תכנות מכוון עצמים ו- C++
יחידה 08
פולימורפיזם

קרן כליף

ביחידה זו נלמד:

- שימוש במצביע לאב ויצירת אובייקט כבן
- מוטיבציה לפולימורפיזם
- קישור סטטי לעומת קישור דינאמי
- הטבלה הוירטואלית
- מערך שאיבריו מטיפוסים שונים בעלי בסיס משותף
- d’tor וירטואלי
- זיהוי טיפוס בזמן ריצה
- dynamic cast
- מימוש נכון של << בפולימורפיזם
- מתודות override
- שיטות ומחלקות אבסטרקטיות
- השיטה clone
- מתודות final

מצביע לאב שבפועל הוא בן

- ראינו בפרק של ההורשה שכאשר יש פונקציה המצפה לקבל משתנה מטיפוס האב, אפשר בפועל לשלוח אליה בן
- ניתן גם כאשר מגדירים מצביע לאב, לייצר את האובייקט בפועל כבן

```
void main()
{
    Person* p1 = new Person(111, "gogo");
    Person* p2 = new Student(Person(222, "momo"), 87.3);

    cout << "p1 is having fun: ";
    p1->haveFun();

    cout << "p2 is having fun: ";
    p2->haveFun();

    delete p1;
    delete p2;
}
```

```
p1 is having fun: Yeah! Going to the sea!
p2 is having fun: Yeah! Going to the sea!
```

למרות שהמימוש של haveFun שונה בין האב לבן, עדיין אנו רואים שהופעלה השיטה שהוגדרה באב..

מצביע לאב שבפועל הוא בן (2)

```
void main()
{
    Person* p1 = new Person(111, "gogo");
    Person* p2 = new Student(Person(222, "momo"), 87.3);

    cout << "p1 is having fun: ";
    p1->haveFun();

    cout << "p2 is having fun: ";
    ((Student*)p2)->haveFun();

    delete p1;
    delete p2;
}
```

כדי שתופעל השיטה שמומשה בבן ניתן לעשות casting
לבן (אנו יודעים ש- p2 הוא מטיפוס Student)

```
p1 is having fun: Yeah! Going to the sea!
p2 is having fun: Yeah! Doing homework!
```

שיטות שניתן להפעיל על המצביע

- על מצביע ניתן להפעיל רק שיטות שהמצביע מכיר בזמן קומפילציה
  - מאחר ובזמן קומפילציה הקומפיילר מכיר רק את טיפוס ההצבעה (ולא את טיפוס האובייקט בפועל) ניתן להפעיל רק שיטות של המצביע (האב)

```
void main()
{
    Person* p = new Student(Person(222, "momo"), 87.3);

    p->registerToCourse();
    ((Student*)p)->registerToCourse();

    delete p;
}
```

אם p אינו מטיפוס Student
התוכנית עלולה לעוף בזמן ריצה.
נראה פתרון אלגנטי בהמשך...

דוגמה מדוע הקומפיילר אינו מכיר את הטיפוס בפועל בזמן קומפילציה

```
void main()
{
    Person* p = nullptr;
    int type;

    cout << "Enter 1 for person, 2 for student: ";
    cin >> type;

    switch (type)
    {
    case 1:
        p = new Person(111, "gogo");
        break;
    case 2:
        p = new Student(111, "gogo", 87.3);
        break;
    }

    if (p)
        p->haveFun();

    delete p;
}
```

```
Enter 1 for person, 2 for student: 2
Yeah! Going to the sea!
```

```
Enter 1 for person, 2 for student: 2
Yeah! Doing homework!
```

הקומפיילר לא יכול לנחש מה יהיה הטיפוס בפועל,
שיוקלד ע"י המשתמש בזמן ריצה.
בכל זאת היינו רוצים שתופעל השיטה עם המימוש
המתאים, בלי סירבול של ה- main בשימוש ב- casting

פולימורפיזם (רב-תצורתיות) מוטיבציה

1. הפעלת שיטה לפי טיפוס האובייקט בפועל, ולא לפי טיפוס המצביע
   - ראינו שאם יש מצביע לטיפוס כלשהו והאובייקט בפועל הוא מטיפוס יורש, תקרא השיטה הממומשת במחלקת הבסיס
   - כדי לפנות לשיטה מטיפוס האובייקט בפועל, היינו צריכים לבצע casting
   - מנגנון הפולימורפיזם מאפשר לפנות לשיטה לפי האובייקט שנוצר בפועל, ולא לפי ההפניה, ללא שימוש ב- casting!

2. אוספים של אובייקטים שונים
   - ע"י יצירת מערך של מצביעים לבסיס מסויים, נוכל לייצר מערך שכל אחד מאיבריו מטיפוס שונה בפועל
     - דוגמא: מחלקת בסיס "חיה", שממנה יורשים "דג", "סוס" ו"חתול"
   - מאפשר רב-תצורתיות למצביע, כל פעם להיות אובייקט שונה

קישור סטטי

- "קישור סטטי": כאשר הקומפיילר פונה לשיטה שמומשה בטיפוס המצביע, ולא לפי הטיפוס שנוצר בפועל
  - כלומר, כבר בזמן קומפילציה הקומפיילר רוצה לדעת באיזו מחלקה נמצא המימוש לשיטה המבוקשת
- מאחר ובזמן קומפילציה לא ניתן לדעת מה יהיה טיפוס האובייקט בפועל, בסיס או יורש, הקומפיילר מתייחס לשיטות של המצביע, שטיפוסו תמיד ידוע בזמן קומפילציה
- הקישור הסטטי הוא מה שקורה במערכת כברירת-מחדל, ואין צורך להוסיף לתחביר דבר

קישור דינאמי (קישור מאוחר, late binding)

- בקישור דינאמי הקומפיילר דוחה את ההחלטה לאיזה מימוש של השיטה לפנות לזמן ריצה, בו כבר ידוע מה טיפוס האובייקט בפועל
- כאשר הקומפיילר נתקל בשיטה של אובייקט מטיפוס המצביע, הוא בודק האם עליו לחפש מימוש בעל עדיפות גבוהה יותר במחלקה שממנה נוצר האובייקט בפועל
- כדי לתמוך בקישור דינאמי צריך לבצע תוספת לתחביר

קישור דינאמי תחביר

```
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; }

    virtual void haveFun() const { cout << "Yeah! Going to the sea!\n"; }
};
class Student : public Person
{
private:
    float average;
public:
    Student(int id, const char* name, float average) {…}
    Student(const Person& base, float average) {…}

    void registerToCourse(const char* courseName) const {…}

    virtual void haveFun() const {cout << "Yeah! Doing homework!\n";}
};
```

השימוש במילה virtual בפולימורפיזם שונה מהשימוש בה בהורשה, ואין כל קשר בינהם!

ציון שיש לדחות את הקישור למימוש השיטה לזמן ריצה. חובה לציין באב.

במקרה של הפרדת המימוש מההצהרה, את המילה virtual מציינים בהצהרה בלבד

נהוג לציין גם ביורש, אך אין חובה לכך

מערך של איברים בעלי בסיס משותף

```
void main()
{
    Person* arr[3];
    int type;

    for (int i = 0; i < 3; i++)
    {
        cout << "Enter 1 for person, 2 for student: ";
        cin >> type;

        switch (type)
        {
        case 1:  arr[i] = new Person(111, "gogo");          break;
        case 2:  arr[i] = new Student(111, "gogo", 87.3); break;
        default: cout << "Invalid option!\n";               break;
        }
    }

    for (int i =0 ; i < 3 ; i++)
    {
        if (arr[i])
            arr[i]->haveFun();
    }

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

עבור כל אובייקט מופעלת השיטה המתאימה עבורו
(בהנחה שהשיטה מוגדרת כ- virtual באבא)

```
Enter 1 for person, 2 for student: 1
Enter 1 for person, 2 for student: 2
Enter 1 for person, 2 for student: 1
Yeah! Going to the sea!
Yeah! Doing homework!
Yeah! Going to the sea!
```

דוגמה 1

```
class A
{
public:
            void foo() const { cout << "In A::foo\n"; }
            void goo() const { cout << "In A::goo\n"; }
    virtual void moo() const { cout << "In A::moo\n"; }
    virtual void koo() const { cout << "In A::koo\n"; }
};


class B : public A
{
public:
    virtual void foo() const { cout << "In B::foo\n"; }
            void moo() const { cout << "In B::moo\n"; }
    virtual void koo() const { cout << "In B::koo\n"; }
};


class C : public B
{
public:
    virtual void foo() const { cout << "In C::foo\n"; }
            void moo() const { cout << "In C::moo\n"; }
};
```

```
void main()
{
    A* obj = new B();

    obj->foo();
    obj->goo();
    obj->moo();
    obj->koo();

    delete obj;
}
```

קישור סטטי

קישור דינאמי, בגלל שצוין virtual לפני שם השיטה בטיפוס המצביע (באב)

```
In A::foo
In A::goo
In B::moo
In B::koo
```

דוגמה 2

```
class A
{
public:
            void foo() const { cout << "In A::foo\n"; }
            void goo() const { cout << "In A::goo\n"; }
    virtual void moo() const { cout << "In A::moo\n"; }
    virtual void koo() const { cout << "In A::koo\n"; }
};


class B : public A
{
public:
    virtual void foo() const { cout << "In B::foo\n"; }
            void moo() const { cout << "In B::moo\n"; }
    virtual void koo() const { cout << "In B::koo\n"; }
};


class C : public B
{
public:
    virtual void foo() const { cout << "In C::foo\n"; }
            void moo() const { cout << "In C::moo\n"; }
};
```

```
void main()
{
    A* obj = new C();

    obj->foo();
    obj->goo();
    obj->moo();
    obj->koo();

    delete obj;
}
```

קישור סטטי

קישור דינאמי, עבור koo משתמש
במימוש שב- B משום שהוא יותר ספציפי

```
In A::foo
In A::goo
In C::moo
In B::koo
```

דוגמה 3

```
class A
{
public:
            void foo() const { cout << "In A::foo\n"; }
            void goo() const { cout << "In A::goo\n"; }
    virtual void moo() const { cout << "In A::moo\n"; }
    virtual void koo() const { cout << "In A::koo\n"; }
};


class B : public A
{
public:
    virtual void foo() const { cout << "In B::foo\n"; }
            void moo() const { cout << "In B::moo\n"; }
    virtual void koo() const { cout << "In B::koo\n"; }
};


class C : public B
{
public:
    virtual void foo() const { cout << "In C::foo\n"; }
            void moo() const { cout << "In C::moo\n"; }
};
```

הפעם קישור דינאמי, למרות שב- A לא צויין virtual

המימוש שהתקבל מ- A בירושה

קישור דינאמי מאחר ובמימוש ב- A צויין virtual, אז הוא כאילו רשום בכל היורשים, אפילו אם לא צויין במפורש

```
void main()
{
    B* obj = new C();

    obj->foo();
    obj->goo();
    obj->moo();
    obj->koo();

    delete obj;
}
```

```
In C::foo
In A::goo
In C::moo
In B::koo
```

d’tor וירטואלי

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

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

void main()
{
    Father* f = new Son();
    delete f;
}
```

מבחינת הקומפיילר f הוא מטיפוס Father ובעת הפקודה delete זו הפעלת שיטת ה- d’tor של Father

```
In Father::~Father
```

- במקרה זה לא יופעל ה- d’tor של הבן, ובמידה והיו בבן הקצאות דינאמיות, הן לא היו משוחררות
- לכן ה- d’tor בבסיס צריך להיות וירטואלי!

d'tor וירטואלי (2)

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

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

void main()
{
    Father* f = new Son();
    delete f;
}
```

```
In Son::~Son
In Father::~Father
```

תמיד כאשר משתמשים במנגנון של הפולימורפזים נגדיר את ה- d'tor כ- virtual, אפילו אם אין בו צורך!

- d'tor הוא שיטה כמו כל שיטה
- ברגע שהגדרנו אותו בבסיס כ- virtual הקומפיילר מחפש את המימוש של האוביקט בפועל, ולכן מפעיל את ה- d'tor של הבן
  - מתוקף חוקי ההורשה, עם סיום ה- d'tor של הבן מופעל ה- d'tor של האב, כדרוש

הטבלה הוירטואלית (vtable)

```
class Base
{
public:
    virtual void foo() const {}
    virtual void goo() const {}
};

class Derived1 : public Base
{
public:
    virtual void foo() const {}
};

class Derived2 : public Base
{
public:
    virtual void goo() const {}
};

int main()
{
    Base* b1 = new Derived1();
    Base* b2 = new Derived2();

    delete b1;
    delete b2;
}
```

מכיל גם את השדה __vptr*

- כאשר יש ולו שיטה וירטואלית אחת במחלקה, לאובייקט יתווסף פוינטר המצביע לטבלה הוירטואלית, שבעזרתה ניתן לדעת איזה מימוש של שיטה בפועל להפעיל

ה- __vptr* נמצא בבסיס, והצבעתו תשתנה ל- vtable המתאים בהתאם לטיפוס האובייקט

b1	0x00a66310 {...}	Base * {Derived1}
[Derived1]	{...}	Derived1
__vfptr	0x009c7b44 {08- virtual table.exe!const Derived1::`vftabl	void * *
[0]	0x009c13d9 {08- virtual table.exe!Derived1::foo(void)}	void *
[1]	0x009c11d1 {08- virtual table.exe!Base::goo(void)}	void *
b2	0x00a66340 {...}	Base * {Derived2}
[Derived2]	{...}	Derived2
__vfptr	0x009c7b54 {08- virtual table.exe!const Derived2::`vftabl	void * *
[0]	0x009c11ef {08- virtual table.exe!Base::foo(void)}	void *
[1]	0x009c119a {08- virtual table.exe!Derived2::goo(void)}	void *

הטבלה הוירטואלית גודל האובייקט

- גודלו של אובייקט המכיל טבלה וירטואלית (כלומר שיש לו לפחות שיטה אחת וירטואלית) מכיל בנוסף לסכום גודל שדותיו, 4 בתים עבור הפוינטר לטבלה הוירטואלית

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

class B
{
    int x;
public:
    virtual void foo() const {}
};

void main()
{
    cout << sizeof(class A) << endl;
    cout << sizeof(class B) << endl;
}
```

```
4
8
```

הפעלת שיטה וירטואלית מה- c'tor או מה- d'tor

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

    virtual ~A()
    {
        cout << "In A::~A\n";
        foo();
        goo();
    }

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

```
In A::A
In A::foo
In A::goo
----------
In A::foo
----------
In A::~A
In A::foo
In A::goo
==========
In A::A
In A::foo
In A::goo
In B::B
In B::foo
In A::goo
----------
In B::foo
----------
In B::~B
In B::foo
In A::goo
In A::~A
In A::foo
In A::goo
```

```
class B : public A
{
public:
    B() {
        cout << "In B::B\n";
        foo();
        goo();
    }
    virtual ~B() {
        cout << "In B::~B\n";
        foo();
        goo();
    }
    virtual void foo() const {
        cout << "In B::foo\n";
    }
};
void main() {
    A* f1 = new A();
    cout << "----------\n";
    f1->foo();
    cout << "----------\n";
    delete f1;
    cout << "==========\n";

    A* f2 = new B();
    cout << "----------\n";
    f2->foo();
    cout << "----------\n";
    delete f2;
}
```

הפעלת שיטה וירטואלית מה- c’tor או מה- d’tor חוקים

- ראינו שכאשר מפעילים שיטה וירטאלית מה- c’tor או מה- d’tor של האב, מופעלת תמיד השיטה שמומשה באב!
  - ביצירת אובייקט, נכנסים ל- c’tor של האב לפני כניסה לגוף ה- c’tor של הבן, ולכן באב עדיין לא יודעים מהו טיפוס הבן
  - בהריסת אובייקט, כאשר נכנסים ל- d’tor של האב הבן כבר מת, ולכן לא יודעים מהו טיפוס הבן

זיהוי טיפוס בזמן ריצה מוטיבציה

```
void main()
{
    Person* p = nullptr;
    int type;

    cout << "Enter 1 for person, 2 for student: ";
    cin >> type;

    switch (type)
    {
        case 1:
            p = new Person(111, "gogo");        break;
        case 2:
            p = new Student(222, "momo", 87.3); break;
        default:
            cout << "Invalid option!\n";        break;
    }

    if (p)
    {
        p->haveFun();
        p->
    }

    delete p
}
```

- getName
- haveFun
- operator=
- ~Person

כאשר כותבים את הקוד, נרצה אפשרות לברר מהו טיפוס האובייקט בזמן ריצה, כדי שנוכל לקרוא לשיטות שספציפיות לאובייקט

מוצגות רק השיטות של Person, כי זה כל מה שהקומפיילר יודע על p

זיהוי טיפוס בזמן ריצה פתרון 1

הפתרון: הוספת שיטה וירטואלית המחזירה את שם המחלקה

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

public:
    …

    virtual const char* getType() const { return "Person"; }
};

class Student : public Person
{
private:
     float average;
public:
    …

    virtual const char* getType() const { return "Student"; }
};
```

זיהוי טיפוס בזמן ריצה שימוש בפתרון 1

```
void main()
{
    Person* p = nullptr;
    int type;

    cout << "Enter 1 for person, 2 for student: ";
    cin >> type;

    switch (type)
    {
        case 1:  p = new Person(111, "gogo");        break;
        case 2:  p = new Student(222, "momo", 87.3); break;
        default: cout << "Invalid option!\n";        break;
    }

    if (p)
    {
        p->haveFun();
        if (strcmp(p->getType(), "Student") == 0)
            ((Student*)p)->registerToCourse("C++");
    }

    delete p;
}
```

ה- casting תמיד יעבוד כי וידאנו
כי המצביע אכן מטיפוס Student

זיהוי טיפוס בזמן ריצה פתרון 2

- מאחר ובירור סוג האובייקט היא פעולה נפוצה בזמן ריצה, יש בספריה typeinfo.h את הפונקציה typeid המקבלת כפרמטר שם של משתנה או טיפוס, ומחזירה משתנה מטיפוס **type_info**
- לטיפוס type_info יש שיטה name המחזירה את שם הטיפוס של הפרמטר שהועבר ל- typeid

```
void main()
{
    Person* p1 = new Person(111, "gogo");
    Person* p2 = new Student(Person(222, "momo"), 98.7f);

    cout << typeid(*p1).name() << endl;
    cout << typeid(*p2).name() << endl;
    cout << typeid(p1).name() << endl;
    cout << typeid(p2).name() << endl;
}
```

```
class Person
class Student
class Person *
class Person *
```

זיהוי טיפוס בזמן ריצה שימוש בפתרון 2

```
void main()
{
    Person* p = nullptr;
    int type;

    cout << "Enter 1 for person, 2 for student: ";
    cin >> type;

    switch (type)
    {
    case 1:  p = new Person(111, "gogo");        break;
    case 2:  p = new Student(222, "momo", 87.3); break;
    default: cout << "Invalid option!\n";        break;
    }

    if (p)
    {
        p->haveFun();
        if (strcmp(typeid(*p).name(), typeid(Student).name()) == 0)
            ((Student*)p)->registerToCourse("C++");
    }

    delete p;
}
```

מקבלת את המשתנה

מקבלת את הטיפוס

שימוש ב- typeid מחליף את הצורך בהגדרת שיטה המחזירה את שם טיפוס האובייקט

זיהוי טיפוס בזמן ריצה
האופטור == ל- type_info

```
void main()
{
    Person* p = nullptr;
    int type;

    cout << "Enter 1 for person, 2 for student: ";
    cin >> type;

    switch (type)
    {
    case 1:  p = new Person(111, "gogo");        break;
    case 2:  p = new Student(222, "momo", 87.3); break;
    default: cout << "Invalid option!\n";        break;
    }

    if (p)
    {
        p->haveFun();
        if (typeid(*p) == typeid(Student))
            ((Student*)p)->registerToCourse("C++");
    }

    delete p;
}
```

ניתן לבצע בדיקת שוויון בין 2 אובייקטים מטיפוס typeinfo ← האופרטור== מועמס במחלקה

אם p* הוא יורש של Student, אז התשובה לבדיקת השיוויון תחזיר false

זיהוי טיפוס בזמן ריצה שימוש ב- typeid

- כדי שהפונקציה typeid תחזיר את הטיפוס האמיתי, עבור המחלקה צריכה להיות קיימת הטבלה הוירטואלית (כלומר, במחלקה צריכה להיות לפחות פונקציה וירטואלית אחת)

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

class B : public A
{

};

void main()
{
    A* a1 = new A();
    A* a2 = new B();

    cout << typeid(*a1).name() << endl;
    cout << typeid(*a2).name() << endl;
}
```

בדוגמא זו אין אף פונקציה וירטואלית בבסיס ← אין טבלה וירטואלית ← typeid מתבצע בזמן קומפילציה ולכן מחזיר את טיפוס המצביע

כאשר עומדים להשתמש במנגנון של הפולימורפיזם תמיד נגדיר את ה- d'tor כ- virtual, אפילו אם אין בו צורך (כדי לעבור ב- d'tor של הבן), ולכן אנו לא אמורים להתקל בבעיה זו

```
class A
class A
```

זיהוי טיפוס בזמן ריצה מוטיבציה לפתרון 3

- בהינתן מצביע לאב, כדי לפנות לשיטה שקיימת רק במחלקה היורשת, וידאנו שהוא מטיפוס המחלקה היורשת, ורק אז בצענו casting למשתנה

```
void main()
{
    Person* p = nullptr;
    int type;

    cout << "Enter 1 for person, 2 for student: ";
    cin >> type;

    switch (type)
    {
    case 1:  p = new Person(111, "gogo");        break;
    case 2:  p = new Student(222, "momo", 87.3); break;
    default: cout << "Invalid option!\n";        break;
    }

    if (p)
    {
        p->haveFun();
        if (typeid(*p) == typeid(Student))
            ((Student*)p)->registerToCourse("C++");
    }
    delete p;
}
```

dynamic_cast

- במקרה בו לא ידוע האם האובייקט באמת מטיפוס המחלקה היורשת, נוכל בביטחה להשתמש ב- casting כפי שראינו בשקף הקודם, רק לאחר בירור הטיפוס האובייקט בפועל
- מאחר ו- casting לצורך קבלת טיפוס האובייקט בפועל היא פעולה שכיחה, ישנו פתרון מובנה בשפה והוא dynamic_cast:

```
Person* p = new Student();
Student* temp = dynamic_cast<Student*>(p);
```

הטיפוס אליו נמיר את p, יהיה לרוב מחלקה יורשת

temp אינו אובייקט נוסף, אלא מצביע לאובייקט המקורי, רק עם משקפיים המאפשרות לבצע פעולות שיש ב- Student

במידה ו- p אינו מטיפוס Student או יורשיו, ה- dynamic_cast יחזיר nullptr

dynamic_cast דוגמה לערך המוחזר

```
void main()
{
    Person* p1 = new Person(111, "gogo");
    Person* p2 = new Student(Person(222, "momo"), 87.3f);

    Student* temp1 = dynamic_cast<Student*>(p1);
    Student* temp2 = dynamic_cast<Student*>(p2);
}
```

p1 אינו Student לכן temp1 הוא NULL

Locals

Name	Value	Type
p1	0x0087eab8 {id=111 name=0x0087e770 "gogo" }	Person *
p2	0x00879fb0 {average=87.3000031 }	Person * {Student}
temp1	0x00000000 <NULL>	Student *
temp2	0x00879fb0 {average=87.3000031 }	Student *

p2 הוא Student לכן ל- temp2 ול- p2 אותה כתובת. רק דרך temp2 הקומפיילר יכול לפנות לשיטות שהוגדרו ב- Student

שימוש dynamic_cast

```
void main()
{
    Person* p = nullptr;
    int type;

    cout << "Enter 1 for person, 2 for student: ";
    cin >> type;

    switch (type)
    {
    case 1:  p = new Person(111, "gogo");          break;
    case 2:  p = new Student(222, "momo", 87.3); break;
    default: cout << "Invalid option!\n";          break;
    }

    if (p)
    {
        p->haveFun();
        Student* temp = dynamic_cast<Student*>(p);
        if (temp)
            temp->
    }

    delete p;
}
```

- getName
- getType
- haveFun
- operator=
- registerToCourse
- ~Person

במידה ו- p אינו מטיפוס Student או יורשיו, ההמרה תחזיר NULL

כעת ניתן להפעיל שיטה של היורש על המצביע

סיכום 3 דרכים לזיהוי סוג האובייקט בזמן ריצה

1. כתיבת מתודה המחזירה את שם המחלקה וביצוע השוואת מחרוזות
2. השוואת הערך המוחזר מ- typeid באמצעות strcmp או אופרטור ==
3. שימוש ב- dynamic_cast

שליחת בן במקום אב

איך היה משתנה הפלט אם ל- A לא היה virtual d'tor?

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

class B : public A
{
public:
    B() { cout << "In B::B\n"; }
    B(const B& other) { cout << "In B::B(copy)\n"; }
    ~B() { cout << "In B::~B\n"; }
};

void foo(A a) { cout << "In foo, a is |" << typeid(a).name() << "|\n"; }
void goo(A& a) { cout << "In goo, a is |" << typeid(a).name() << "|\n"; }
void moo(A* a) { cout << "In moo, a is |" << typeid(*a).name() << "|\n"; }
```

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

```
In A::A
In B::B
--------
In A::A(copy)
In foo, a is |class A|
In A::~A
--------
In goo, a is |class B|
--------
In moo, a is |class B|
--------
In B::~B
In A::~A
```

const_cast

- כידוע היטב לכולם, על אובייקטים שהם const ניתן להפעיל אך ורק שיטות שהן const!
- לעיתים במערכות גדולות **שתכנונן לא היה מושלם**, יתכן ויגיע לידנו אובייקט שהוא const אבל כן נרצה להפעיל עליו שיטה שאינה const
- const_cast מאפשר לנו להסתכל על האובייקט במשקפיים שאינן const
- **שימו לב:** אם יוצא לכם להשתמש בכלי זה כנראה משהו בתכנון המערכת אינו מדויק!

const_cast דוגמה

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

void main()
{
    const A a;
    a.foo();

    A* temp = const_cast<A*>(&a);
    temp->foo();
}
```

האובייקט a הוא const, לכן לא ניתן להפעיל עליו את השיטה foo שאינה const

'void A::foo(void)': cannot convert 'this' pointer from 'const A' to 'A &'

נסתכל על a במשקפיים שאינן const

Watch 1

Name	Value	Type
&a	0x013bfd8f {...}	A *
temp	0x013bfd8f {...}	A *

ניתן לראות ש- temp אינו אובייקט חדש, אלא מצביע לאותה כתובת של a, רק יכול להפעיל שיטות שאינן const

מימוש אופרטור == בפולימורפיזם האב

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

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

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

    virtual bool operator==(const Person& other) const
    {
        return id == other.id && strcmp(name, other.name) == 0;
    }
    ...
};
```

במימוש באב אין שום הבדל...

מימוש אופרטור == בפולימורפיזם הבן

```
class Student : public Person
{
private:
    float average;
public:
    Student(const Person& base, float average)
                : Person(base), average(average) {}

    virtual bool operator==(const Student& other) const
    {
        if (!((Person&)(*this) == other))
        if (!Person::operator==(other))
            return false;

        return average == other.average;
    }

    …
};
```

ניתן לראות שלמעשה הופעל אופרטור == של האב, ולא של הבן...

תזכורת: כך נראתה החתימה של הפעולה במחלקה Person:

```
virtual bool operator==(const Person& other) const {...}
```

או

הסיבה היא שיש לנו פה העמסת פונקציות ולא דריסה, מאחר והמימוש באב מקבל Person והמימוש בבן מקבל Student...

```
void main()
{
    Person* p1 = new Student(Person(111, "gogo"), 98.3);
    Person* p2 = new Student(Person(111, "gogo"), 87.5);

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

```
true
```

מימוש אופרטור == בפולימורפיזם הפתרון

```
class Student : public Person
{
private:
    float average;
public:
    Student(const Person& base, float average)
                : Person(base), average(average) {}

    virtual bool operator==(const Person& other) const
    {
        //if (!((Person&)(*this) == other))
        if (!Person::operator==(other))
                return false;

        const Student* temp = dynamic_cast<const Student*>(&other);
        return average == temp->average;
    }

    …
};
```

כעת מאחר והפרמטר הוא כמו בחתימה באבא זוהי דריסה, ומה- main נגיע למימוש זה

ביצוע שורה זו יגרור רקורסיה אינסופית כי השיטה באב וירטואלית...

כי other הוא const

```
void main()
{
    Person* p1 = new Student(Person(111, "gogo"), 98.3);
    Person* p2 = new Student(Person(111, "gogo"), 87.5);

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

false

C++11

Override

- אחת החולשות של תחביר הפולימפורפיזם הינו שמתכוונים לדרוס מתודה מסויימת, אבל בפועל מבצעים העמסה

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

class Derived : public Base
{
public:
    virtual void foo() { cout << "In Derived::foo\n"; }
};

void main()
{
    Base* b = new Derived();
    b->foo();

    delete b;
}
```

זוהי העמסה ולא דריסה!

```
In Base::foo
```

בגרסת C++11 נוספה מילת המפתח override שתפקידה להעיד שמתודה דורסת מתודה שנכתבה בבסיס ובכך למנוע את מצב ההעמסה במקום הדריסה

override דוגמת שימוש

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

class Derived : public Base
{
public:
  ✓ virtual void foo() const override { cout << "In Derived::foo\n"; }
  ✗ virtual void goo()       override { cout << "In Derived::goo\n"; }
};
```

מעיד שמתודה זו דורסת מתודה שבבסיס.
תקין תחבירית גם בלעדיה.

לא ניתן לדרוס מאחר והמתודה אינה קיימת בבסיס. היתרון בשימוש ב- override הוא קבלת השגיאה בזמן קומפילציה

override דוגמת שימוש נוספת

```
class A
{
public:
    virtual void foo1() const {}
            void foo2() const {}
    virtual void koo()  const {}
};

class B : public A
{
public:
✓   virtual void foo1() const override {}
✗           void foo2() const override {}
✗           void goo()  const override {}
✗           void koo()        override {}
};
```

לא ניתן להגדיר override למתודה שלא הוגדרה כ- virtual בבסיס

המתודה goo אינה קיימת בבסיס ולכן לא ניתן להגדירה כ- override

המתודה koo מוגדרת כ- const בבסיס ולכן זוהי העמסה ולא דריסה ← מילת המפתח override אינה תקינה

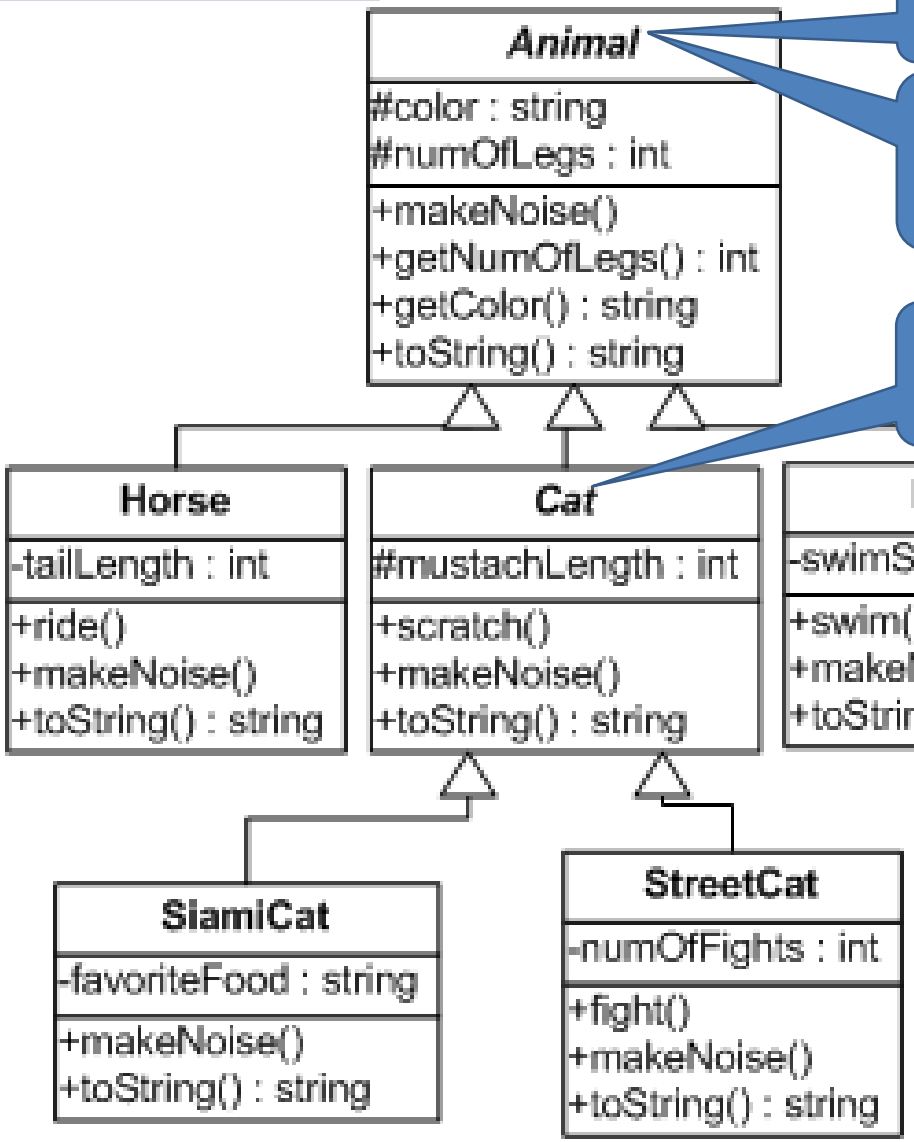
מחלקות אבסטרקטיות

בתרשים Class Diagram שם של מחלקה אבסטרקטית כתוב *בפונט מוטה (Italic)*

לא נרצה לאפשר יצירת אובייקטים מהמחלקה Animal, אלא רק של חיות ספציפיות

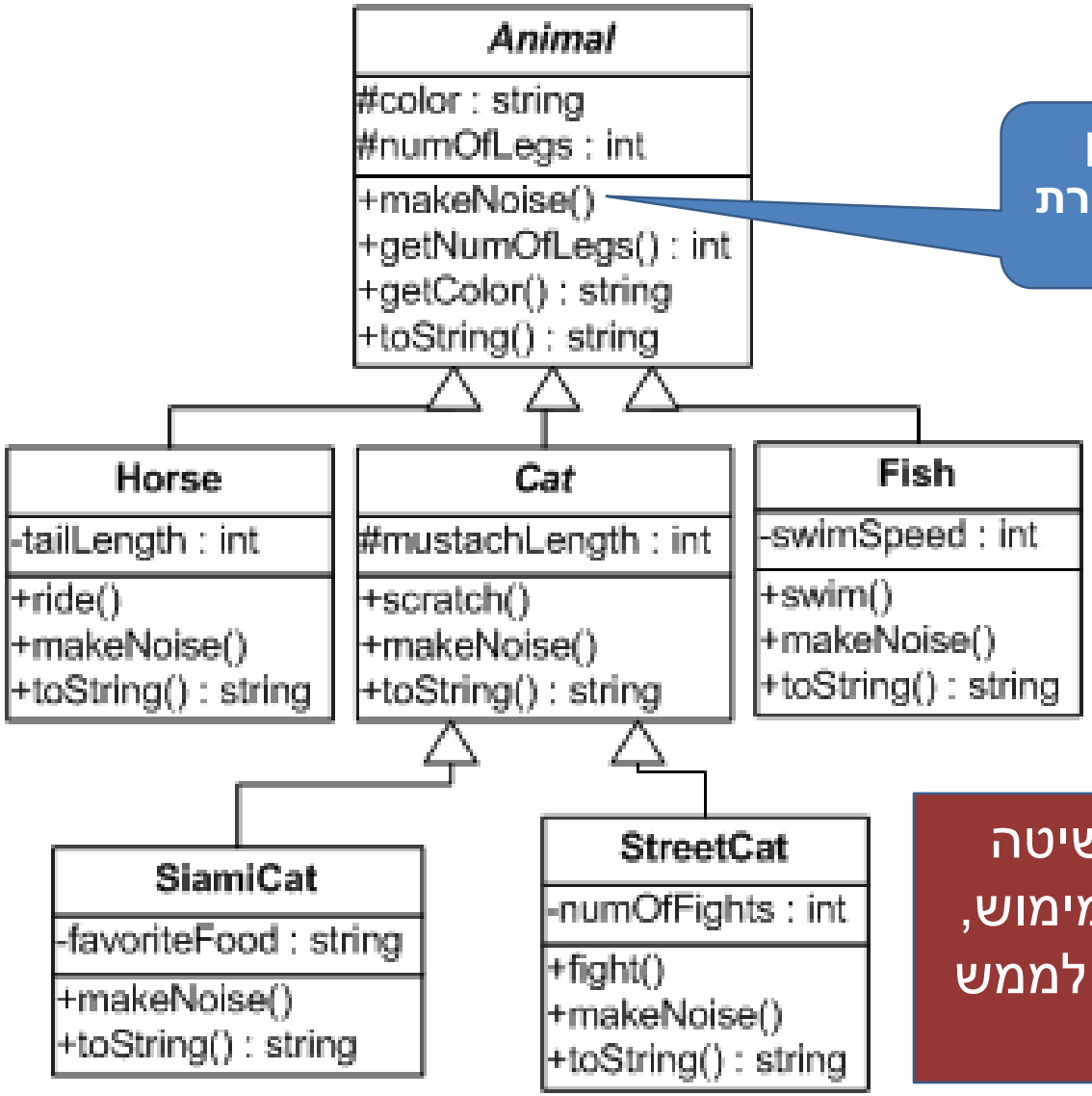
קיומה של המחלקה Animal חשוב שכן היא מכילה נתונים המשותפים לכל החיות – היא מהווה בסיס משותף

גם לא נרצה לאפשר יצירה של "סתם" חתול – כל חתול הוא או חתול רחוב או חתול סיאמי



מחלקה אבסטרקטית היא מחלקה שלא ניתן לייצר ממנה אובייקטים

שיטות אבסטרקטיות



כל חיה עושה קול ספציפי, ולכן במחלקה Animal אין מימוש ברירת מחדל למתודה makeNoise

שיטה אבסטרקטית היא שיטה שנרצה להגדיר בבסיס ללא מימוש, רק כדי להכריח את היורשים לממש אותה באופן ספציפי

מחלקת הבסיס האבסטרקטית Animal

```
class Animal
{
protected:
    char* color;
    int numOfLegs;
public:
    Animal(const char* color, int numOfLegs);
    Animal(const Animal& other);
    virtual ~Animal();
    const Animal& operator=(const Animal& other);


    virtual void makeNoise() const = 0;


    virtual void show() const = 0
    {
        cout << (typeid(*this).name() + 6) << " --> "
             << "Color: " << color << ", NumOfLegs:" << numOfLegs;
    }
};
```

בגלל שבמחלקה זו יש לפחות שיטה אבסטרקטית אחת, המחלקה הפכה לאבסטרקטית ולא ניתן לייצר אובייקטים מטיפוס Animal

שיטה ללא מימוש נקראית "אבסטרקטית טהורה" (pure virtual)

לא לשכוח לעשות את ה- d'tor וירטואלי!

איך "חיה" עושה קול? תלוי בחיה, לכן לא נרצה לממש שיטה זו באב, אלא להכריח כל אחד מהיורשים לממש שיטה זו

נרצה גם שכל חיה תממש את show בעצמה, ותציג גם את הנתונים הנוספים. יחד עם זאת כן רצינו לספק מימוש בסיסי ומשותף

קבלת שם המחלקה מהמחרוזת החוזרת מ- typeid

המחלקה היורשת Fish

```
class Fish : public Animal
{
    int numOfFins;
public:
    Fish(const char* color, int numOfFins)
                : Animal(color, 0), numOfFins(numOfFins) {}

    virtual void makeNoise() const override { cout << "Blu-Blu\n";  }
            void swim()      const          { cout << "Swimming\n"; }

    virtual void show() const override
    {
        Animal::show();
        cout << ", numOfFins: " << numOfFins;
    }
};
```

מחלקה יורשת חייבת לממש את כל השיטות האבסטרקטיות של הבסיס, אחרת גם היא תהיה אבסטרקטית

makeNoise ו- show דורסות את המימוש שבאב, ולכן מחלקה זו אינה אבסטרקטיות

המחלקה היורשת האבסטרקטית Cat

```
class Cat : public Animal
{
protected:
    int whiskersLen;

    Cat(const char* color, int whiskersLen)
        : Animal(color, 4), whiskersLen(whiskersLen) {}
public:
    virtual void makeNoise() const override { cout << "Miyaoooo!\n";}
            void scratch()   const          { cout << "Scratching!\n";}

    virtual void show() const override
    {
        Animal::show();
        cout << ", whiskerslen: " << whiskersLen;
    }
};
```

במידה והיינו מגדירים מצביע ל- Cat ובפועל יוצרים אותו כאחד הבנים, היינו מוסיפים במחלקה זו d'tor וירטואלי!

המחלקה Cat מממשת את כל השיטות שבאב, ולכן אינה אבסטרקטית.
מאחר ולא רצינו לאפשר יצירת Cat, שמנו את ה- c'tor ב- protected, כך שיהיה נגיש רק לבנים.

המחלקה SiamiCat

נשים לב שה- c'tor לא מקבל כפרמטר Cat, אלא את שדותיו כבודדים, מאחר ומי שכותב את ה- main, לא יכול לייצר אובייקט מטיפוס Cat (ה- c'tor שלו ב- protected ולכן נגיש רק ליורשים)

```
class SiamiCat : public Cat
{
    char* favoriteFood;
public:
    SiamiCat(const char* color, int whiskersLen, const char* favoriteFood);
    SiamiCat(const SiamiCat& other);
    ~SiamiCat();

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

    virtual void show() const override
    {
        Cat::show();
        cout << ", favoriteFood: " << favoriteFood;
    }
};
```

המחלקה StreetCat

```
class StreetCat : public Cat
{
    int numOfFights;
public:
    StreetCat(const char* color, int whiskersLen, int numOfFights)
        : Cat(color, whiskersLen), numOfFights(numOfFights) {}

    virtual void show() const override
    {
        Cat::show();
        cout << ", numOfFights: " << numOfFights;
    }
};
```

מחלקה אבסטרקטית שימוש (1)

מערך הטרוגני של סוגים שונים של חיות

```
void main()
{
    Animal* animals[3];

    for (int i = 0; i < 3; i++)
    {
        int type;
        cout << "Enter 1 for Fish, 2 for SiamiCat, 3 for StreetCat: ";
        cin >> type;

        switch (type)
        {
        case 1: animals[i] = new Fish("gold", 2);              break;
        case 2: animals[i] = new SiamiCat("gray", 3, "mice"); break;
        case 3: animals[i] = new StreetCat("black", 5, 31);   break;
        default: cout << "Invalid option\n";                  break;
        }
    }

    ...

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

```
Enter 1 for Fish, 2 for SiamiCat, 3 for StreetCat: 1
Enter 1 for Fish, 2 for SiamiCat, 3 for StreetCat: 2
Enter 1 for Fish, 2 for SiamiCat, 3 for StreetCat: 3
```

מחלקה אבסטרקטית שימוש (2)

```
void main()
{
    Animal* animals[3];
    ...

    for (int i = 0; i < 3; i++)
    {
        cout << "\nAnimal #" << (i + 1) << ": ";
        animals[i]->show();
        cout << endl;
        animals[i]->makeNoise();


        Cat* tempCat = dynamic_cast<Cat*>(animals[i]);
        if (tempCat)
            tempCat->scratch();

        Fish* tempFish = dynamic_cast<Fish*>(animals[i]);
        if (tempFish)
            tempFish->swim();
    }

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

מופעלת השיטה show של האובייקט המתאים, כי השיטה הוגדרה ב- Animal כ- virtual

אם makeNoise לא הייתה מוגדרת ב- Animal, לא ניתן היה לקרוא לה פה דרך המצביע, והיה צורך ב- casting

"לגרד" רק אם החיה היא "חתול"

"לשחות" רק אם החיה היא "דג"

מחלקה אבסטרקטית הפלט

```
Enter 1 for Fish, 2 for SiamiCat, 3 for StreetCat: 1
Enter 1 for Fish, 2 for SiamiCat, 3 for StreetCat: 2
Enter 1 for Fish, 2 for SiamiCat, 3 for StreetCat: 3

Animal #1:
Fish -> Color: gold, NumOfLegs:0, numOfFins: 2
Blu-Blu
Swimming

Animal #2:
SiamiCat -> Color: gray, NumOfLegs:4, whiskerslen: 3, favoriteFood: mice
Miyaoooo!
Scratching!

Animal #3:
StreetCat --> Color: black, NumOfLegs:4, whiskerslen: 5, numOfFights: 31
Miyaoooo!
Scratching!
```

מימוש האופרטור<< הבעיה

```
class Base
{
    int x;
public:
    Base(int x) : x(x) {}
    friend ostream& operator<<(ostream& os, const Base& b)
    {
        cout << "In operator<<(Base&)\n";
        os << b.x;
        return os;
    }
};
```

הבעיה: עבור b יש פניה לאופרטור << שהוגדר באב, מאחר והשיטה << לא יכולה להיות וירטואלית (היא גלובלית, ולכן אין אובייקט מפעיל)

```
class Derived : public Base
{
private:
    int y;
public:
    Derived(int x, int y) : Base(x), y(y) {}
    friend ostream& operator<<(ostream& os, const Derived& d)
    {
        cout << "In operator<<(Derived&)\n";
        os << (Base&)d << " " << d.y;
        return os;
    }
};
```

קריאה לאופרטור << שבאב

```
void main()
{
    Derived d(1, 2);
    cout << d << endl;
    cout << "--------------\n";

    Base* b = new Derived(3, 4);
    cout << *b << endl;
}
```

```
In operator<<(Derived&)
In operator<<(Base&)
1 2
--------------
In operator<<(Base&)
3
```

מימוש האופרטור<< הפתרון

```
class Base
{
    int x;
public:
    Base(int x) : x(x) {}

    virtual void toOs(ostream& os) const {}

    friend ostream& operator<<(ostream& os, const Base& b)
    {
        cout << "In operator<<(Base&)\n";
        os << b.x;
        b.toOs(os);
        return os;
    }
};

class Derived : public Base
{
private:
    int y;
public:
    Derived(int x, int y) : Base(x), y(y) {}

    virtual void toOs(ostream& os) const override {os << " " << y;}
    friend ostream& operator<<(ostream& os, const Derived& d) {…}
};
```

```
void main()
{
    Derived d(1, 2);
    cout << d << endl;
    cout << "--------------\n";

    Base* b = new Derived(3, 4);
    cout << *b << endl;
}
```

```
In operator<<(Base&)
1 2
--------------
In operator<<(Base&)
3 4
```

שיכפול איברים

```
void main()
{
    Animal* noahsArk[3];
    noahsArk[0] = new Fish("gold", 2);
    noahsArk[1] = new SiamiCat("gray", 3, "mice");
    noahsArk[2] = new StreetCat("black", 5, 31);

    Animal* dupArk[3];
    for (int i = 0; i < 3; i++)
    {
        if (typeid(*noahsArk[i]) == typeid(Fish))
            dupArk[i] = new Fish(*(Fish*)noahsArk[i]);
        else if (typeid(*noahsArk[i]) == typeid(SiamiCat))
            dupArk[i] = new SiamiCat(*(SiamiCat*)noahsArk[i]);
        else if (typeid(*noahsArk[i]) == typeid(StreetCat))
            dupArk[i] = new StreetCat(*(StreetCat*)noahsArk[i]);
    }

    for (int i = 0; i < 3; i++)
        cout << typeid(*dupArk[i]).name() + 6 << endl;

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

בדיקת מה טיפוס האיבר ה- i כדי לדעת איזה copy c'tor להפעיל...

```
Fish
SiamiCat
StreetCat
```

מימוש זה אינו אלגנטי, ותוספת של מחלקה יורשת חדשה תגרור שינוי בקוד המשכפל את המערך

שיכפול איברים הקוד הרצוי

```
void main()
{
    Animal* noahsArk[3];
    noahsArk[0] = new Fish("gold", 2);
    noahsArk[1] = new SiamiCat("gray", 3, "mice");
    noahsArk[2] = new StreetCat("black", 5, 31);

    Animal* dupArk[3];
    for (int i = 0; i < 3; i++)
    {
        if (typeid(*noahsArk[i]) == typeid(Fish))
            dupArk[i] = new Fish(*(Fish*)noahsArk[i]);
        dupArk[i] = noahsArk[i]->clone();
        else if (typeid(*noahsArk[i]) == typeid(StreetCat))
            dupArk[i] = new StreetCat(*(StreetCat*)noahsArk[i]);
    }

    for (int i = 0; i < 3; i++)
        cout << typeid(*dupArk[i]).name() + 6 << endl;

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

שיכפול כל איבר, בלי צורך לבדוק מהו טיפוס האובייקט בפועל!

שיכפול איברים פתרון השיטה clone

```
class Animal
{
protected:
    char* color;
    int numOfLegs;
public:
    Animal(const char* color, int numOfLegs);
    Animal(const Animal& other) ;
    virtual ~Animal();
    const Animal& operator=(const Animal& other) ;

    virtual void makeNoise() const = 0;
    virtual void show()      const = 0;

    virtual Animal* clone() const = 0;
};

class Fish : public Animal
{
    int numOfFins;
public:
    Fish(const char* color, int numOfFins);

    virtual void makeNoise() const override;
    virtual void show()      const override;
    void swim() const;

    virtual Animal* clone() const override { return new Fish(*this); }
};
```

משמש ליצירת העתק של עצם בזמן ריצה, מאחר ורק טיפוס הבסיס ידוע בזמן קומפילציה

- מטרת השיטה להחזיר העתק של האובייקט
- צריכה להיות ממומשת בכל היורשים
- תמיד תחזיר מצביע לאבא, שכן כך הוא יוכל להיות כל יורש

מימוש השיטה בכל היורשים, כך שיצירת העתק האובייקט תהייה דרך מעבר ב- copy c'tor

C++11

מתודות final

- עד כה לא הייתה דרך בשפה למנוע ממחלקה יורשת דריסת מתודה המוגדרת בבסיס
- לכן נוספה מילה המפתח `final`

```
class A
{
public:
    virtual void foo() const       {}
    virtual void goo() const final {}
    virtual void moo() const       {}
};

class B : public A
{
public:
  ✓ virtual void foo() const override       {}
  ✗ virtual void goo() const override       {}
  ✓ virtual void moo() const override final {}
};

class C : public B
{
public:
  ✗ virtual void moo() const override {}
};
```

חסר משמעות לשים בבסיס virtual+final

כי המתודה בבסיס היא final

נשים virtual+final רק כאשר יש גם override, ואז לא ניתן לדרוס מעבר לרמה זו

כי המתודה בבסיס היא final

תזכורת מוטיבציה להורדת רמת ההרשאה

- 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);
}
```

משמעות נוספת לשינוי הרשאת הירושה

```
class A
{
};

class B : protected A
{
};

void main()
{
    A* a = new B();
}
```

הקוד אינו מתקמפל מאחר ושינוי הרשאת הירושה באה לומר כי רוצים להסתיר את העובדה ש- B יורש מ- A, ולכן לא ניתן להגדיר את B כהפניה ל- A, שכן פעולה זו מסגירה את קיום ההורשה

conversion to inaccessible base class "A" is not allowed

'type cast': conversion from 'B *' to 'A *' exists, but is inaccessible

ביחידה זו למדנו:

- שימוש במצביע לאב ויצירת אובייקט כבן
- מוטיבציה לפולימורפיזם
- קישור סטטי לעומת קישור דינאמי
- הטבלה הוירטואלית
- מערך שאיבריו מטיפוסים שונים בעלי בסיס משותף
- d’tor וירטואלי
- זיהוי טיפוס בזמן ריצה
- dynamic cast
- מימוש נכון של << בפולימורפיזם
- מתודות override
- שיטות ומחלקות אבסטרקטיות
- השיטה clone
- מתודות final