תכנות מכוון עצמים ו- C++
יחידה 07
הורשה

קרן כליף

ביחידה זו נלמד:

- מוטיבציה להורשה
- ההרשאה protected
- מעבר בבנאים בהורשה
- מעבר ב- d’tor בהורשה
- ה- copy c’tor שניתן במתנה
- אופרטור ההשמה שניתן במתנה
- שליחת יורש לפונקציה המצפה לקבל בסיס
- הורשה בשרשרת
- הורשה מרובה
  - הורשה מרובה עם אב-קדמון משותף
- סדר האתחול
- הורשת protected ו- private

הורשה לעומת הכלה

```
class Person
{
    char name[20];
    int height;
    double weight;
};

class Canibal
{
    Person me;
    Person* lastEaten;
};

class Canibal : public Person
{
    //Person me;
    Person* lastEaten;
};
```

קניבליזם

התופעה בה בני אדם אוכלים בשר אדם

[ויקיפדיה]

תכונות אדם:
שם
גובה
משקל

תכונות קניבל:
שם
גובה
משקל
האדם האחרון שאכל

מה הקשר בין אדם לקניבל?

קניבל הוא סוג של אדם וגם מכיל אדם

דוגמת הבדל בין הורשה להכלה

- לחתול נשמור את צבעו ואורך השפם שלו
  - לחתול-רחוב נשמור בנוסף את מספר הקרבות שלו
  - לחתול-סיאמי נשמור בנוסף את האוכל המועדף שלו
- חתול-רחוב וחתול-סיאמי הם סוג של חתול, כלומר מרחיבים אותו מבחינת הנתונים, לכן נשתמש בהורשה כדי לתאר מחלקות אלו

- לכלב נשמור את מספר הפעמים בדקה שהוא מכשכש בזנב והאם הוא רודף אחרי חתולים
  - יש 2 סוגי כלבים: לאסי ופודל
- מאחר וללאסי ולפודל אין נתונים נוספים המייחדים אותם, נחזיק את נתון סוג הכלב בתור שדה במחלקה המתארת כלב
- אם היינו אומרים שלכלב פודל יש לשמור גם את התדירות בה הוא מקבל דלקות אוזניים, היינו משתמשים בהורשה

מהי הורשה?

- הורשה היא הרחבה של מחלקה מסוימת
  - אם קיים היחס B הוא סוג A, אז B יורש מ- A
  - למשל SiamiCat הוא סוג של Cat, הרחבה שלו
  - נשים לב לא להתבלבל בין הכלה לבין הורשה!
    - אם B הוא חלק מהנתונים של A, אז זו הכלה
    - למשל Date מוכל בתוך Person מאחר ותאריך לידה זה חלק מנתוני Person
    - למשל קניבל הוא גם סוג של בן-אדם (ירושה) וגם מכיל בן-אדם (הכלה)

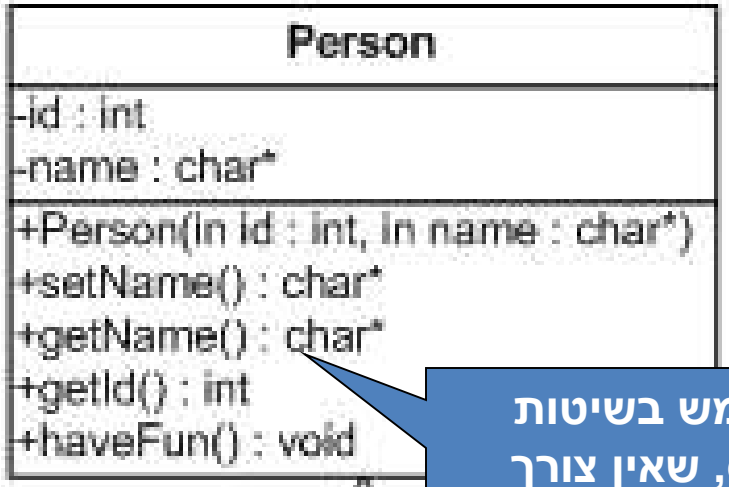
- הורשה היא יצירת מחלקה בעלת כל התכונות והשיטות של המחלקה ממנה ירשה ויכולה להשתמש בהן (בהתאם להרשאה)

דוגמה Person ו- Student
תרשים Class Diagram

המחלקה שיורשים ממנה
נקראת "בסיס" (base class)
והמחלקה היורשת נקראת derived

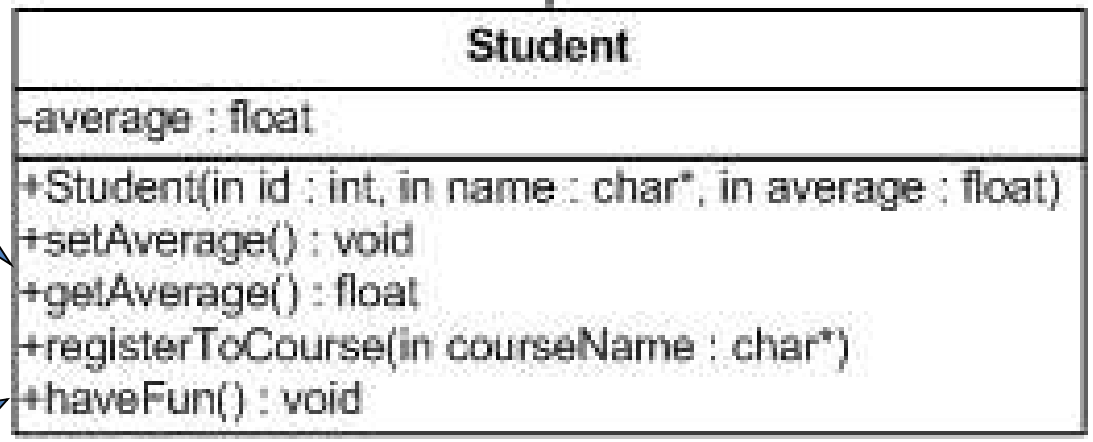
- הוא סימון ל- private

+ הוא סימון ל- public



Student ישתמש בשיטות
שהוגדרו בבסיס, שאין צורך
לשנותן, ולא יממש אותן מחדש

סימון של הורשה



Student מממש בעצמו
את השיטות שלא הוגדרו
ב- Person

Student דורס את
המימוש של haveFun
שמומש ב- Person

מחלקת הבסיס בה נשתמש לדוגמה

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

public:
    Person(int id, const char* name) {…}
    Person(const Person& other) {…}
    ~Person() {…}

    const Person& operator=(const Person& other){…}

    void haveFun() const { cout << "Yeah! Going to the sea!\n"; }

    friend ostream& operator<<(ostream& os, const Person& p) {…}
};
```

הורשה תחביר המחלקה היורשת

```
class Student : public Person
{
private:
    float average;
public:
    Student(int id, const char* name, float average)
    {
        this->id = id;
        this->name = strdup(name);
        this->average = average;
    }

    void registerToCourse(const char* courseName) const
    {
        cout << name << " registers to '" << courseName << "'\n";
    }

    void haveFun() const
    {
        cout << "Yeah! Going to the sea!\n";
        cout << "Yeah! Doing homework!\n";
    }
};
```

תחביר ההורשה

שכפול עם הקוד של הבסיס!!!

הגדרת שיטה שלא מוגדרת בבסיס

דריסת שיטה המוגדרת בבסיס, אין רמז סינטקטי

שכפול, מופיע גם במימוש של הפונקציה בבסיס

שימו לב: קוד בשקף זה אינו מתקמפל!!
השינויים הדרושים יוסברו בהמשך

מוטיבציה להורשה דוגמת Person ו- Student

- ל- Person יש שם ו- ת.ז, הוא יודע להחזיר מחרוזת עם נתוניו ולהחזיר את שמו
- ל- Student יש נתונים זהים וממוצע, ובנוסף הוא יודע להירשם לקורסים
- ייצור 2 מחלקות אלו יגרור שיכפול בקוד וייצר בעיית תחזוקה
  - אם נחליט שלכל Person צריך לשמור גם את תאריך הלידה שלו, נצטרך לתחזק זאת ב- 2 מקומות שונים..
- מאחר ו- Student הוא סוג של Person נרצה להשתמש במנגנון הורשה

לצורך <u>שימוש חוזר</u> בקוד
ב- Person יש שיטות ותכונות שרלוונטיות גם ל- Student ולא נרצה לשכפלן

לצורך <u>דריסת פעולות</u> מסוימות בלי לשנות את הקוד המקורי
במחלקה Person השיטה haveFun כתובה בצורה מסוימת, ובמחלקה Student היינו רוצים אותה באופן שונה

יתכן ואין לנו גישה לקוד המקורי, ובכל זאת היינו רוצים לבצע בו שינויים

הרשאות

- מחלקה יורשת מכילה את כל תכונות ושיטות הבסיס, אך לא תוכל לגשת אליהם ישירות במידה והוגדרו בבסיס כ- private

```
void Student::registerToCourse(const char* courseName) const
{
    cout << name << " registers to '" << courseName << "'\n";
}
```

- מחלקה יורשת יכולה לגשת ישירות לכל תכונה או שיטה שהוגדרה בבסיס כ- public

- הבעיה:
  - לא נרצה להגדיר את כל תכונות הבסיס כ- public רק כדי שהמחלקה היורשת תוכל לגשת אליהם
  - בכל זאת נרצה שהמחלקה היורשת תוכל לגשת לשדות ולתכונות שרלוונטיים עבורה

הרשאת protected

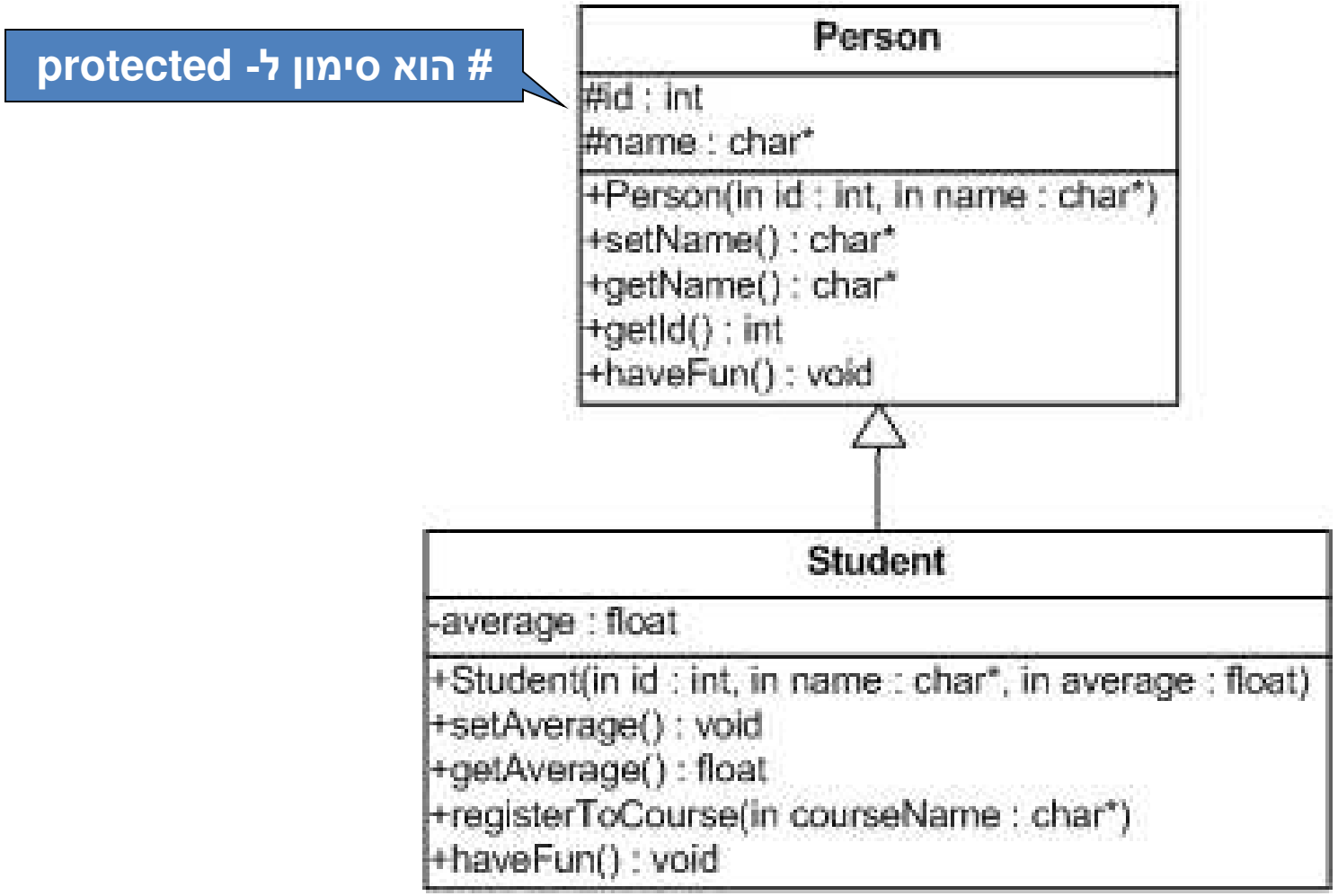
- כדי לפתור בעיה זו קיימת ההרשאה protected

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

public:
    ...
};
```

- תכונה או שיטה המוגדרת תחת ההרשאה protected מאפשרת גישה ישירה לתכונות ולשיטות במחלקה עצמה ובמחלקות היורשות בלבד
- כלפי חוץ, זה כמו private
- בצורה זו אנו לא חושפים את תכונות המחלקה כלפי חוץ, ויחד עם זאת מאפשרים למחלקות יורשות לגשת ישירות לתכונות ולשיטות הרלוונטיים עבורם

דוגמת Person ו- Student תרשים UML



הפעלת שיטה מהבסיס ביורש

- כדי להימנע משכפול קוד של הבסיס ביורש, ניתן בשיטה של היורש להפעיל ישירות שיטה שמומשה בבסיס:

```
void Student::haveFun() const
{
    cout << "Yeah! Going to the sea!\n";
    Person::haveFun();
    cout << "Yeah! Doing homework!\n";
}
```

הפעלת שיטה הממומשת בבסיס

- <u>היתרון</u>: כל שינוי בשיטה במחלקת הבסיס ישפיע מיידית על המימוש של המחלקה היורשת, ללא צורך עדכון הקוד
- כנ"ל עבור c'tor'ים...

הפעלת שיטה ב- main

- למרות שדרסנו מימוש של שיטה מסויימת בבן, ניתן לפנות למימוש שבאב:

```
void main()
{
    Student s(111, "momo", 98.5);

    s.haveFun();
    cout << "-------------\n";
    s.Person::haveFun();
}
```

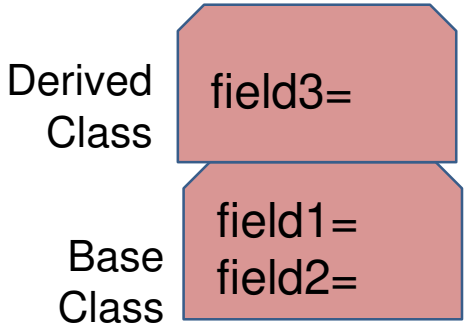
מפעיל את השיטה שבבן

מפעיל את השיטה שבאב

```
Yeah! Going to the sea!
Yeah! Doing homework!
-------------
Yeah! Going to the sea!
```

מעבר בבנאים בהורשה

- כאשר יוצרים אובייקט עוברים בבנאי
- כאשר יוצרים אובייקט מטיפוס מחלקה שיש לה בסיס, צריך לעבור קודם בבנאי של הבסיס, כדי לבצע אתחול של חלק הבסיס, ורק לאחר מכן עוברים בבנאי של המחלקה היורשת
  - כמו שבבניית בניין בונים קודם את הבסיס

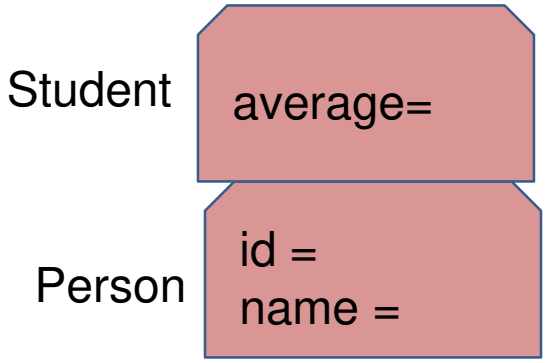


קריאה לבנאי הבסיס תחביר

קריאה ל- c'tor של הבסיס המקבל
כפרמטר ראשון int וכפרמטר שני מחרוזת.
אם לא קיים בנאי כזה תתקבל שגיאת קומפילציה.

```
Student::Student(int id, const char* name, float average)
        : Person(id, name)
{
    /*this->id = id;
    this->name = strdup(name);*/
    this->average = average;
}
```

מיותר מאחר
ואתחול זה מבוצע
ב- c'tor של בסיס



- מבחינת התחביר, בבנאי של המחלקה היורשת יש לקרוא לאחד הבנאים של מחלקת הבסיס בצורה מפורשת בשורת האתחול
- אם לא קראנו לאחד מבנאי הבסיס, הקומפיילר ינסה לעבור ב- default c'tor של הבסיס, ובמידה ואינו קיים, תתקבל שגיאת קומפילציה:

no appropriate default c'tor availabale

מעבר בין בנאים בהורשה דוגמה

```
class A
{
    int x;
public:
    A(int x = 0) { cout << "In A::A x=" << x << endl; }
    void foo() const { cout << "In A::foo\n"; }
};

class B : public A
{
public:
    B() { cout << "In B::B(ver 1)\n"; }
    B(int x) : A(x) { cout << "In B::B(ver 2)\n"; }

    void foo() const
    {
        A::foo();
        cout << "In B::foo\n";
    }
};
```

מעבר בבנאי הבסיס, למרות שלא ציינו זאת באופן מפורש!

הפעלת בנאי של הבסיס

```
In A::A x=5
In B::B(ver 2)
----------
In A::A x=0
In B::B(ver 1)
----------
In A::foo
In B::foo
----------
```

```
void main()
{
    B b1(5);
    cout << "----------\n";
    B b2;
    cout << "----------\n";
    b1.foo();
    cout << "----------\n";
}
```

הקומפיילר מזהה שהאובייקט מטיפוס המחלקה היורשת, ומאחר ויש בה מימוש לשיטה, מפעיל מימוש זה

הורשה מעבר ב- d’tor

דוגמה איך לא הורסים בניין >>

- מאחר וסדר הריסת האובייקטים הפוך לסדר יצירתם, עם היציאה מה- d’tor של המחלקה היורשת, הקומפיילר עובר באופן אוטומטי ב- d’tor של הבסיס

```
class A
{
    int x;
public:
    A(int x = 0) { cout << "In A::A x=" << x << endl; }
    ~A() { cout << "In A::~A\n"; }
};

class B : public A
{
public:
    B(int x) : A(x) { cout << "In B::B\n"; }
    ~B() { cout << "In B::~B\n"; }
};
```

```
void main()
{
    B b(5);
    cout << "----------\n";
}
```

```
In A::A x=5
In B::B
----------
In B::~B
In A::~A
```

מעבר ב- copy c'tor של הבסיס דוגמה

```
class A
{
    int x;
public:
    A(int x = 0) { cout << "In A::A x=" << x << endl; }
    A(const A& other) { cout << "In A::A(copy)\n"; }
    ~A() { cout << "In A::~A\n"; }
};

class B : public A
{
public:
    B() { cout << "In B::B(ver 1)\n"; }
    B(int x) : A(x) { cout << "In B::B(ver 2)\n"; }
    ~B() { cout << "In B::~B\n"; }
};
```

```
void main()
{
    B b1(5);
    cout << "-----\n";
    B b2(b1);
    cout << "-----\n";
}
```

```
In A::A x=5
In B::B(ver 2)
----------
In A::A(copy)
----------
In B::~B
In A::~A
In B::~B
In A::~A
```

הורשה מעבר ב- copy c’tor

- כאשר יש מחלקה יורשת, ה- copy c’tor שמקבלים במתנה עובר ראשית ב- copy c’tor של הבסיס
- אם נממש בעצמנו את ה- copy c’tor של היורש, עלינו לקרוא בשורת האתחול לבנאי כלשהו של הבסיס, לרוב ה- copy c’tor
- במידה ולא נקרא לבנאי כלשהו של הבסיס, יהיה ניסיון לפנות ל- default c’tor, ותתקבל שגיאת קומפילציה אם אינו קיים

מימוש copy c'tor במחלקה יורשת דוגמה

```
class A
{
    int x;
public:
    A(int x = 0) { cout << "In A::A x=" << x << endl; }
    A(const A& other) { cout << "In A::A(copy)\n"; }
    ~A() { cout << "In A::~A\n"; }
};

class B : public A
{
public:
    B() { cout << "In B::B(ver 1)\n"; }
    B(int x) : A(x) { cout << "In B::B(ver 2)\n"; }
    B(const B& other) : A(other) { cout << "In B::B(copy)\n"; }
    ~B() { cout << "In B::~B\n"; }
};
```

הפעלת ה- copy c'tor של הבסיס. הקומפיילר יודע להתייחס רק לחלק הבסיס של האובייקט

```
void main()
{
    B b1(5);
    cout << "-----\n";
    B b2(b1);
    cout << "-----\n";
}
```

```
In A::A x=5
In B::B(ver 2)
----------
In A::A(copy)
In B::B(copy)
----------
In B::~B
In A::~A
In B::~B
In A::~A
```

מימוש copy c'tor במחלקה יורשת דוגמה (2)

```
class A
{
    int x;
public:
    A(int x = 0) { cout << "In A::A x=" << x << endl; }
    A(const A& other) { cout << "In A::A(copy)\n"; }
    ~A() { cout << "In A::~A\n"; }
};

class B : public A
{
public:
    B() { cout << "In B::B(ver 1)\n"; }
    B(int x) : A(x) { cout << "In B::B(ver 2)\n"; }
    B(const B& other) /*: A(other)*/ { cout << "In B::B(copy)\n"; }
    ~B() { cout << "In B::~B\n"; }
};
```

במקרה בו אין קריאה מפורשת ל-
c'tor כלשהו של הבסיס,
הקומפיילר פונה ל- default c'tor
של הבסיס. במקרה ולא קיים,
תתקבל שגיאת הקומפילציה.

```
void main()
{
    B b1(5);
    cout << "-----\n";
    B b2(b1);
    cout << "-----\n";
}
```

```
In A::A x=5
In B::B(ver 2)
----------
In A::A x=0
In B::B(copy)
----------
In B::~B
In A::~A
In B::~B
In A::~A
```

אופרטור ההשמה המתקבל במתנה עבור היורש

```
class A
{
    int x;
public:
    A(int x = 0) { cout << "In A::A x=" << x << endl; }
    A(const A& other) { cout << "In A::A(copy)\n"; }
    ~A() { cout << "In A::~A\n"; }

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

class B : public A
{
public:
    B() { cout << "In B::B(ver 1)\n"; }
    B(int x) : A(x) { cout << "In B::B(ver 2)\n"; }
    B(const B& other) : A(other) { cout << "In B::B(copy)\n"; }
    ~B() { cout << "In B::~B\n"; }
};
```

```
void main()
{
    B b1(5);
    cout << "----------\n";
    B b2;
    cout << "----------\n";
    b2 = b1;
    cout << "----------\n";
}
```

```
In A::A x=5
In B::B(ver 2)
----------
In A::A x=0
In B::B(ver 1)
----------
In A::operator=
----------
In B::~B
In A::~A
In B::~B
In A::~A
```

הורשה מעבר באופרטור ההשמה של הבסיס

- כאשר יש מחלקה יורשת, אופרטור ההשמה המתקבל במתנה עובר ראשית באופרטור ההשמה של הבסיס
- אם נממש בעצמנו את אופרטור ההשמה של היורש, עלינו לקרוא לאופרטור ההשמה של הבסיס
- במידה ולא נקרא לאופרטור ההשמה של הבסיס, לא נקבל שגיאת קומפילציה, לכן חשוב לזכור לקרוא לו!

מימוש אופרטור ההשמה של היורש

```
class A
{
    int x;
public:
    A(int x = 0) { cout << "In A::A x=" << x << endl; }

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

class B : public A
{
public:
    B() { cout << "In B::B(ver 1)\n"; }
    B(int x) : A(x) { cout << "In B::B(ver 2)\n"; }

    const B& operator=(const B& other)
    {
        cout << "In B::operator=\n";
        A::operator=(other);    // same as: (A&)*this = other;
        return *this;
    }
};
```

הפעלת אופרטור ההשמה של הבסיס

```
void main()
{
    B b1(5);
    cout << "----------\n";
    B b2;
    cout << "----------\n";
    b2 = b1;
    cout << "----------\n";
}
```

```
In A::A x=5
In B::B(ver 2)
----------
In A::A x=0
In B::B(ver 1)
----------
In B::operator=
In A::operator=
----------
```

שליחה לפונקציה יורש במקום בסיס

- מאחר ואובייקט יורש מכיל את כל הנתונים שיש בבסיס, ניתן לשלוח אותו לפונקציה המצפה לקבל אוביקט מטיפוס הבסיס
- אם הפרמטר הועבר by ref, הפונקציה תתייחס רק לחלק הבסיס של האובייקט
- אם הפרמטר הועבר by val נעבור ב- copy c'tor של הבסיס ביצירת ההעתק לפונקציה
- פונקציה המצפה לקבל יורש, לא יכולה לקבל בסיס במקום (כי יהיו חסרים נתונים)
  - אלא אם קיים בנאי ליורש המקבל פרמטר מטיפוס הבסיס (casting)

שליחה לפונקציה יורש במקום בסיס דוגמה

```
void foo(A a)
{
    cout << "In foo\n";
}

void main()
{
    B b;
    cout << "----------\n";
    foo(b);
    cout << "----------\n";
}
```

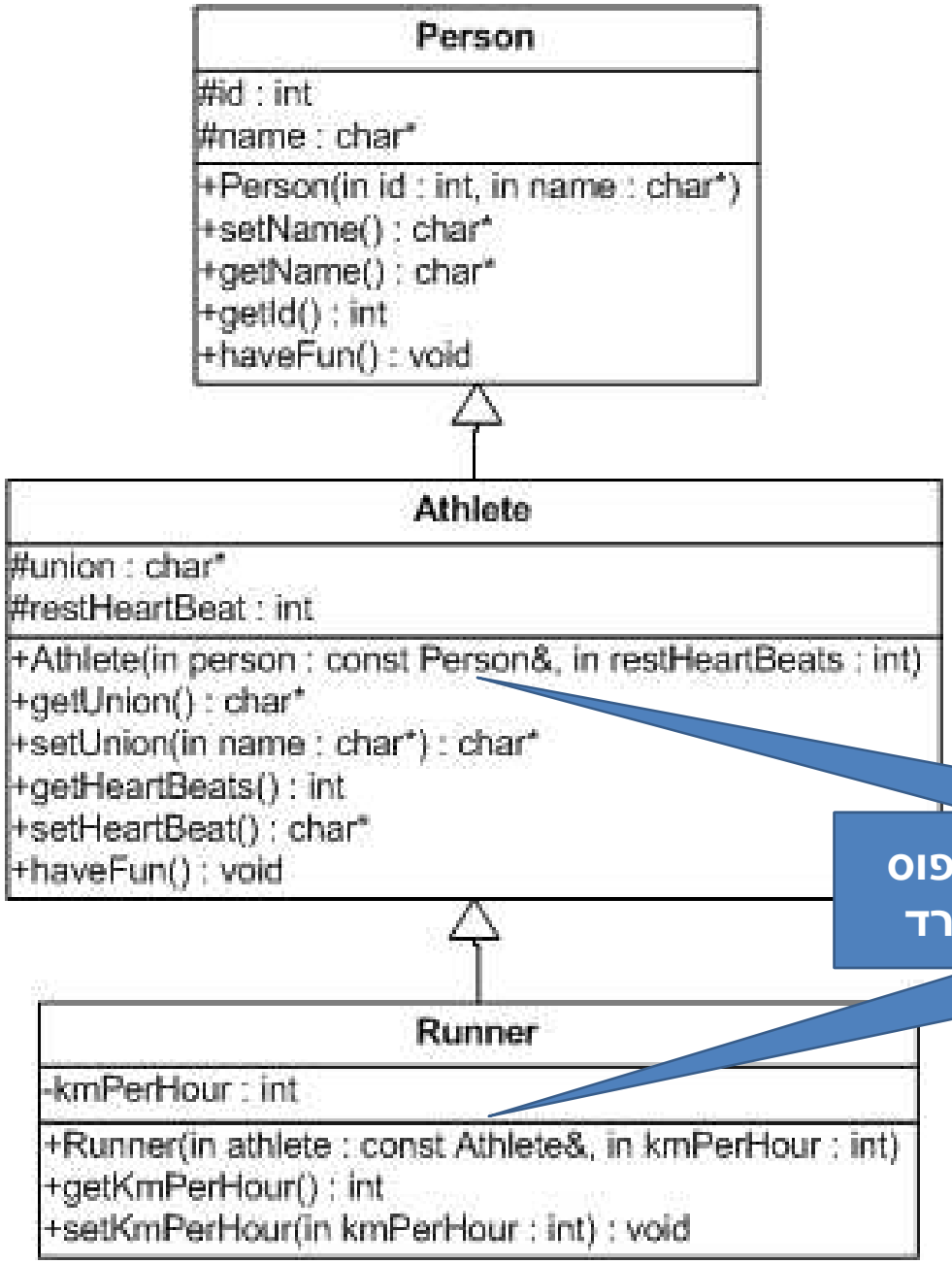
```
In A::A x=0
In B::B(ver 1)
----------
In A::A(copy)
In foo
In A::~A
----------
In B::~B
In A::~A
```

מאחר והפונקציה מצפה לקבל A, עוברים ב- copy של A

הורשה בשרשרת

- אין מניעה לרשת ממחלקה שיורשת ממחלקה אחרת
- במקרה כזה, כל מחלקה יורשת צריכה לאתחל רק את הבסיס הישיר שלה בלבד

הורשה בשרשרת דוגמה



עדיף שכל c'tor יקבל כפרמטר את טיפוס הבסיס כפרמטר, מאשר כל שדה בנפרד

הורשה בשרשרת הקוד (1)

```
class Person
{
protected:
     int id;
     char* name;
public:
     Person(int id, const char* name) : id(id), name(strdup(name))
                                                {cout << "In Person::Person\n";}
     Person(const Person& other) : name(NULL)
     {
          cout << "In Person::Person(copy)\n";
          *this = other;
     }
     ~Person()
     {
          cout << "In Person::~Person\n";
          delete[]name;
     }
     void show() const { cout << "Id: " << id << ", Name: " << name << endl; }
     const Person& operator=(const Person& other)
     {
          cout << "In Person::operator=\n";
          if (this != &other)
          {
               delete[]name;
               name = strdup(other.name);
               id = other.id;
          }
          return *this;
     }
};
```

הורשה בשרשרת הקוד (2)

```
class Athlete : public Person
{
protected:
    int restHeartBeat;
public:
    Athlete(const Person& base, int restHearBeat)
                : Person(base) , restHeartBeat(restHearBeat)
    {
        cout << "In Athlete::Athlete\n";
    }

    Athlete(const Athlete& other)
                : Person(other) , restHeartBeat(other.restHearBeat)
    {
        cout << "In Athlete::Athlete(copy)\n";
    }

    ~Athlete() {cout << "In Athlete::~Athlete\n"; }

    void show() const
    {
        Person::show();
        cout << "\tRest Heartbeat: " << restHeartBeat << endl;
    }
};
```

קריאה ל- copy c'tor של האבא

הפעלת שיטה של האבא

הורשה בשרשרת הקוד (3)

```
class Runner : public Athlete
{
    double kmPerHour;
public:
    Runner(const Athlete& base, double kmPerHour)
                : Athlete(base) , kmPerHour(kmPerHour)
    {
        cout << "In Runner::Runner\n";
    }

    Runner(const Runner& other) : Athlete(other)
    {
        cout << "In Runner::Runner(copy)\n";
    }

    ~Runner() { cout << "In Runner::~Runner\n"; }

    void show() const
    {
        Athlete::show();
        cout << "\tKmPerHour: " << kmPerHour << endl;
    }
};
```

קריאה ל- copy c'tor של האבא

הפעלת שיטה של האבא הישיר

הורשה בשרשרת דוגמת main (1)

```
void main()
{
    Athlete a(Person(111, "gogo"), 55);
    cout << "---------\n";
    a.show();
    cout << "---------\n";
}
```

```
In Person::Person
In Person::Person(copy)
In Person::operator=
In Athlete::Athlete
In Person::~Person
---------
Id: 111, Name: gogo
        Rest Heartbeat: 55
---------
In Athlete::~Athlete
In Person::~Person
```

הורשה בשרשרת דוגמת main (2)

```
void main()
{
    Person p(111, "gogo");
    cout << "---------\n";
    Runner r(Athlete(p, 58), 13);
    cout << "---------\n";
    r.show();
    cout << "---------\n";
}
```

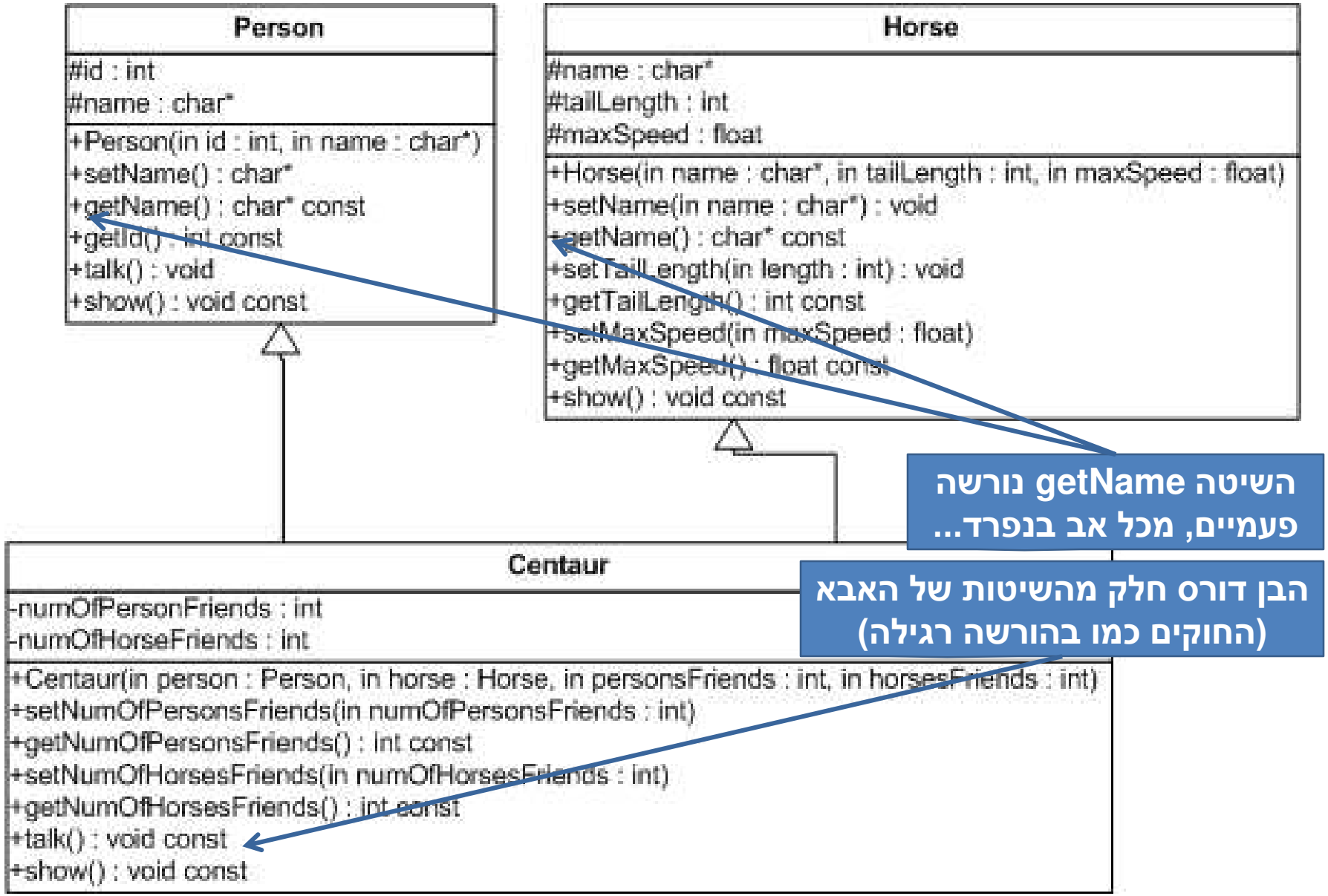
```
In Person::Person
---------
In Person::Person(copy)
In Person::operator=
In Athlete::Athlete
In Person::Person(copy)
In Person::operator=
In Athlete::Athlete(copy)
In Runner::Runner
In Athlete::~Athlete
In Person::~Person
---------
Id: 111, Name: gogo
        Rest Heartbeat: 58
        KmPerHour: 13
---------
In Runner::~Runner
In Athlete::~Athlete
In Person::~Person
In Person::~Person
```

הורשה בשרשרת דוגמת main (3)

```
void main()
{
    Runner r(Athlete(Person(111, "gogo"), 58), 13);
    cout << "---------\n";
    r.show();
    cout << "---------\n";
}
```

```
In Person::Person
In Person::Person(copy)
In Person::operator=
In Athlete::Athlete
In Person::Person(copy)
In Person::operator=
In Athlete::Athlete(copy)
In Runner::Runner
In Athlete::~Athlete
In Person::~Person
In Person::~Person
---------
Id: 111, Name: gogo
        Rest Heartbeat: 58
        KmPerHour: 13
---------
In Runner::~Runner
In Athlete::~Athlete
In Person::~Person
```

הורשה מרובה: קנטאור: תרשים



הורשה מרובה דוגמת centaur הקוד (1)

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

public:
     Person(int id, const char* name);
     Person(const Person& other);
     ~Person();

     const Person& operator=(const Person& other);

     const char* getName() const { return name; }

     void show() const { cout << "I'm a person named " << name << endl; }
     void talk() const { cout << "Person says: bla-bla\n"; }
};
```

```
class Horse
{
protected:
     char* name;
     int tailLength;
     float maxSpeed;
public:
     Horse(const char* name, int tailLength, float maxSpeed);
     Horse(const Horse& other);
     ~Horse();

     const Horse& operator=(const Horse& other);

     const char* getName() const { return name; }

     void show() const { cout << "I'm a horse named " << name << endl; }
};
```

נשים לב שבשתי המחלקות יש את השיטות getName ו- show

הורשה מרובה דוגמת centaur הקוד (2)

```
class Centaur : public Person, public Horse
{
private:
    int numOfPersonsFriends;
    int numOfHorsesFriends;
public:
    Centaur(const Person& p, const Horse& h,
                int numOfPersonsFriends, int numOfHorsesFriends)

        : Person(p), Horse(h),
                numOfPersonsFriends(numOfPersonsFriends),
                numOfHorsesFriends(numOfHorsesFriends) {}

    void show() const
    {
        cout << "I'm a centaur:\n";
        Person::show();
        Horse::show();
    }

    void talk() const { cout << "Centaur says: Haaaaa!\n"; }
};
```

ציון כל המחלקות מהן המחלקה יורשת

אתחול האבות בשורת האתחול

דריסת שיטות שהוגדרו באב

קריאה לשיטה שמומשה באב

הורשה מרובה דוגמת centaur הקוד (2)

```
void main()
{
    Centaur c(Person(111, "gogo"), Horse("Rockey", 70, 39), 3, 3);
    c.show();
    cout << "The cenatur name is " << c.getName() << endl;
}
```

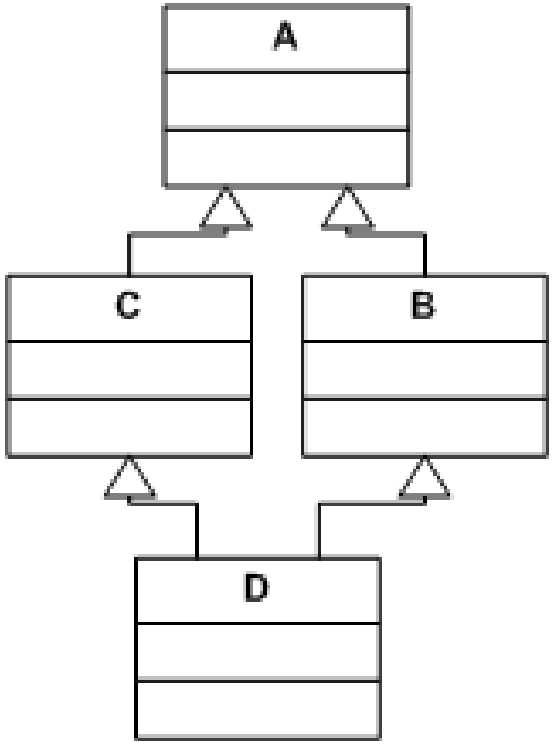
```
error C2385: ambiguous access of 'getName'
could be the 'getName' in base 'Person'
or could be the 'getName' in base 'Horse'
```

	"Centaur::getName" is ambiguous	multiple inheritance - centaur
C2385	ambiguous access of 'getName'	multiple inheritance - centaur

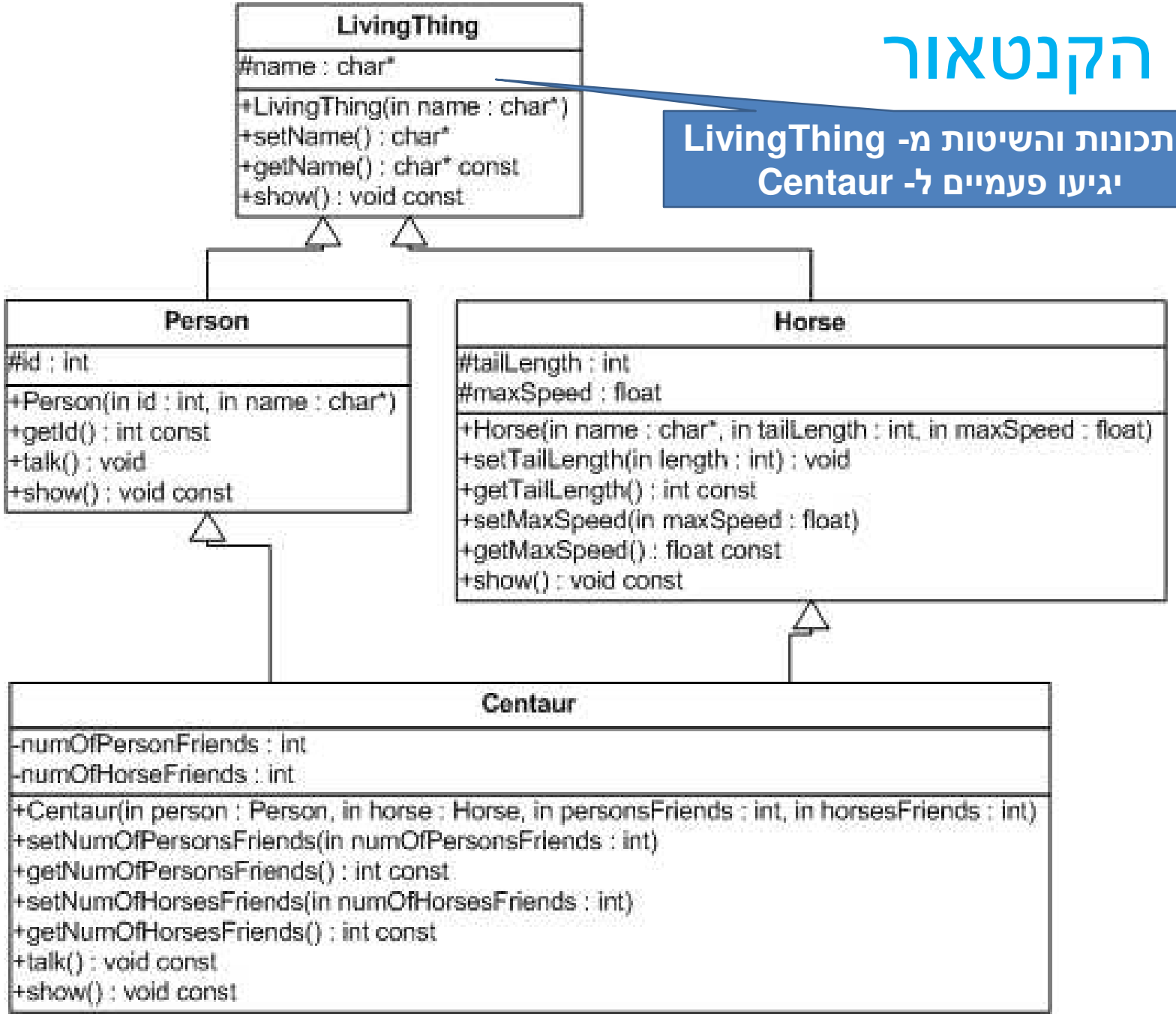
בהורשה המרובה קיבלנו את כל השיטות והתכונות שהיו שאבות. המצב הוא שיש ב- Centaur את התכונה name פעמיים וכן את השיטה getName. הקומפיילר לא יודע לאיזה מימוש לפנות ולכן נותן שגיאת קומפילציה. הדרך לפתור שיטה זו היא לדרוס את השיטה שנורשה מכמה אבות ביורש עצמו.

הורשה מרובה עם אב קדמון משותף ירושת יהלום

- במקרה כזה תהייה כפילות בנכד עבור כל התכונות שיש באב-הקדמון
  - תכונות אלו יגיעו מכל אב בנפרד
- ברוב המקרים, לא נרצה את כפילות התכונות (בעיית ambiguity וכן בזבוז זכרון)



ירושת יהלום דוגמת הקנטאור



התכונות והשיטות מ- LivingThing
יגיעו פעמיים ל- Centaur

הורשה וירטואלית (1)

```
class LivingThing
{
protected:
    char* name;
public:
    LivingThing(const char* name);
    LivingThing(const LivingThing& other);
    ~LivingThing() {}

    const LivingThing& operator=(const LivingThing& other);

    const char* getName() const { return name; }

    void show() const { cout << "My name is " << name << endl; }
};
```

במחלקת הבסיס (האב הקדמון) אין שום שינוי

הורשה וירטואלית (2)

במחלקות היורשות נציין את המילה virtual

```
class Person : virtual public LivingThing
{
protected:
    int id;
public:
    Person(int id, const char* name) : LivingThing(name), id(id) {}
    //Person(const Person& other);
    //~Person() {}
    //const Person& operator=(const Person& other);

    //const char* getName() const { return name; }

    void show() const
    {
        LivingThing::show();
        cout << "My id is " << id << endl;
    }

    void talk() const { cout << "Person says: bla-bla\n"; }
};
```

תוספת זו אינה משפיעה על שימוש במחלקה הנוכחית, אלא רק מצהירה שאם בעתיד ירשו ממחלקה זו וממחלקה נוספת היורשת מהאב-הקדמון, תכונות האב הקדמון יועברו פעם אחת בלבד לנכד

מאחר וכבר אין הקצאה דינאמית במחלקה, יש להוריד את השלישיה

הורשה וירטואלית (3)

```
class Horse : virtual public LivingThing
{
protected:
    int tailLength;
    float maxSpeed;
public:
    Horse(const char* name, int tailLength, float maxSpeed) : LivingThing(name),
                                                    tailLength(tailLength), maxSpeed(maxSpeed) {}
    //Horse(const Horse& other);
    //~Horse();
    //const Horse& operator=(const Horse& other);

    //const char* getName() const { return name; }

    void show() const
    {
        LivingThing::show();
        cout << "My tailen is " << tailLength << " and my max speed is " << maxSpeed << endl;
    }
};
```

הורשה וירטואלית (4)

```
class Centaur : public Person, public Horse
{

private:
    int numOfPersonsFriends;
    int numOfHorsesFriends;
public:
    Centaur(const Person& p, const Horse& h,
                int numOfPersonsFriends, int numOfHorsesFriends)
        : LivingThing(p.getName()) , Person(p), Horse(h),
            numOfPersonsFriends(numOfPersonsFriends),
            numOfHorsesFriends(numOfHorsesFriends) {}

    void show() const
    {
        cout << "I'm a centaur:\n";
        Person::show();
        Horse::show();
    }

    void talk() const { cout << "Centaur says: Haaaaa!\n"; }
};
```

כאשר יורשים ממחלקות שירשו באופן virtual מהסבא, יש לקרוא ל- c'tor של הסבא מהנכד. הסיבה היא ההתלבטות איזה אבא יקרא לאב-הקדמון.
כאשר שניים 2 רבים, שלישי זוכה!
רק הנכד קורא ל- c'tor של הסבא.

עכשיו גם לא תהייה ambiguity מאחר ו- Centaur מגדיר בעצמו את השיטה show

הורשה וירטואלית דוגמת פלט

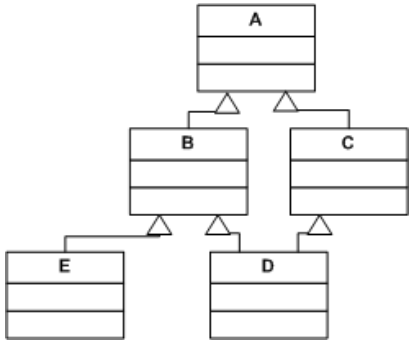
```
class A
{
public:
    A(int x) { cout << "In A::A x=" << x << endl; }
    ~A() { cout << "In A::~A\n"; }
};
class B : virtual public A
{
public:
    B() : A(1) { cout << "In B::B\n"; }
    ~B() { cout << "In B::~B\n"; }
};
class C : virtual public A
{
public:
    C() : A(2) { cout << "In C::C\n"; }
    ~C() { cout << "In C::~C\n"; }
};
class D : public B, public C
{
public:
    D() : A(3) { cout << "In D::D\n"; }
    ~D() { cout << "In D::~D\n"; }
};

class E : public B
{
public:
    E() : A(4) { cout << "In E::E\n"; }
    ~E() { cout << "In E::~E\n"; }
};
```

כאשר יורשים ממחלקה שירשה וירטואלית ממחלקה כלשהי, חובה לקרוא ל- c'tor של האב הקדמון

```
In A::A x=1
In B::B
-------
In A::A x=2
In C::C
-------
In A::A x=3
In B::B
In C::C
In D::D
-------
In A::A x=4
In B::B
In E::E
-------
In E::~E
In B::~B
In A::~A
In D::~D
In C::~C
In B::~B
In A::~A
In C::~C
In A::~A
In B::~B
In A::~A
```

```
void main()
{
    B b;
    cout << "---\n";
    C c;
    cout << "---\n";
    D d;
    cout << "---\n";
    E e;
    cout << "---\n";
}
```



הורשה סדר המעבר בבנאים ב- init line (1)

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

class Father
{
public:
    Father() { cout << "In Father::Father\n"; }
};

class Son : public Father
{
    Inner i;
public:
    Son() : i(), Father() { cout << "In Son::Son\n"; }
};

void main()
{
    Son s;
}
```

```
In Father::Father
In Inner::Inner
In Son::Son
```

כאשר בשורת האתחול מאתחלים גם את האב וגם אובייקט מוכל, ראשית יאותחל האב ורק אח"כ האובייקט המוכל, ללא קשר לסדר כתיבתם ב- init line

הורשה סדר המעבר בבנאים ב- init line (2)

```
class Father1
{
public:
    Father1() { cout << "In Father1::Father1\n"; }
};

class Father2
{
public:
    Father2() { cout << "In Father2::Father2\n"; }
};

class Son : public Father1, public Father2
{
public:
    Son() : Father2(), Father1() { cout << "In Son::Son\n"; }
};

void main()
{
    Son s;
}
```

```
In Father1::Father1
In Father2::Father2
In Son::Son
```

במקרה של הורשה מרובה, סדר הקריאות לאתחול האבות יהיה כסדר הירושה בהגדרת המחלקה, ללא קשר לסדר כתיבתם ב- init line

הורשה סדר המעבר בבנאים ב- init line (3)

```
class Grandfather
{
public:
    Grandfather() { cout << "In Grandfather::Grandfather\n"; }
};

class Father1 : virtual public Grandfather
{
public:
    Father1() { cout << "In Father1::Father1\n"; }
};

class Father2 : virtual public Grandfather
{
public:
    Father2() { cout << "In Father2::Father2\n"; }
};

class Son : public Father1, public Father2
{
public:
    Son() : Father2(), Grandfather(), Father1() { cout << "In Son::Son\n"; }
};

void main()
{
    Son s;
}
```

```
In Grandfather::Grandfather
In Father1::Father1
In Father2::Father2
In Son::Son
```

במידה ויש הורשה מרובה וירטואלית, סדר הקריאות יהיה ראשית לסבא, ורק אח"כ לאבות עפ"י סדר הירושה בהגדרת המחלקה, ללא קשר לסדר כתיבתם ב- init line

מוטיבציה להורדת רמת ההרשאה

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

public:
    /* c'tors and stuff… */

    void haveFun() const { cout << "Yeah! Going to the sea!\n"; }
};
```

protected

```
class Student : public Person
{
private:
    float average;
public:
    /* c'tors and stuff… */

    void haveFun() const;
};
```

```
void main()
{
    Student s(111, "momo", 98.5);

    s.haveFun();
    s.Person::haveFun();
}
```

לא נרצה לאפשר שימוש של מתודות שהוגדרו באב (נרצה להסתירן)

הורדת רמת הרשאת הירושה מוטיבציה

- בהינתן המחלקה Stack שיורשת מ- Array, לא נרצה לחשוף את כל המתודות שקיימות ב- Array עבור משתנה מטיפוס Stack

```
class Array
{
    int* values;
    int maxSize;
    int currentSize;
public:
    void insertFirst(int newVal);
    void insertLast(int newVal);
    void insertAfterValue(int insertAfter, int newVal);
    void removeAllInstancesOf(int val);
    int  removeFirst();
    int  removeLast();
};

class Stack : protected Array
{
public:
    void push(int val) { insertLast(val); }
    int pop()          { return removeLast(); }
};
```

```
void main()
{
    Stack s;

    s.insertLast(5);
    s.push(5);
}
```

הרשאת ההורשה

- תחביר: `class B : public A`

- באופן זה משמע שכל התכונות והשיטות שב- A מתקבלות ב- B בהתאם להרשאתם המקורית

- ניתן להגדיר שהרשאת ההורשה היא private או protected
  - כאשר הרשאת ההורשה היא private משמע שכל מה שבבסיס יהפוך להיות private ברמה של היורש
  - כאשר הרשאת ההורשה היא protected משמע שכל מה שבבסיס יהפוך להיות protected ברמה של היורש (פרט למה שהיה private בבסיס וישאר private)

- כלומר, הורשה עם הרשאת השונה מ- public מורידה את רמת הנגישות של התכונות והשיטות ברמת המחלקה היורשת

הרשאות בהורשה דוגמאות

```
class A
{
public:
    void foo() const {}
};

class B : public A
{
public:
    void print() const
    {
        foo();
    }
};

class C : protected A
{
public:
    void print() const
    {
        foo();
    }
};
```

```
class D : private A
{
public:
    void print() const
    {
        foo();
    }
};

class E : public C
{
public:
    void print() const
    {
        foo();  ✓
    }
};

class F : public D
{
public:
    void print() const
    {
        foo();  ✗
    }
};
```

```
void main()
{
    B b;
    b.print();  ✓
    b.foo();    ✓

    C c;
    c.print();  ✓
    c.foo();    ✗

    D d;
    d.print();  ✓
    d.foo();    ✗
}
```

ביחידה זו למדנו:

- מוטיבציה להורשה
- ההרשאה protected
- מעבר בבנאים בהורשה
- מעבר ב- d’tor בהורשה
- ה- copy c’tor שניתן במתנה
- אופרטור ההשמה שניתן במתנה
- שליחת יורש לפונקציה המצפה לקבל בסיס
- הורשה בשרשרת
- הורשה מרובה
  - הורשה מרובה עם אב-קדמון משותף
- סדר האתחול
- הורשת protected ו- private