נכתוב פונקציה גלובלית שתקבל כפרמטר גם את האובייקט istream וגם את האובייקט אליו נרצה לקרוא את הנתונים
- מאחר ופונקציה זו קשורה לוגית למחלקה שלנו, נרצה שהפונקציה תהייה כתובה במחלקה
  - כלומר תהיה פונקצית friend

```
void main()
{
    Point p;

    cout << "Enter x and y coordinates: ";
    cin >> p;

    cout << "The point is " << p << endl;
}
```

העמסת האופרטור istream מימוש

```
class Point
{
    int x, y;

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

פונקציה גלובלית שקשורה למחלקה

מחזירים &istream כדי לתמוך בהכנסה מרובה

הפרמטר istream מועבר by ref כדי שניתן יהיה להחזירו by ref

```
    friend istream& operator>>(istream& in, Point& p)
    {
        in >> p.x >> p.y;
        return in;
    }

};
```

הפרמטר הפעם אינו const כי כן משנים את האובייקט שאותו מקבלים

אופרטור [ ]

```
class Triangle
{
    Point allPoints[3];
public:
    Point& operator[](int index) { return allPoints[index]; }

    const Point& operator[](int index) const { return allPoints[index]; }
};

int main()
{
    Triangle t;
    t[0] = Point(10, 10);
    t[1] = Point(3, 9);
    t[2] = Point(12, 24);
    cout << t[1] << endl;

    const Triangle t2;
    cout << t2[1] << endl;
}
```

בדוגמה זו האופרטור מחזיר הפניה לאיבר ספציפי במערך הנקודות באובייקט

מימוש זהה למימוש העליון אך מתחייב שאינו משנה את האובייקט המפעיל

פניה לגרסה הלא-const של המחלקה וקבלת הפניה לאובייקט שלתוכו עושים השמה

פניה לגרסה ה-const של המחלקה וקבלת הפניה לאובייקט שאותו מדפיסים

במידה ולא הייתה קיימת גרסת ה- const, הפניה לאופרטור [] לא הייתה עוברת קומפילציה

```
(3, 9)
(0, 0)
```

אופרטור [ ] דגשים

- האופרטור מאפשר לפנות לשדותיו של אובייקט באמצעות [ ]
  - משמש רבות לקבלת איבר במערך פנימי של מחלקה (כמו בדוגמה הקודמת)
- לרוב הפרמטר יהיה משתנה מטיפוס int, למרות שיכול להיות מכל טיפוס
- נרצה לתמוך באופרטור זה משני צידי ההשמה לכן נממשו בשתי גרסאות:
  - x = p[i]
  - p[i] = x ← לכן עליו להחזיר reference לתכונה אותה ישנה

אופרטור [ ] פרמטר שאינו בהכרח int

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

    int& operator[](int index)
    {
        cout << "In operator[](int)\n";
        return index == 0 ? x : y;
    }

    int& operator[](char index)
    {
        cout << "In operator[](char)\n";
        return index == 'x' ? x : y;
    }

    void show() const {...}
};
```

```
int main()
{
    Point p1(5, 6);

    cout << p1[0] << endl;
    cout << p1['y'] << endl;

    p1[1] = 8;
    p1.show();
}
```

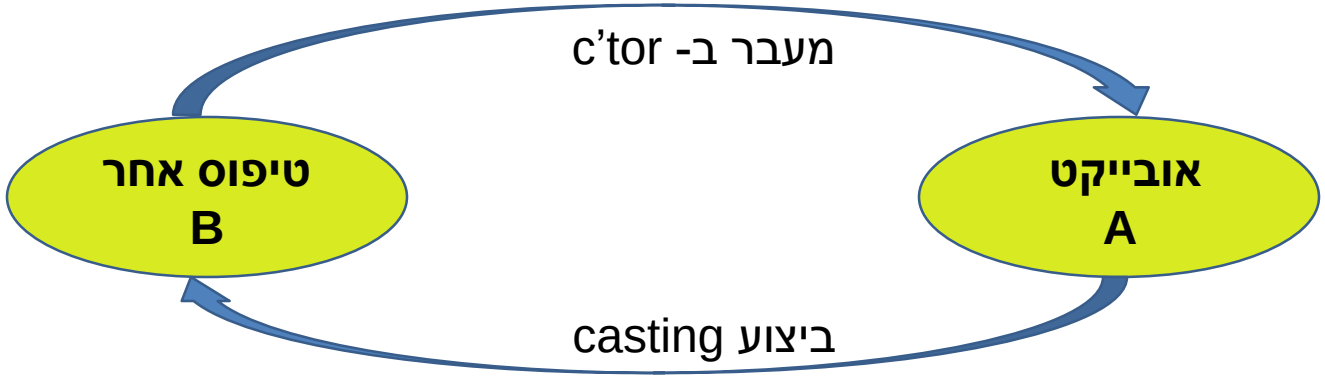
פניה לאופרטור [ ]

```
In operator[](int)
5
In operator[](char)
6
In operator[](int)
(5, 8)
```

ref לערך המוחזר, כי יתכן ונרצה לשנותו

אופרטור casting מוטיבציה

- כאשר למדנו על c’tor'ים ראינו שאם יש פונקציה המצפה לקבל אובייקט A, ושולחים לה טיפוס B, הקומפיילר מנסה לבצע המרה
- כלומר, מנסה לחפש c’tor של A המקבל כפרמטר B, וכך לייצר אובייקט זמני



- יתכן המקרה ההפוך: שדווקא נרצה בהינתן אובייקט מסויים, לקבל טיפוס אחר
- לשם כך רצה לבצע casting

אופרטור casting שימוש ומימוש

```
void foo(int x)
{
    cout << "In foo x=" << x << endl;
}

void main()
{
    Point p(3, 4);
    foo(p);
}
```

```
In foo x=7
```

שליחת Point לפונקציה המצפה לקבל int

```
class Point
{
    int x, y;

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

    operator int() const
    {
        return x + y;
    }

};
```

כמובן שניתן לממש יותר מאופרטור casting אחד

לא משנים את האובייקט המפעיל

במימוש אופרטור casting לא נציין ערך מוחזר, הוא צריך להיות מטיפוס ההמרה

אופרטור ה- casting אינו יכול לקבל פרמטרים

אופרטורי casting מרובים (1)

```
class Person
{
    int id;
    char name[10];
public:
    Person(int id, const char* name) {...}

    operator int()          const  { return id; }
    operator const char*() const  { return name; }
};

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

    cout << (int)p1 << endl;
    cout << (const char*)p1 << endl;
    cout << p1 << endl;
}
```

```
111
gogo
111
```

לאופרטור casting ל- int יש עדיפות על פני אופרטור ה- casting ל- const char*

אופרטורי casting מרובים (2)

```
class Person
{
    int id;
    char name[10];
public:
    Person(int id, const char* name) {...}

    operator int()          const  { return id; }
    operator double()       const  { return id; }
    operator const char*() const  { return name; }
};

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

    cout << p1 << endl;
}
```

לאופרטור casting ל- int אין עדיפות על-פני אופרטור casting ל- double ולכן ישנה שגיאת קומפילציה:

more than one operator "<<" matches these operands: multiple operator casting

אופרטורי casting מרובים (3)

```
class Person
{
    int id;
    char name[10];
public:
    Person(int id, const char* name) {...}

    operator int()          const  { return id; }
    operator double()       const  { return id; }
    operator const char*() const  { return name; }

    friend ostream& operator<<(ostream& os, const Person& p)
    {
        os << "Name: " << p.name << ", Id: " << p.id << endl;
        return os;
    }
};

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

    cout << p1 << endl;
}
```

```
Name: gogo, Id: 111
```

במידה וקיים מימוש לאופרטור <<
לא יהיה ניסיון ל- casting

אופרטור סוגריים ()

- אופרטור מיוחד שיכול לקבל ולהחזיר מה שכותב הקוד יבחר

```
class Point
{
    int x, y;

public:
    Point(int x = 0, int y = 0) : x(x), y(y)
    {
        cout << "In c'tor: " << *this << endl;
    }
    void operator()(int x, int y)
    {
        this->x = x;
        this->y = y;
        cout << "In operator(): " << *this << endl;
    }
    friend ostream& operator<<(ostream& os, const Point& p)
    {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};
```

שם האופרטור

הפרמטרים שהאופרטור מקבל

הפעלת האופרטור ()

```
void main()
{
    Point p(3, 4);
    p(5, 6);
}
```

```
In c'tor: (3, 4)
In operator(): (5, 6)
```

משמש בעיקר למימוש Object Function בפרק של ה- STL

המושגים L-Value ו- R-Value תזכורת

- L-Value הינו משתנה שיש לו שם וניתן לגשת אליו ישירות
- R-Value הינו משתנה זמני שאין לו שם

- ראינו עד כה את ה- move c’tor
- ראינו כי הקומפיילר יודע לזהות מקרה בו פונקציה מקבלת כפרמטר אובייקט זמני (שתיכף ימות)

תזכורת המחלקה Person

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

    Person(const Person& other)
    {
        this->name = new char[strlen(other.name) + 1];
        strcpy(this->name, other.name);
        cout << "In Person::Person(copy) name is " << name << " at address " << (void*)name << "\n";
    }

    Person(Person&& other)
    {
        name = other.name;
        cout << "In Person::Person(move) name is " << name << " at address " << (void*)name << "\n";

        other.name = nullptr;
    }
    ...
};
```

תזכורת המחלקה Person

```
class Person
{   ...
    ~Person()
    {
        cout << "In Person::~Person ";
        if (name != nullptr)
          cout << "delete " << name << " ";
        cout << "at address " << (void*)name << "\n";
        delete[]name;
    }
    const Person& operator=(const Person& other)
    {
        if (this != &other)
        {
            delete[]name;
            this->name = new char[strlen(other.name) + 1];
            strcpy(this->name, other.name);
        }
        cout << "In Person::operator= " << name << " at address " << (void*)name << "\n";
        return *this;
    }
    void print() const
    {
       cout << "Person's name is " << name << endl;
    }
    ...
};
```

move operator=

```
Person foo()
{
    return Person("yoyo");
}

int main()
{
    Person p1("gogo");
    p1 = foo();
}
```

נעבור בו כאשר תהיה השמה מאובייקט שהוא R-Value

השמה מאובייקט שתיכף ימות – R-Value

```
class Person
{   ...


    const Person& operator=(Person&& other)
    {
        if (this != &other)
        {
            delete[]name;
            this->name = other.name;
            other.name = nullptr;
        }
        cout << "In Person::operator=&& " << name
                        << " at address " << (void*)name << "\n";
        return *this;
    }
};
```

move assignment operator

שימוש move operator=

```
Person foo()
{
    return Person("yoyo");
}

int main()
{
    Person p1("gogo");
    p1 = foo();
    cout << "-----------\n";
}
```

```
In Person::Person name is gogo at address 0157F0C8
In Person::Person name is yoyo at address 0157EE60
In Person::operator=&& yoyo at address 0157EE60
In Person::~Person at address 00000000
-----------
In Person::~Person delete yoyo at address 0157EE60
```

move operator= מימוש אלטרנטיבי

```
const Person& operator=(Person&& other)
{
    if (this != &other)
        std::swap(name, other.name);

    cout << "In Person::operator=&& " << name
                << " at address " << (void*)name << "\n";
    return *this;
}
```

מחליפה בין הכתובות

```
Person foo()
{
    return Person("yoyo");
}
int main()
{
    Person p1("gogo");
    p1 = foo();
    cout << "-----------\n";
}
```

```
In Person::Person name is gogo at address 003BF438
In Person::Person name is yoyo at address 003BF400
In Person::operator=&& yoyo at address 003BF400
In Person::~Person delete gogo at address 003BF438
-----------
In Person::~Person delete yoyo at address 003BF400
```

std::swap ו- std::move

- std:swap מקבלת שני פרמטרים ומחליפה את ערכיהם
  - הרבה יותר יעיל מאשר להתחיל למחוק ולהעתיק את הערכים

- std::move מקבלת כפרמטר משתנה ומחזירה אותו כ- r-value
  - שימושי כאשר ישנן שתי מתודות אחת המקבלת r-value והשניה l-value ונרצה להיכנס דווקא לראשונה
  - הפעולה שקולה ל- casting לטיפוס כ- &&

דוגמת שימוש std::swap -ו std::move

```
void swap1(Person& p1, Person& p2)
{
    Person tmp = p1;
    p1 = p2;
    p2 = tmp;
}
void swap2(Person& p1, Person& p2)
{
    Person tmp = std::move(p1);
    p1 = std::move(p2);
    p2 = std::move(tmp);
}
int main()
{
    Person p1("gogo"), p2("momo");

    p1.print();
    p2.print();
    cout << "---------\n";

    cout << "swap 1:\n";
    swap1(p1, p2);
    p1.print();
    p2.print();
    cout << "---------\n";

    cout << "swap 2:\n";
    swap2(p1, p2);
    p1.print();
    p2.print();
    cout << "---------\n";
}
```

```
In Person::Person name is gogo at address 015FED28
In Person::Person name is momo at address 015FEE40
Person's name is gogo
Person's name is momo
---------
swap 1:
In Person::Person(copy) name is gogo at address 015FF0E0
In Person::operator= momo at address 015FEEB0
In Person::operator= gogo at address 015FEF20
In Person::~Person delete gogo at address 015FF0E0
Person's name is momo
Person's name is gogo
---------
swap 2:
In Person::Person(move) name is momo at address 015FEEB0
In Person::operator=&& gogo at address 015FEF20
In Person::operator=&& momo at address 015FEEB0
In Person::~Person at address 00000000
Person's name is gogo
Person's name is momo
---------
In Person::~Person delete momo at address 015FEEB0
In Person::~Person delete gogo at address 015FEF20
```

std::move ו- std::swap דוגמת שימוש

```
void swap2(Person& p1, Person& p2)
{
    Person tmp = std::move(p1);
    p1 = std::move(p2);
    p2 = std::move(tmp);
}
void swap3(Person& p1, Person& p2)
{
    std::swap(p1, p2);
}

int main()
{
    Person p1("gogo"), p2("momo");

    p1.print();
    p2.print();
    cout << "---------\n";

    cout << "swap 2:\n";
    swap2(p1, p2);
    p1.print();
    p2.print();
    cout << "---------\n";

    cout << "swap 3:\n";
    swap3(p1, p2);
    p1.print();
    p2.print();
    cout << "---------\n";
}
```

ניתן לראות שהמימוש של std::swap מפעיל את std::move

```
In Person::Person name is gogo at address 0001EF78
In Person::Person name is momo at address 0001F138
Person's name is gogo
Person's name is momo
---------
swap 2:
In Person::Person(move) name is gogo at address 0001EF78
In Person::operator=&& momo at address 0001F138
In Person::operator=&& gogo at address 0001EF78
In Person::~Person at address 00000000
Person's name is momo
Person's name is gogo
---------
swap 3:
In Person::Person(move) name is momo at address 0001F138
In Person::operator=&& gogo at address 0001EF78
In Person::operator=&& momo at address 0001F138
In Person::~Person at address 00000000
Person's name is gogo
Person's name is momo
---------
In Person::~Person delete momo at address 0001F138
In Person::~Person delete gogo at address 0001EF78
```

ביחידה זו למדנו:

- מהי העמסת אופרטורים ומוטיבציה
- אופרטור אונארי לעומת אופרטור בינארי
- העמסת אופרטורים:
  - אופרטורים + ו- -
  - אופרטורים כפונקציות friend
  - אופרטור השמה
    - שימוש באופרטור השמה מ- copy c'tor
    - ההבדל בין copy c'tor לאופרטור השמה
  - אופרטור מינוס (- אונארי)
  - אופרטור =+, =-
  - אופרטורים ++ ו- -- (prefix, postfix)
  - אופרטור [ ]
  - אופרטורים לוגיים: ==, >, < , =<, =>
  - אופרטור <<, >>
  - אופרטור casting
  - אופרטור ()
- אופרטור השמה המקבל &&
- std:swap ו- std::move