

תכנות מונחה עצמים ב C++ תרגול

אוהד ברזילי
אוניברסיטת תל אביב

ירושה ב ++C

חלק מהדוגמאות במצגת לקוחות מתוך מצגת של אלחנן בורנשטיין.
למצגת המלאה: <http://www.cs.tau.ac.il/~borens/courses/oop-03b/>

כל הזכויות שמורות למחברים

Submission guidelines

- ❑ Submit the doxygen files as a zip
- ❑ The sources still should be submitted as is - no zip.
- ❑ In the file naming the ID of the submitter should be part of the file.
Such as `ex#q#_ID#.cpp`

תחביר ירושה

```
class Base1
{
    int var1;

public:
    Base1() {...} // constructor
    void OldFunc() {...}
    void Init() {...}
};

class Derived1 : [public|protected|private] Base1
{
    int var2;

public:
    Derived1() {...} // constructor
    void Init() {...} // override
    void Do(){...}
};
```

Base1 b;

var1

Derived1 d;

var1

var2

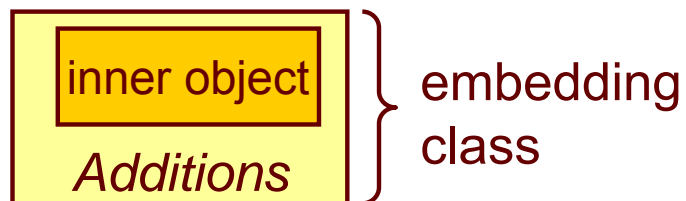
**d.Init();
d.OldFunc();
d.Do();**

הבן מכיל את אביו

- ניתן לראות את מחלקת הבסיס ממש כמו השדה הראשון במחלקה הנגזרת



- בהקשר זה ירושה מספקת כלי טכני למימוש נוח של האצלה (delegation)



- ההבדלים בין השניים לא תמיד ברורים:
 - בירושה הבן הוא סוג של אביו (is-a)
 - בהאצלה העצם המכיל הוא לקוח (uses) של השדה המוכל

דוגמא

```
class GraphicItem
{
    int color;
public:
    GraphicItem() {...} //constructor
    void Draw() {...}
    void ChangeColor(int n_clr){...}
};
```

```
class CPoint : public GraphicItem
{
    int m_x, m_y;
public:
    CPoint() {...} // constructor
    void Draw() {...} // override
    void Align(){...}
};
```

```
...
GraphicItem G, *pG;
CPoint P, *pP;
...

```

	<u>Logical</u>	<u>Physical</u>
G = P	<input checked="" type="checkbox"/>	Slicing
P = G;	<input type="checkbox"/>	<input checked="" type="checkbox"/>
pG = &P	<input checked="" type="checkbox"/>	Polymorphism
pP = &P	<input type="checkbox"/>	<input checked="" type="checkbox"/>
...		

ירושה

- על אף שמחלקה נורשת היא בדרך כלל מקרה פרטי (צמצום) של מחלקת הבסיס, היא ממומשת ע"י הרחבה פיזית (הוספת שדות) של הבסיס
- המחלקה הנגזרת יורשת את מחלקת הבסיס על כל שדותיה ותכונותיה, אך לא כולם נגישים לה (כתלות בהרשאות הגישה)
- בעת הסתרה של תכונה ע"י הדגרה מחדש שלה עדיין ניתן לגשת לתכונה המקורית ע"י ציון השם המלא: `Base::function`

בנאים

- בנאים, מפרקים, אופרטור ההשמה לא נורשים
- ממש כשם ששדות מוכלים יש לאתחל בשורת האתחול של הבנאי (אחרת הם יאותחלו ע"י בנאי ברירת המחדל) כך יש לאתחל את מחלקת הבסיס (ממש כאילו היא שדה של המחלקה הנגזרת)
- סדר הבנייה:
 - בנאי מחלקת הבסיס (שמאתחל את שדותיו לפי הצורך)
 - בנאי השדות הנוספים של המחלקה הנגזרת
 - בנאי המחלקה הנגזרת

מפרקים

□ מפרקים יקראו בסדר ההפוך

Embedded
Objects

```
class CRect
{
    CPoint pnt1, pnt2;

public:
    CRect(int x1, int y1, ...) : pnt1(x1,y1), pnt2(0,0) {...} // c'tor
};
```

Inheritance

```
class CGraphicRect : public GraphicItem
{
    ...
public:

    CGraphicRect(int color) : GraphicItem(color) {...} // c'tor
};
```

אופרטור ההשמה

- אופרטור ההשמה לא נורש
- כאשר מתבצעת השמה בין שני עצמים של מחלקה שלא מימשה את אופרטור ההשמה מתבצעת השמת ברירת המחדל שדה-שדה
- במהלך השמת ברירת המחדל מופעל אופרטור ההשמה של כל אחד מהשדות וכן של מחלקת הבסיס
- יש להקפיד במימוש אופרטור השמה לקרוא מפורשות לאופרטור ההשמה של כל אחד מהשדות וכן לאופרטור ההשמה של מחלקת הבסיס

אופרטור ההשמה

- תזכורת: נכתוב בעצמנו אופרטור השמה למחלקה רק אם היא מקצה משאבים בצורה דינאמית (כגון הקצאת זכרון)
- אותו הכלל תקף גם לגבי בנאי ההעתקה והמפרק

דוגמא: מחסנית

```
class CStack
{
    int*      m_arr;
    int      m_size;
    int      m_pos;

public:
    CStack(int size) : m_size(size)
    {
        m_arr = new int[m_size] ;
        m_pos = 0;
    }
    CStack(const CStack& S)
    {
        m_arr = NULL;
        *this = S;
    }
    ~CStack() { delete[] m_arr; }
    const CStack& operator=(const CStack& S);
};
```

הערה:

- הדוגמא אינה חינוכית. אנו לא נשתמש במצביע ל int וננהל זכרון בעצמנו אלא נשתמש מלכתחילה ב vector
- דוגמא זו מוצגת בשביל להדגים מחלקה אשר מנהלת משאב בצורה דינאמית

מחסנית

```
void Push(int val)
{
    if (m_pos >= m_size)
        throw "Out of Range";
    else
        m_arr[m_pos++] = val;
}
int Pop()
{
    if (m_pos <= 0)
        throw "Stack Empty";
    return m_arr[--m_pos];
}
void Print() const
{
    if (m_pos == 0)
        cout<<"Stack Empty"<<endl;
    else
        for (int i=0 ; i<m_pos ; i++)
            cout<<m_arr[i]<<" ";
    cout<<endl;
}
};
```

מחסנית

```
const CStack& CStack::operator=(const CStack& S)
{
    if (&S != this)
    {
        delete[] m_arr;
        m_size = S.m_size;
        m_pos = S.m_pos;
        m_arr = new int[m_size];
        for (int i=0 ; i < m_pos ; i++)
            m_arr[i] = S.m_arr[i];
    }
    return *this;
}
```

מחסנית משוכללת

• נרצה להגדיר מחסנית אשר בנוסף לפעולתה שומרת גם את סכום האברים הנמצאים בתוכה:

```
class CSumStack : public CStack
{
    int m_sum;

public:
    CSumStack(int size) :
        CStack(size), m_sum(0){}

    void Print() const
    {
        CStack::Print();
        cout<<"Sum="<<m_sum<<endl;
    }
};
```

```
void Push(int val)
{
    CStack::Push(val);
    m_sum += val;
}

int Pop()
{
    int val = CStack::Pop();
    m_sum -= val;
    return val;
}

};
```

מחסנית משוכללת - דיון

- במחסנית המשוכללת לא חסכנו שום דבר בירושה ממחסנית. יכולנו באותה מידה להשתמש במחסנית ע"י האצלה (delegation)
- ניזכר בדוגמא של `Vec` שהכיל `vector` כשדה. שם היינו צריכים לחזור על כל המתודות של וקטור עבור `Vec` למרות שלא היה שום שינוי במימוש, פרט לשתי מתודות (אופרטור `[]`)

דוגמא: הקצאת המשאבים בנגזרת

```
class A
{
    int var1;
};

class B : public A
{
    char* str;

public:
    B(const char* s)
    {
        str = new char[strlen(s)+1];
        strcpy(str,s);
    }
    ~B() { delete[]str; }
```

```
B(const B& b): A(b)
{
    str= new char[strlen(b.str)+1];
    strcpy(str,b.str);
}

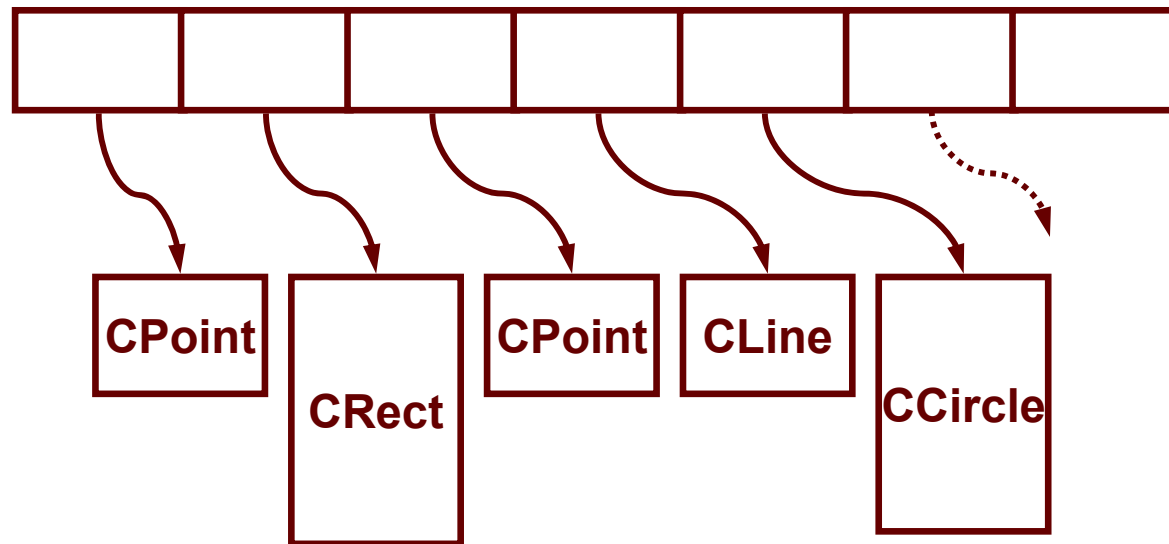
const B& operator=(const B& b)
{
    // Copy the A part
    A::operator=(b);
    // A& a = *this;
    // a=b;

    // Copy the B part
    delete[]str;
    str = new char[strlen(b.str)+1];
    strcpy(str,b.str);
    return *this;
}
};
```

רב צורתיות

ביצוע פעולות לפי הטיפוס הדינאמי של העצם ולא לפי הטיפוס הסטטי של המצביע: □

```
CGraphicalItem* items[7];
```



רב צורתיות ב C++

□ בעיה. ברירת המחדל של קישוריות המתודות היא סטטית ולא דינאמית:

```
class A
{
    void Do1();
    void Do2();
};

class B : public A
{
    void Do2();
    void Do3();
};
```

```
int main()
{
    B b;
    b.Do1();      ✓
    b.Do2();      ✓
    b.Do3();      ✓

    A* pa = new B;
    pa->Do1();    ✓
    pa->Do2();    ☹
    pa->Do3();    ✗
}
```

זימון מתודה דינאמי וסטטי

□ אם המתודה לא הוגדרה כ `virtual`

■ קישור המתודה יתבצע בזמן קומפילציה (`static binding`) ללא קשר לטיפוס העצם המפעיל בפועל

□ אם המתודה הוגדרה כ `virtual`

■ קישור המתודה יתבצע רק בזמן ריצה, וזאת לפי טיפוס העצם המפעיל (הטיפוס הדינאמי)

מה יודפס?

```
class A
{
public:
    virtual void f()
    {cout<<"A";}

    void h()
    {cout<<"a";}

    virtual void g()
    {
        f();
        h();
    }
};
```

```
class B : public A
{
public:
    virtual void f()
    {cout<<"B";}

    void h()
    {cout<<"b";}
};
```

```
class C : public B
{
public:
    virtual void f()
    {cout<<"C";}

    void h()
    {cout<<"c";}
};
```

```
main()
{
    B* p;
    p = new C;
    p->g();
    p->h();
};
```

1. Abc
2. Cab
3. Acb
4. Cac
5. Ccc

מתודות וירטואליות

- מתודה שהוגדרה וירטואלית במחלקה מסוימת תהיה כזו בכל ההיררכיה גם אם בהמשך לא מצוינת המילה השמורה virtual
- כמובן, שרצוי משיקולי קריאות לציין virtual לאורך כל המופעים של אותה מתודה בהמשך
- על כל מחלקה שהמפרק שלה עושה פעולה לא טריוויאלית להגדיר את המפרק כ virtual, ובעצם את כל המפרקים בהיררכית הירושה
- מנגנון הפונקציות הוירטואליות לא עובד עבור מתודות הנקראות מתוך בנאים ומפרקים

מפרקים לא וירטואלים מה יודפס?

```
class Base {
public:
    ~Base(){cout << "Base::~destructor\n"; }
};

class Derived : public Base {
public:
    ~Derived(){cout << " Derived::~destructor\n"; }
};

int main()
{
    Base *bd = new Derived();
    delete bd;
}
```

פונקציות וירטואליות בבנאים ומפרקים מה יודפס?

```
class Base {  
public:  
    Base(){f();}  
    ~Base(){f();}  
    virtual void f() { cout << "Base::f\n"; }  
};
```

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

```
Derived d;
```


דוגמא

```
class CFood
{
public:
    virtual char* Name() const
    { return "Food";}

    virtual int Calories() const
    { return 0; }

    void Print() const
    {
        cout<<"Name:"<<Name()<<endl;
        cout<<"Calories:"<<Calories()<<endl;
    }
};

class CHumus : public CFood
{
public:
    virtual char* Name() const
    { return "Humus";}

    virtual int Calories() const
    { return 1000; }
};
```

```
class CPita : public CFood
{
public:
    virtual char* Name() const
    { return "Pita";}

    virtual int Calories() const
    { return 150; }
};

class CHamutsim : public CFood
{
public:
    virtual char* Name() const
    { return "Hamutsim";}

    virtual int Calories() const
    { return 20; }
};
```

דוגמא

```
CFood* InputOrder()
{
    int type;
    cout<<"Ma Ba Lecha Achi? (1-Humus,
        2-Pita, 3-Hamutsim)?"<<endl;
    cin>>type;
    switch (type)
    {
        case 1: return new CHumus;
        case 2: return new CPita;
        case 3: return new CHamutsim;
        default: return NULL;
    }
}
```

```
void main()
{
    CFood* Order[3];

    // Get the order
    for (int i=0; i<3 ;i++)
        Order[i] = InputOrder();

    // Print the order
    for (i=0; i<3 ;i++)
        Order[i]->Print();

    // Just checking
    for (i=0; i<3 ;i++)
    {
        cout<<Order[i]->Name()<<" , "
            << Order[i]->Calories()<<endl;
        delete Order[i];
    }
}
```

דיון

- דוגמת החומוס מדגימה שימוש במתודות וירטואליות
- ואולם כדי לממש מאכלים שונים היודעים להדפיס את עצמם ולספר כמה קלוריות יש בהם, ניתן גם היה להשתמש במחלקה עם 2 שדות, האחד מחרוזת והאחר `int` וכל אחד מטיפוסי המנות השונות היה עצם של המחלקה הזו
- מתי מבחר עצמים שונים הם מופעים שונים של אותה מחלקה ומתי הן מחלקות שונות בעלות בסיס משותף?

מחלקות מופשטות

- מחלקות אשר לא ניתן ליצור מופעים ישירים שלהן
- לפחות אחת המתודות שלהן `pure virtual`
- שימו לב: ניתן לייצר הפניות ומצביעים למחלקה מופשטת (אשר בפועל יצביעו למחלקת נגזרת)

דוגמא

```
class B
{
public:
    virtual void Print() const = 0;
};

class D1 : public B
{
    int i;
public:
    D1(int ai) : i(ai) {}

    virtual void Print() const
    { cout<<i<<endl; }
};
```

```
void f1 (B* pb) {}

void f2 (B b) {} // ERROR

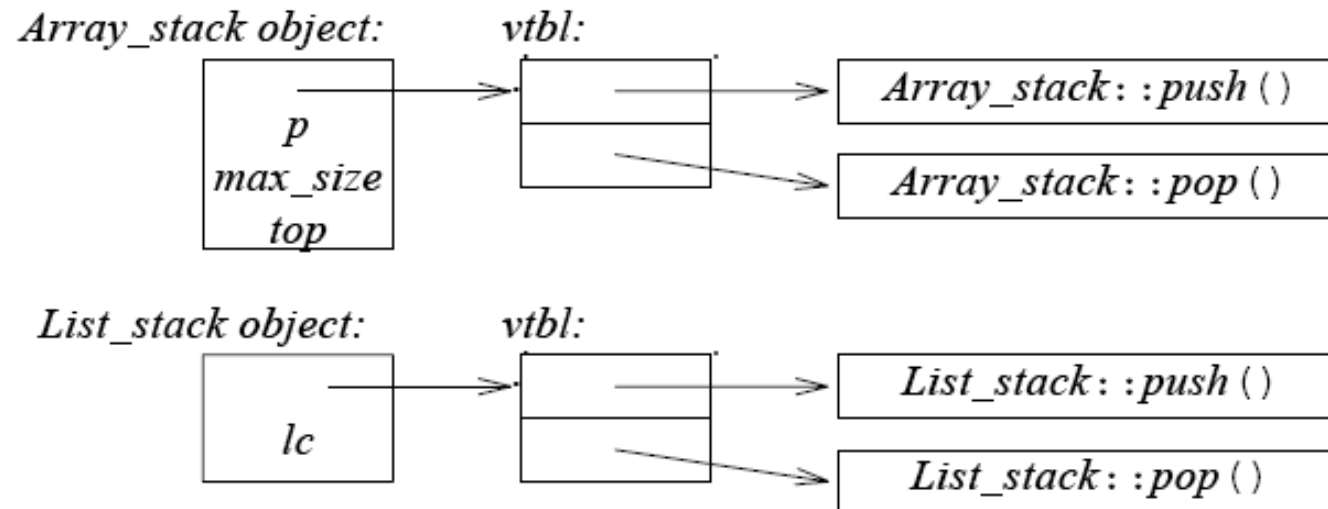
B& f3 () {...}

B f4 () {} // ERROR

int main()
{
    B* pb = new D1(12);
    pb->Print();
    delete pb;
}
```

קישור דינאמי

- למחלקות אשר מכילות מתודה וירטואלית אחת לפחות
- נוסף לכל עצם מצביע לטבלה וירטואלית
- הטבלה (אחת לכל מחלקה) מכילה מצביעים לפונקציות שיש להפעיל לפי טיפוס העצם בפועל



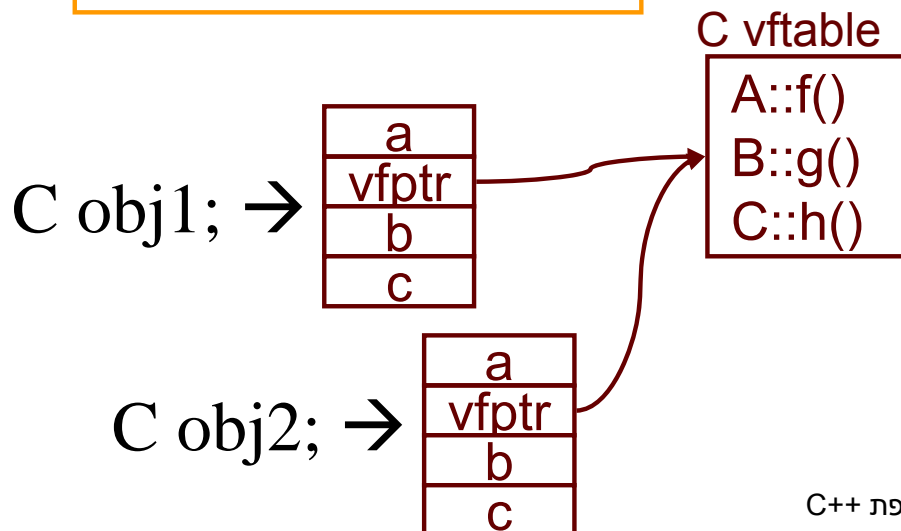
קישור דינאמי

כל קריאה לפונקציה וירטואלית דורשת הסבה של הפוינטר דרך הטבלה לפונקציה המבוקשת: □

```
class A
{
    int a;
public:
    virtual void f() {...}
    virtual void h() {...}
    virtual void g() {...}
};
```

```
class B : public A
{
    int b;
public:
    virtual void g() {...}
};
```

```
class C : public B
{
    int c;
public:
    virtual void h() {...}
};
```



A* p = new C;

p->g();

means: p->vfptr->g();

translates to: (*(p->vfptr)[1])(p,args);