

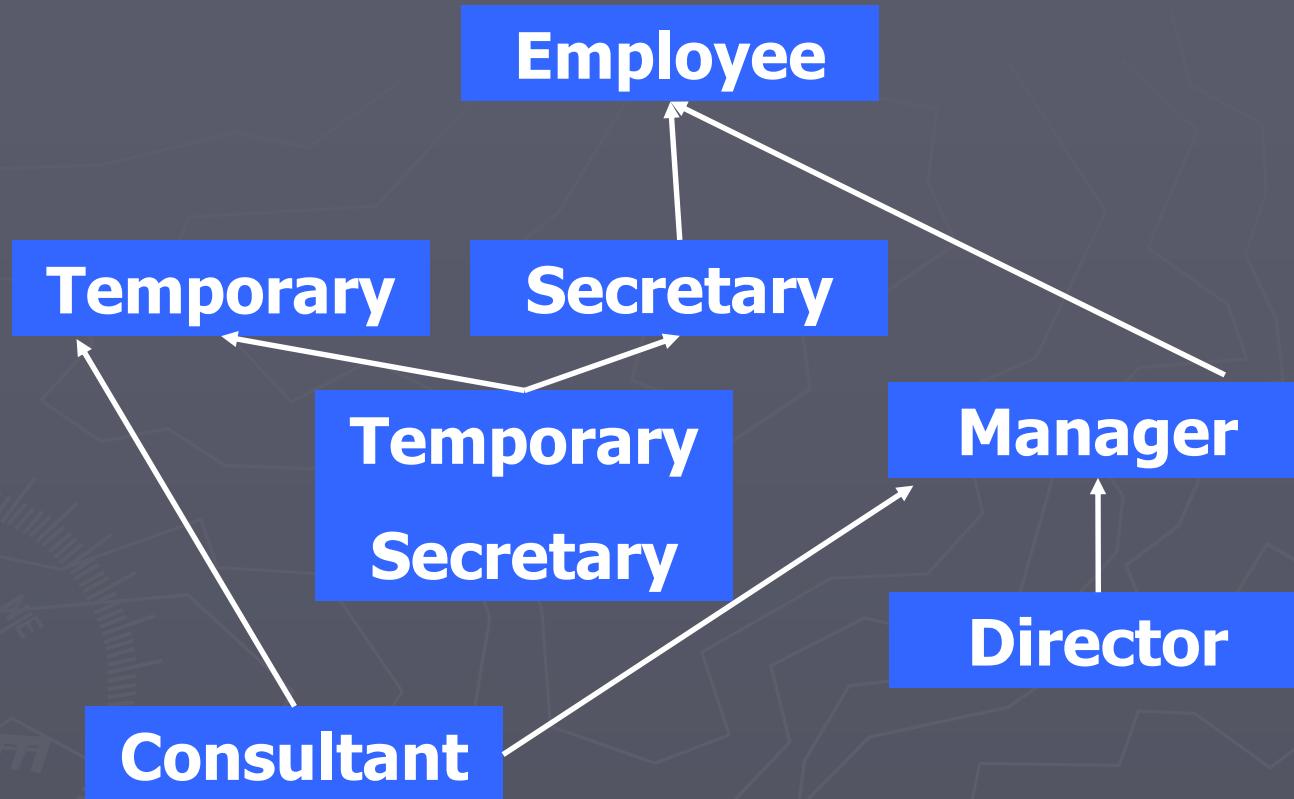
Object Oriented Programming

Lecture # 7 – Inheritance cont...
& Polymorphism

הורשה – חזרה מהירה

- ❖ הורשה – יצרת תת סוג (הורשה פומבית).
 - ❖ כל מה שיש באב – עובר לבן.
 - ❖ Hiding, וגישה דרך שם מלא.
 - ❖ אין לבן גישה למשתני private.
 - ❖ protected.
- ❖ 4 מוגנות הקומפיילר בהורשה.
 - ❖ יצרה והריסה בהורשה ורשימת אתחול.
 - ❖ overriding Hiding
 - ❖ Upcasting
- ❖ הורשה פרטית – בדר"כ להורשת שימוש.
- ❖ הורשה מוגנת (protected) – חסרת שימוש.
- ❖ הורשה ייחודית מול הורשה מרובה.

הורשה מרובה



הורשה מרובה - הנחיות

❖ **העקרונות של הורשה ייחודית תקפים גם בהורשה מרובה!**

- ❖ במידה ומחלוקת האב (Base) חייב לקבל פרמטרים לבנאי – הבן חייב להעביר אותם דרך שורת אתחול!
- ❖ המחלוקת הנגזרת יכולה להשתמש בethods ובדוחות של מחלוקת הבסיס גם אם הן חבויות (hidden) תוך שימוש **בשם המלא** (():func). זה מאד שימושי כדי לטפל בבעיה של **מетодות עם שמות זרים בחלוקת הבסיס השונות** (אחרת יש זו משמעויות – **ambiguity**).
- ❖ המחלוקת הנגזרת יכולה להשתמש (בקיפון) בדוחות הפרטיים של מחלוקות הבסיס – על ידי ממשך הפומבי או המוגן (protected) של מחלוקות הבסיס.
- ❖ בהורשה, סוג ההורשה (public, protected, private) חייב להיות מצוין לכל מחלוקת בסיס בנפרד!

הירושה מרובה - דוגמא

❖ בהינתן שתי מחלקות A ו-B כך שלשתיهن יש רק שני שדות - שלים ושביר, כך שהשבר מייצג את המשקל של הערך השלים..

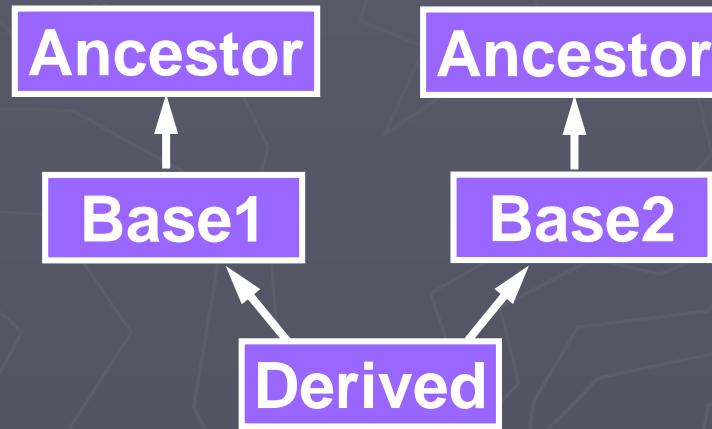
ל-A: float m_y int m_x
ל-B: int m_z float m_x

כיתבו את מחלקת C היורשת משתי המחלקות ולה מתודה אחת בלבד: מתודה המחשבת את הממוצע החבוני של ערכי שתרי המחלקות.

הורשה מרובה – אב קדמון משותף

Common Ancestor

- ❖ ראיינו שכאשר יורשים מספר שדות (או מתיודות) עם שם זהה אנו עדים יכולים (וחייבים) לגשת אליהם תור שימוש **בשם המלא!**
- ❖ אך, ישנו מצב מעניין, בו בהכרח יהיו לנו שדות ומתיודות עם שמות זהים. מתי?



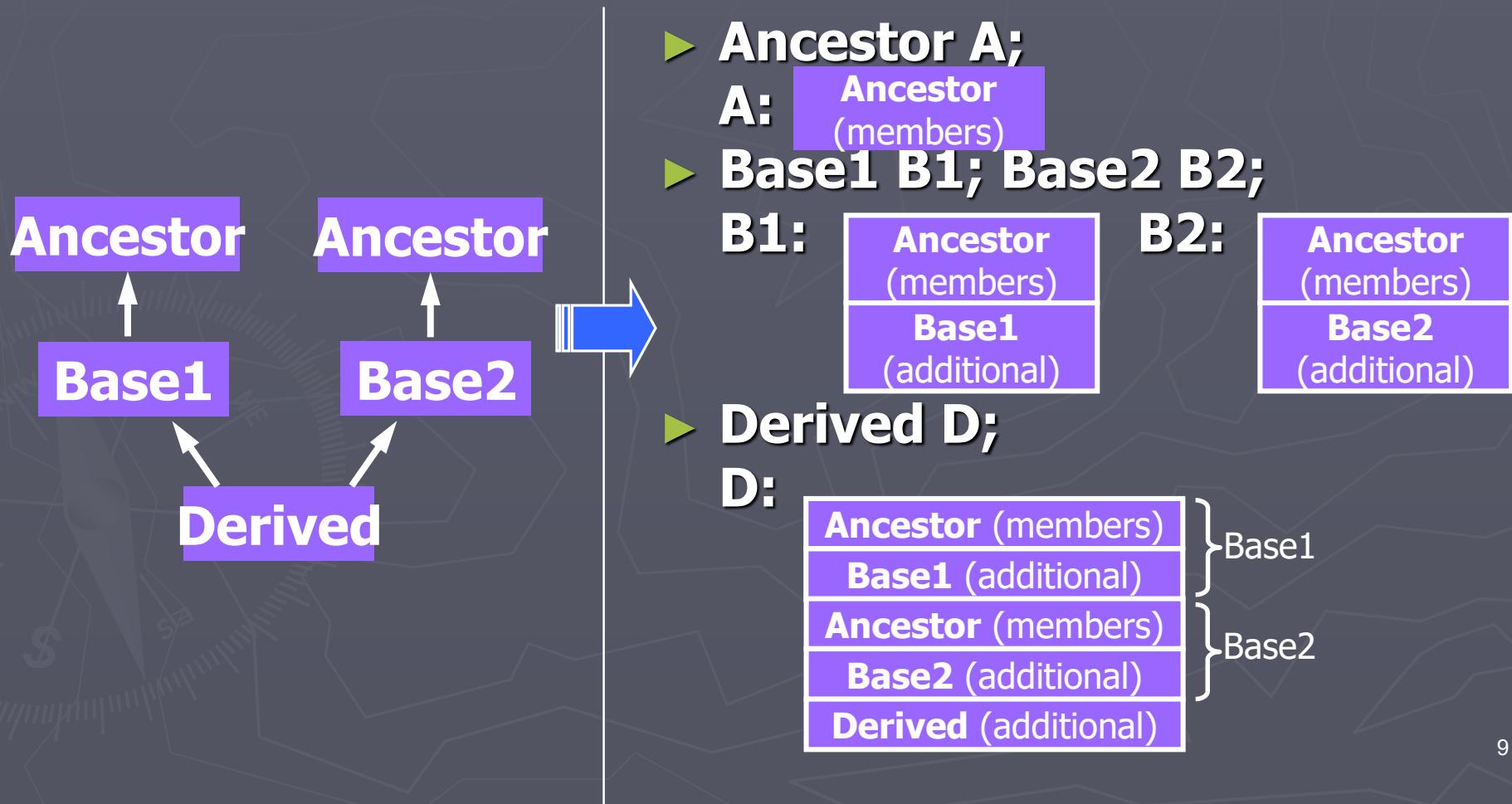
כasher שתי מחלקות הבסיס בעצמן יורשות ממחולקות בסיס משותפת
(אב קדמון – **ancestor**): כל ה-members של האב הקדמון
יופיעו פעמים במחלקה הנגזרת (Derived) -> פעם אחת דרך
1. Base2 ופעם שנייה דרך Base1.

בעיות עם הורשה מרובה

- ❖ מחלוקת בסיס יכולה להופיע רק פעם אחת ברשימת הורשה.
 - ❖ **לדוגמה:**
class A : public B, public B
- ❖ **היא שגיאה!**
- ❖ מחלוקת בסיס יכולה להופיע מספר פעמים בהיררכיות הורשה!
 - ❖ עלול לגרום לדו-משמעות (ambiguity) בקריאה לMETHODS וSHOTS.
 - ❖ גורם להזקמת זיכרון כפול (לא יעיל כמו כן עלול לגרום לטעויות!)

אב קדמון משותף

❖ מדוע אנו בעצם מקבלים העתק כפול? למה זה קורה?



Common Ancestor

למה זה טוב?

- ❖ ירושא שגורם להופעה כפולה של האב קדמון משותף היא דבר שימושים בו רק כאשר בצאצאים סופי יש צורך בהחזקת שתי העתקים מהאב הקדמון! (->טיעון לוגי!).
- ❖ כאשר משתמשים בשדות / מетодות מאותו אב קדמון – יש צורך לציין את **מלוא מסלול ההורשה** לשם על מנת שנדע להבחן באיזה שדה בדיק מדבר!

Base2::Ancestor::member ❖

או: Base1::Ancestor::member ❖

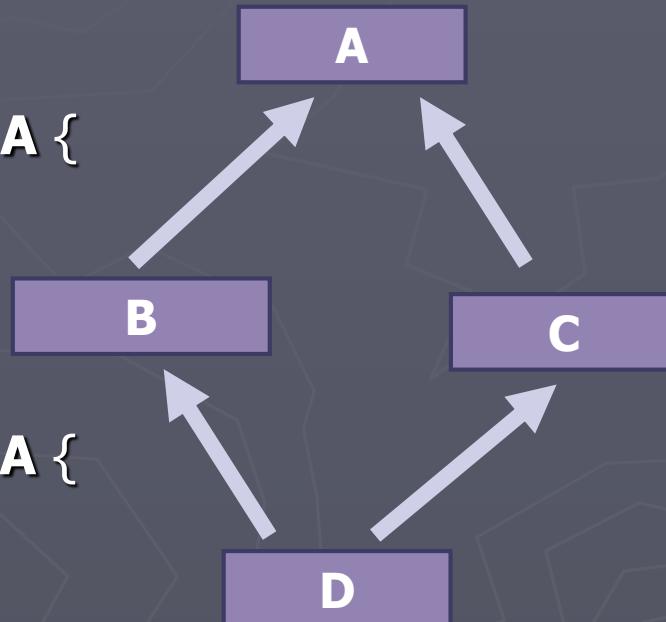
דוג' ממה קורה במקורה הבא?

```
class A {  
public:  
    int a;  
};
```

```
class B: public A {  
public:  
    int b;  
};
```

```
class C: public A {  
public :  
    int c;  
};
```

```
class D: public B, public C {  
public :  
    int d;  
};
```



A a;
B b;
C c;
D d;

ambiguous
conversion
s from 'D *'
to 'A *'

a.a = 1;
b.a = 2;
c.a = 3;
d.a = 4
A* pa = &d;

error:
ambiguous
access of 'a'

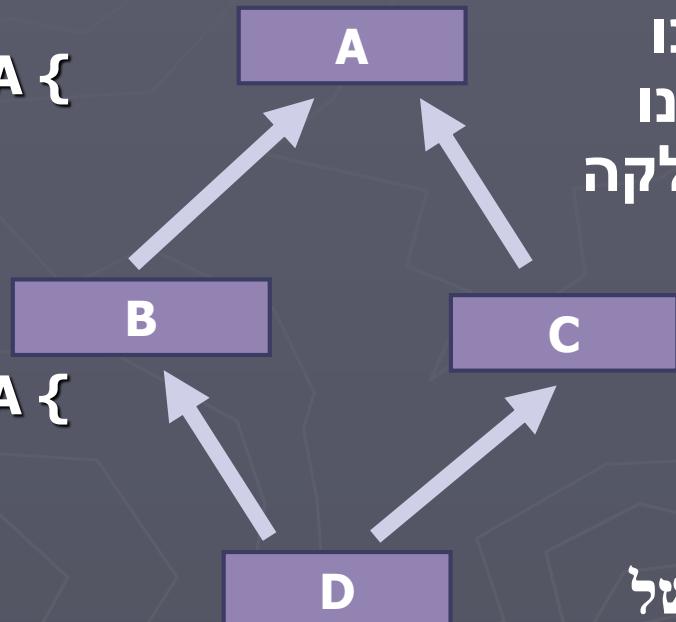
דוג': מה קורה במקרה הבא?

```
class A {  
public:  
    int a;  
};
```

```
class B: public A {  
public:  
    int b;  
};
```

```
class C: public A {  
public :  
    int c;  
};
```

```
class D: public B, public C {  
public :  
    int d;  
};
```

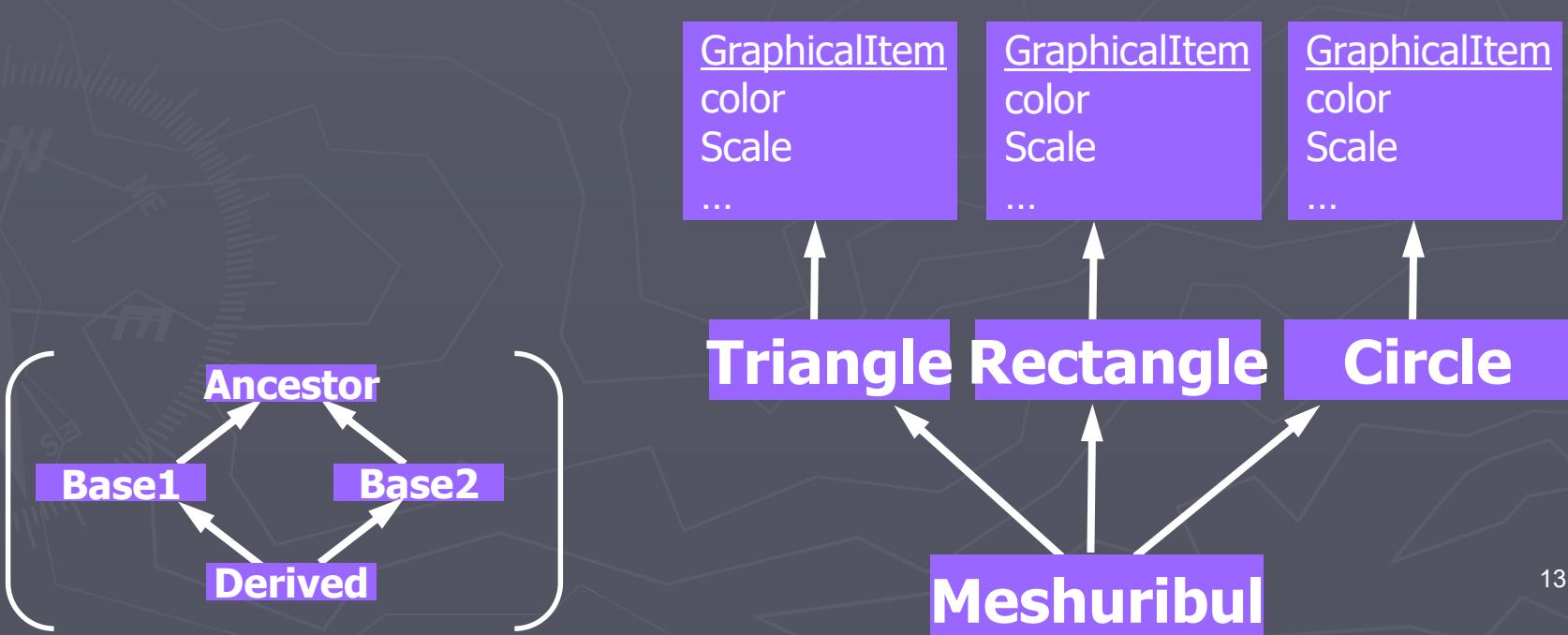


- כמו שאמרנו, כאשר אנחנו יורשים מחלוקת, אזי יש לנו בפועל אובייקט של המחלוקת שירשו בתוך האובייקט של המחלוקת היורשת במקרה זה, יש ל-B אובייקט של A, וכן יש ל-C אובייקט של A. מכיוון של-D יש אובייקט של C (שיש בתוכו A) ואובייקט של B (שיש בתוכו A) אזי ב-D יש שני אובייקטים של A

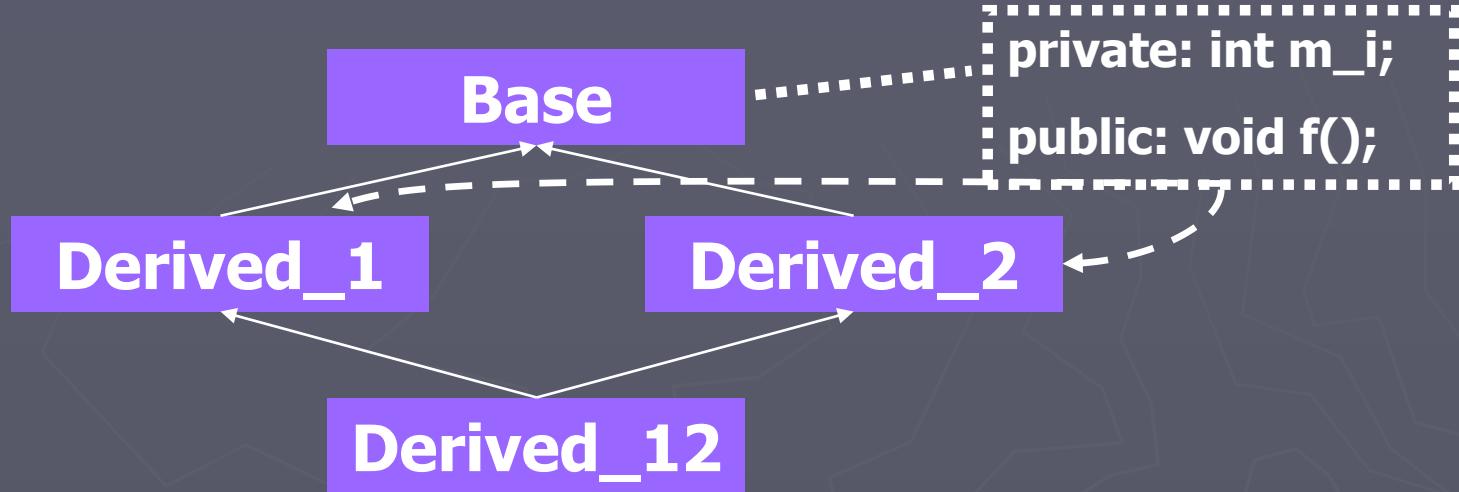
אב קדמון משותף – הורשה וירטואלית

Virtual Inheritance

- ❖ בדוגמה הקודמת, כל השדות של האב הקדמון המשותף הופיעו פעמים במחלקה הנגזרת.
- ❖ **ברוב מוחלט של המקרים אנו לא נרצה מצב זה (ניתוח לוגי!).**



Virtual Inheritance



- ❖ בהורשה וירטואלית יש רק עותק אחד מכל מחלקה אפילו אם (בגלל הורשה מרובה) היא מופיעה מספר פעמים בהיררכיה ההורשה.

```
class Derived_1 : virtual public Base {...};
```

```
class Derived_2 : virtual public Base {...};
```

```
class Derived_12 : public Derived_1, public Derived_2 {...};
```

```
class Base { public: int i; };

class Derived1a : virtual public Base {
    public: int j;
};

class Derived1b : virtual public Base {
    public: int k;
};

class Derived2 : public derived1a, public derived1b {
    public: int product() {return i*j*k;}
};

int main() {
    derived2 obj;
    obj.i=10; obj.j=3; obj.k=5;
    cout<<“product is: “<<obj.product()<<’\n’;
    return 0;
}
```

הורשה וירטואלית - הנחיות

- ❖ אם אנו מעוניינים שיהיה רק עותק אחד מהאב הקדמון – נירש תוך שימוש בהורשה וירטואלית!!! (**Virtual Inheritance**)
- ❖ הורשה וירטואלית מדrica את הקומפיילר לשימוש באותו מקום בזיכרון עבור שני העתקים של מחלוקת האב הקדמון.
- ❖ כל מחלוקת בסיס (שיורשת מהאב הקדמון) תכלול מצביו לאותו מקום בזיכרון.
- ❖ המיקום וכן הגישה למיקום זה שקבפים למכונת (הכל נעשה על ידי הקומפיילר!)
- ❖ קיימ רק עותק אחד ממחלוקת האב הקדמון (ומכל השדות שלה)! ההשמה האחרונה (מבחינת המחלוקות הקוראות) היא שתקבע את הערך לשדות.
- ❖ **שימוש באב קדמון וירטואלי** הוא כאשר האב הקדמון מייצג את אותו אובייקט (ישות) בכל מחלוקות הבסיס ולכן צריך להופיע רק פעם אחת! (-> טיעון לוגי!)

הורשה מרובה וירטואלית – כיצד?

- ❖ **בהורשה רגילה** כאשר המחלוקת נוצרת: הבנאי של המחלוקת היורשת קורא קודם קודם כל לבנאי של מחלוקת האב ע"מ ליצור את אובייקט האב המוכל בתוך המחלוקת היורשת.
 - ❖ לבנאי אב -> לבנאי בן.
- ❖ **בהורשה וירטואלית:** במקום לקרוא לבנאי של האבא לפני שהיא קוראת לבנאי שלה – היא מצפה לקבל מצבייע אל אובייקט מסווג האבא שכבר קיימ. מכאן ולהלאה המחלוקת היורשת תתייחס לאובייקט המוצבע בתוך אובייקט האבא שאמור להיות מוכל בתוכה.
- ❖ **לבנאי אב קדמון ->** יוצר אובייקט והפונטרא אליו נשלח לכל הבנאים האחרים!

הורשה מרובה וירטואלית – כיצד? (2)

- ❖ המשמעות היא ש: **האובייקט בrama התchaptonה אמור לקרוא לבנאי של האב הקדמון** שמננו מתבצעת הורשה ווירטואלית **לפנֵי** שהוא קורא לבנאי של האבא שלו.
- ❖ את האובייקט שנוצר (מהאב הקדמון) כל לבנאי יעביר לבנאי של האבא שלו (פונטר) עד שיגיעו לחלוקת שבת בוצעה ההורשה הווירטואלית.

Common Ancestor

Simple Inheritance

- ▶ **Ancestor A;**
A: **Ancestor**
(members)
- ▶ **Base1 B1; Base2 B2;**
B1: **Ancestor**
(members)
Base1
(additional)
- ▶ **Derived D;**
D:
Ancestor (members)
Base1 (additional)
Ancestor (members)
Base2 (additional)
Derived (additional)

The diagram illustrates simple inheritance. At the top is a purple box labeled "Ancestor (members)". Below it are two blue boxes: "Base1" (containing "near ptr" and "Base1 (additional)") and "Base2" (containing "near ptr" and "Base2 (additional)"). Arrows point from both base boxes to a bottom purple box labeled "Derived (members)". To the left of the base boxes is another purple box labeled "Ancestor (members)" which also points to the "Derived" box. Brackets on the right group "Base1" and "Base2" under the heading "Base1" and "Base2" respectively.

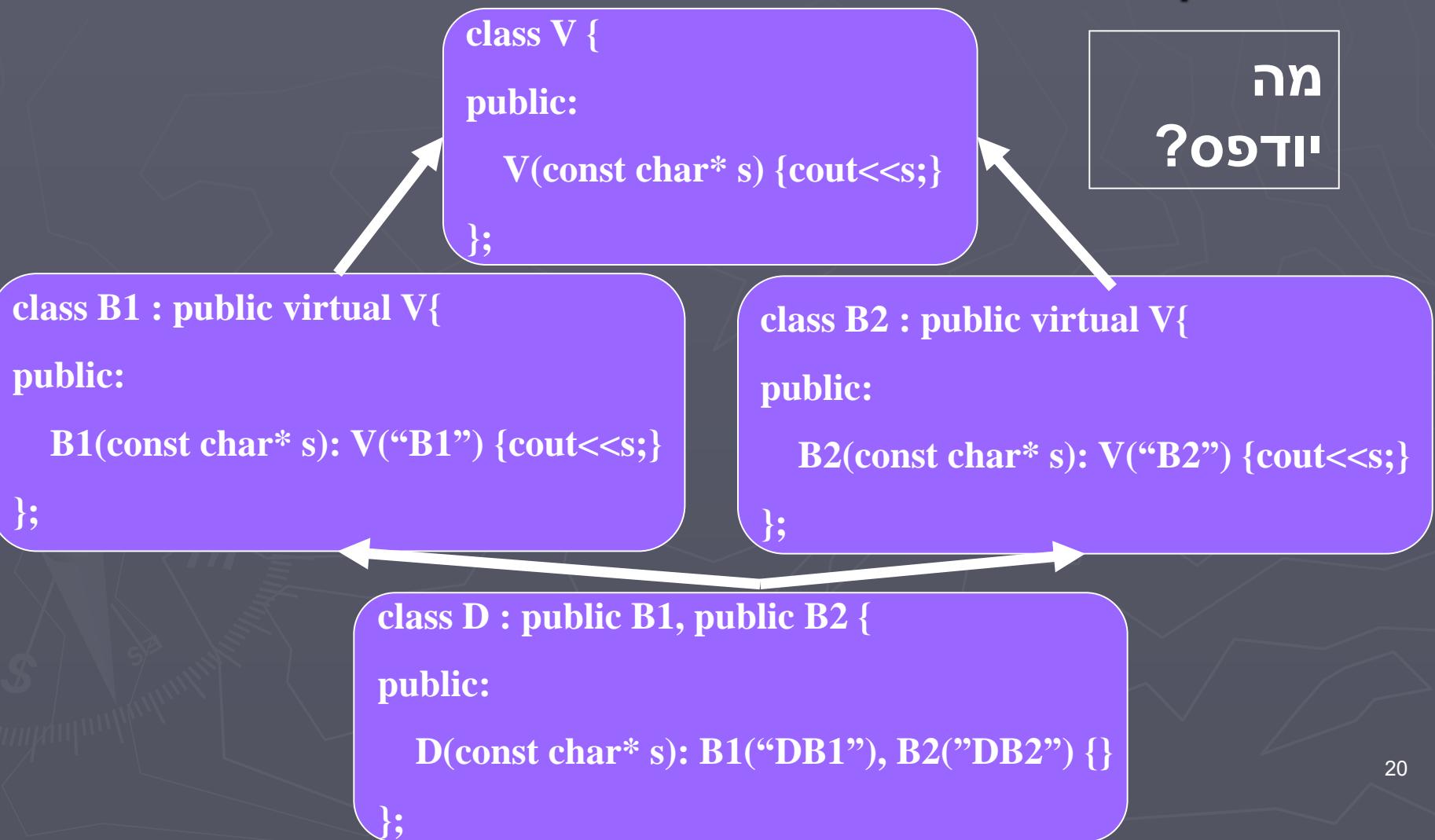
Virtual Inheritance

- ▶ **Ancestor A;**
A: **Ancestor**
(members)
- ▶ **Base1 B1; Base2 B2;**
B1: **near ptr**
Base1
(additional)
- ▶ **Derived D;**
D:
near ptr
Base1 (additional)
near ptr
Base2 (additional)
Derived (additional)

The diagram illustrates virtual inheritance. It features two blue boxes at the top: "Base1" (with "near ptr" and "Base1 (additional)") and "Base2" (with "near ptr" and "Base2 (additional)"). Arrows point from each base box to a bottom purple box labeled "Derived (members)". To the left of the base boxes is another purple box labeled "Ancestor (members)" which also points to the "Derived" box. Curved arrows from the "Base1" and "Base2" boxes point to their respective "Ancestor" boxes. Brackets on the right group "Base1" and "Base2" under the heading "Base1" and "Base2" respectively.

Constructors and Virtual Inheritance

❖ בהינתן המצב הבא:



❖ תשובה: שגיאת קומפילציה!
❖ למה?

- ❖ פתרון:
- ❖ לבנות בנאוי דיפולטי למחלקה 7.
- ❖ לקרוא לבנאוי של 7 יישורת בשורת האתחול של מחלקת C.
- ❖ אבות וירטואליים מאותחלים על ידי מחלקה הנגזרת הנמוכה ביזטר, ומכל שאר האתחולים – מתעלמים.
- ❖ מותר לאותל מבנאי של המחלקה כל אחד מהאבות הווירטואליים שלה, גם אם הם אבות קדמוניים ולא מיידים!
- ❖ מה לגבי אבות קדמוניים לא וירטואליים?

הורשה מרובה **vs** הורשה מרובה וירטואלית

- ❖ ישנים מקרים שבהם מחלוקת אחת היא בעצם שילוב של 2 מחלוקות קיימות (ולפעמים אף יותר).
- ❖ הביעתיות של הורשה מרובה מתחילה כאשר יש לנו אב קדמון משותף.
- ❖ ישנים מקרים שבהם אנחנו רוצים **מופע אחד** של אב קדמון, וישנים מקרים שבהם אנחנו רוצים **מופעים** של אב קדמון.

הורשה מרובה - דוגמא

- ❖ נחשוב על חברה שמייצרת מוצריים עם מסכי מגע.
- ❖ יש לנו מחלקה בשם Touch המהממת מכשיר מגע והיא מכילה אובייקט מסווג Screen.
(נניח שהמחלקה Screen כבר קיימת לנו ושייש לה פונקציונליות שמאפשרת לנו להציג למסך ולקבל ממנו מידע.)

מחלקה זו מקושרת ישירות אל המסר הפיזי של המכשיר ומסוגלת לבצע בו פעולות שונות.)

```
class Touch {  
protected:  
    Screen screen;  
    //...  
};
```

הורשה מרובה - דוגמא

- ❖ בשלב הראשון החלטנו ליצור נגנים עם מסך מגע.
- ❖ נשים לב שיש 2 סוגי של נגנים:
 - ❖ נגני מוזיקה
 - ❖ נגני ווידאו
- ❖ עם זאת, ישנו דברים שימושתיים ל-2 הנגנים שלנו...
 - ❖ בשניהם:
 - ❖ תהיה לנו רשימה רפרנסים לקבצים (File שכבר ממומשת אצלנו)
 - ❖ המתודה ()`getNext()` תחזיר רפרנס לקובץ הבא בראשימה.
 - ❖ המתודה ()`getPrev()` תחזיר רפרנס לקובץ הקודם בראשימה.
 - ❖ המתודה ()`play()` ו-()`stop()` שיפעלו ויפסיקו את הקבצים השונים.
 - ❖ יש להחזיק אינדקס לקובץ הנוכחי שהנגן מפעיל.
 - ❖ (ישן מתודות ומשתנים מחלוקת נוספים).

הורשה מרובה - דוגמא

- ❖ ניתן לראות שבמקרה שלנו יש תכונות שימושיות ל-2 האובייקטים השונים שאנו רוצים ליצור – מקרה קלאי של שימוש בהורשה.

```
class Player : public Touch {  
    int currFileLocation;  
public:  
    File& getNext();  
    File& getPrev();  
    void play();  
    void stop();  
};
```

- ❖ נזכיר כי אנחנו מייצרים נגנים עם מסך מגע – לכן המחלקה Player תירש מהמחלקה Touch
- ❖ 2 הנגנים השונים (הוידיואו והמוזיקה) יירשו ממהמחלקה Player, כאשר השימוש של פונקציית play() יהיה שונה (כי ניגון של קובץ מוסיקה שונה מניגון של קובץ ווידיואו (פע忸חים שונים וכו')).

הורשה מרובה - דוגמא

```
class Player : public Touch {  
    int currFileLocation;  
public:  
    File& getNext();  
    File& getPrev();  
    void play();  
    void stop();  
};
```

```
class VideoPlayer : public  
Player {  
public:  
    void play();  
};
```

```
class MusicPlayer : public  
Player {  
public:  
    void play();  
};
```

- ❖ בשלב זה החברה החליטה להיכנס לתחום הטלפונים הסולריים.
- ❖ מכיוון שיש לנו כבר אובייקט Touch מוכן (אנו יודעים לייצר מכשירי מגע), אנחנו מעוניינים להיכנס לשירות לתחום של טלפונים סולריים מבוססי מסך מגע.

הורשה מרובה - דוגמא

- ❖ אילו יכולות נרצה לטלורי שלנו?
- ❖ להזיא שיחות – ()
makeCall()
- ❖ לקבל שיחות – ()
answer()
- ❖ לשЛОח הודעות טקסט – ()
sendText()
- ❖ לצפות בהודעות טקסט – ()
readText()
- ❖ נדרש משתנים שיהוו את הזיכרון של המכשיר.
(ישן מethodים וממשתני מחלוקת נוספים)

```
class Phone: public Touch {  
    // data members...  
  
public:  
    void makeCall();  
    void answer();  
    void sendText();  
    void readText();  
};
```

הורשה מרובה - דוגמא

- ❖ כתה, נסתכל על Smartphone כלשהו שנרצה ליצור
- ❖ הוא גם מכשיר סלולרי, גם נגן מוזיקה וגם נגן ווידיאו
- ❖ לכן, המחלקה Smartphone תירש מ-3 המחלקות הללו

```
class Smartphone: public Phone, public VideoPlayer, public  
MusicPlayer {
```

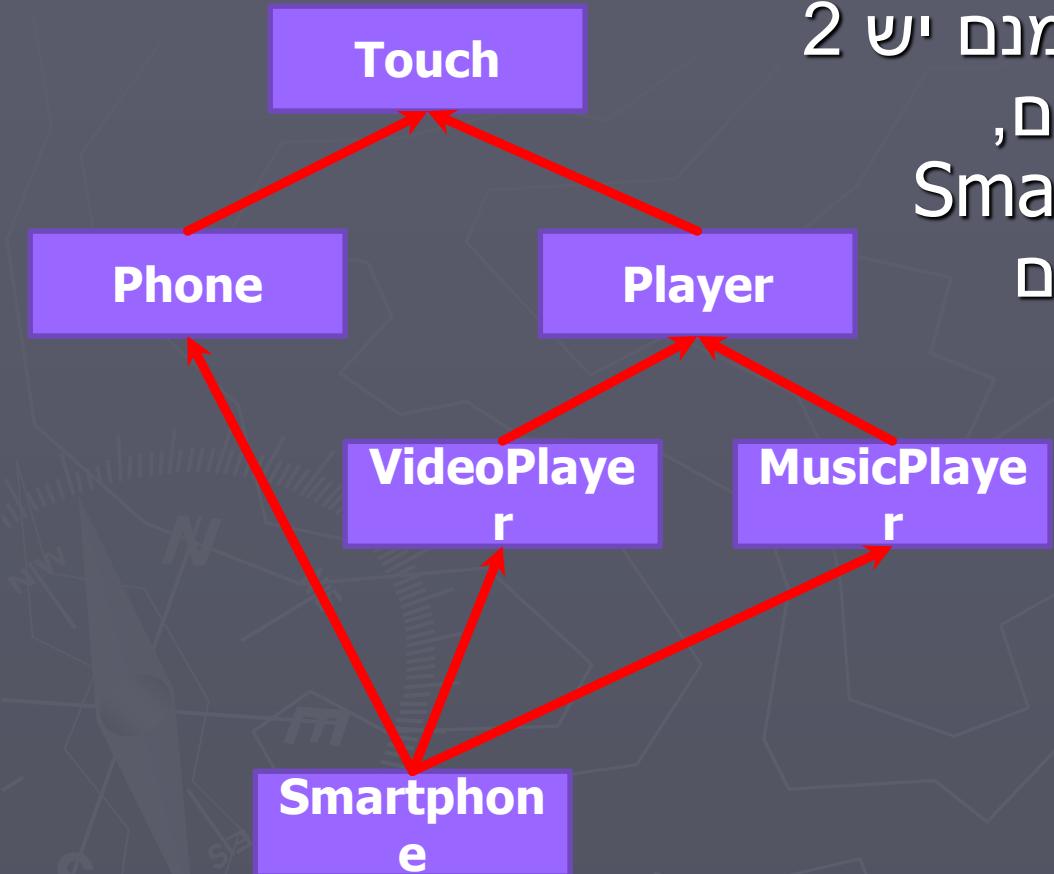
```
    // some data members and methods...
```

```
public:  
    homeButtonPressed();
```

```
};
```

- ❖ אבל, יש לנו בעיה בימוש זהה....

הורשה מרובה - דוגמא

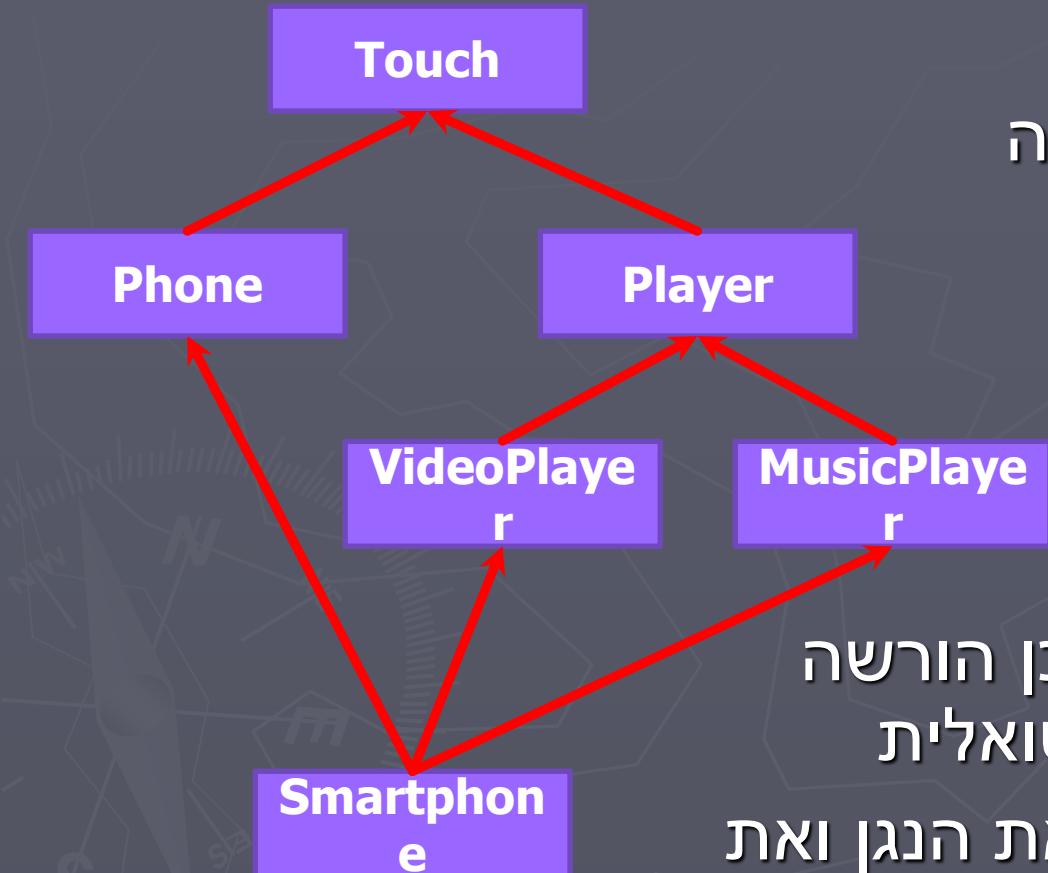


❖ ל-Smartphone שלו אמנים יש 2 אובייקטים שונים של נגנים, זהה בסדר כי ה-Smartphone מכיל באמת 2 נגנים שונים

❖ אבל, יש לו גם 3 (!) אובייקטים שונים של המחלוקת Screen. כי הוא מכיל 3 מופעים של המחלוקת Touch

❖ כיצד נמנע את זה?

הורשה מרובה - דוגמא



- ❖ הפיתרון – ההורשה של Phone ו-Player תהיה הורשה ווירטואלית
- ❖ למה רק שני אלה?
- ❖ משומש כל מכשיר VideoPlayer ו-MusicPlayer מכיל בתוכו 2 נגנים שונים, ולכן ההורשה זו לא אמורה להיות ווירטואלית
- ❖ לעומת זאת, מי שיורש את הנגן ואת הטלפון אינו מכיל 2 מסכים אלא אחד בלבד

```

class Touch {
    protected:
        Screen screen;
};

class Player : virtual public Touch {
    int currFileLocation;
public:
    File& getNext();
    File& getPrev();
    void play();
    void stop();
};

class VideoPlayer : public Player
{
public:
    void play();
};

class MusicPlayer : public Player
{
public:
    void play();
};

```

הורשה מרובה - דוגמא

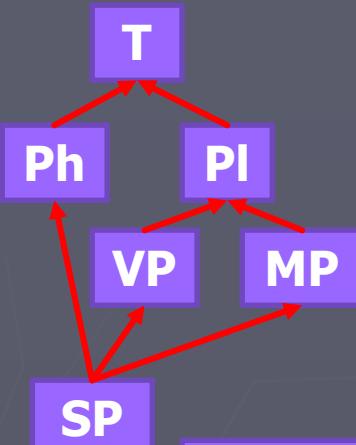
```

class Phone: virtual public Touch {
    // some data members...
public:
    void makeCall();
    void answer();
    void sendText();
    void readText();
};

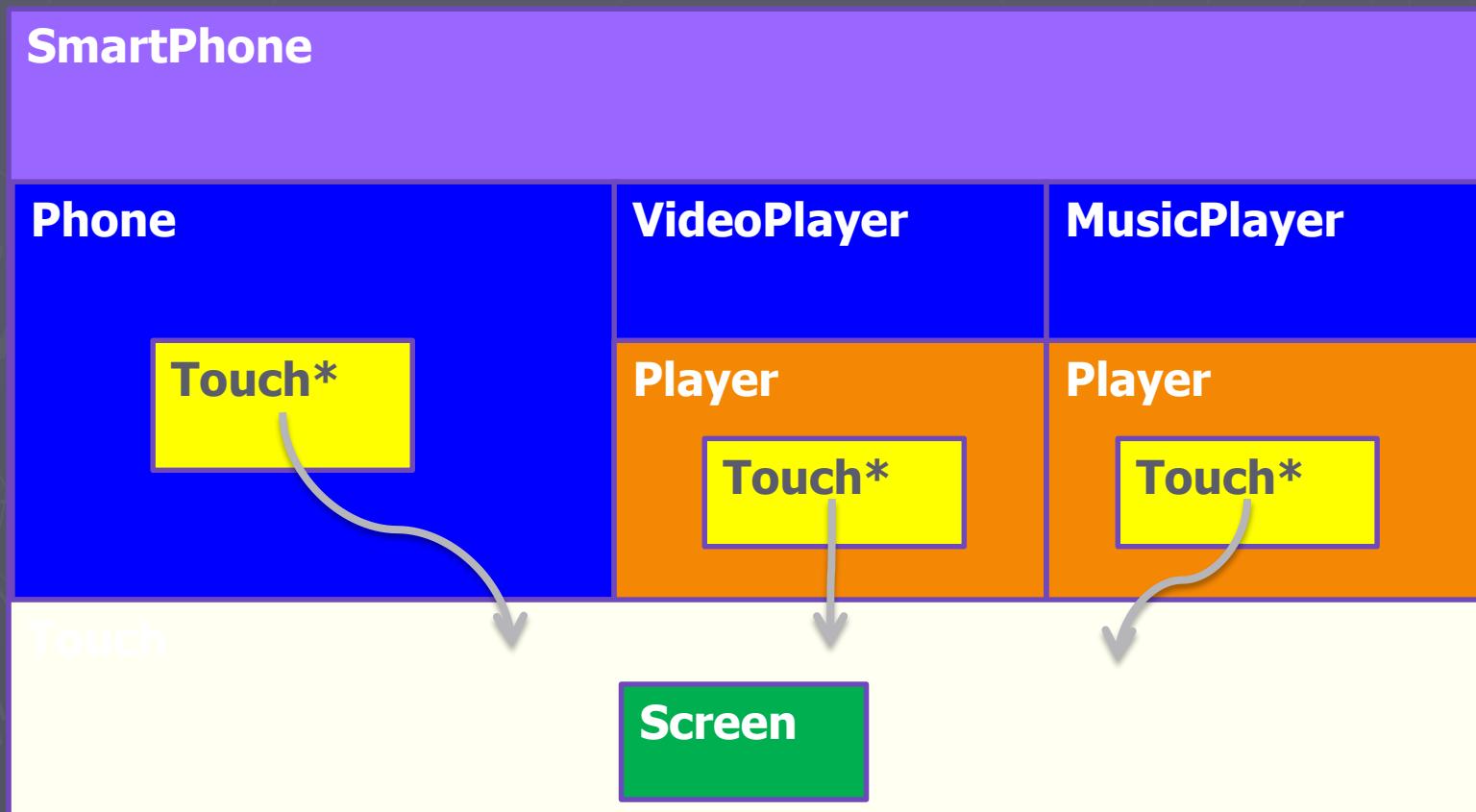
class Smartphone: public Phone,
public VideoPlayer, public
MusicPlayer {
    // some data members
    // and methods...
public:
    homeButtonPressed();
};

```

הירושה מרובה - דוגמא



כיצד יראה האובייקט ?SPhone



הורשה מרובה – דוגמת קוד

```
class A{  
public:  
    int a;  
    A() {cout<<"Ctor A"<<endl;}  
    ~A() {cout<<"Dtor A"<<endl;}  
};  
  
class B : virtual public A {  
public:  
    int b;  
    B() {cout<<"Ctor B"<<endl;}  
    ~B() {cout<<"Dtor B"<<endl;}  
};  
  
class C : public B {  
public:  
    int c;  
    C() {cout<<"Ctor C"<<endl;}  
    ~C() {cout<<"Dtor C"<<endl;}  
};  
  
class D : public B {  
public:  
    int d;  
    D() {cout<<"Ctor D"<<endl;}  
    ~D() {cout<<"Dtor D"<<endl;}  
};  
  
class E : public C, public D {  
public:  
    int e;  
    E() {cout<<"Ctor E"<<endl;}  
    ~E() {cout<<"Dtor E"<<endl;}  
};
```

```
int main ()  
{  
    E e;  
}
```

Output:

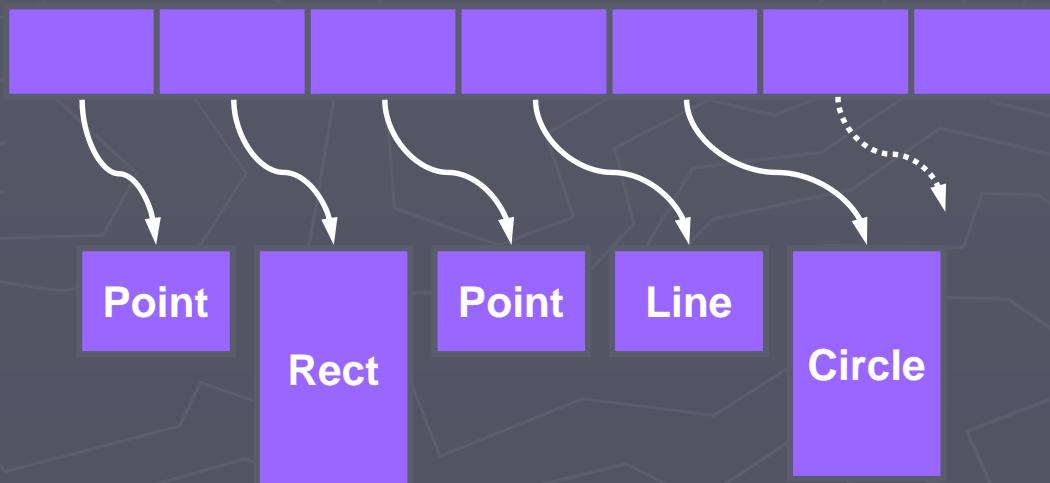
Ctor A
Ctor B
Ctor C
Ctor B // **A is constructed only once**
Ctor D
Ctor E
Dtor E
Dtor D
Dtor B
Dtor C
Dtor B
Dtor A

Polymorphism

הקדמה

- ❖ כפי שציינו אחד הגורמים החזקים בהורשה היא היכולת לשולח את אובייקט הבן למקום שמצוים לאובייקט האב.
- ❖ מצביע (או רפנס) מסוג האב יכולים להצביע לאובייקט מסוג הבן (**upcasting**).
לא יהיה אובדן מידע!
- ❖ לדוגמה:

Shape* items[7];



בתכנות מונחה עצמים, ההוראה:

"סובב א מעЛОת בכיוון סיבוב השעון"

צריכה להבצע בזורה שונה על ידי אובייקטים מסווגים
שונים (מחלקות שונות):



Figure = Do nothing if $n=360k$,
otherwise: rotate...



Circle = Do nothing.



Square = Do nothing if $n=90k$,
otherwise: rotate...

Introduction cont...

- ❖ מנגנון הפלימורפיזם מאפשר לנו לבצע את אותה פעולה על אובייקטים שונים ולדאוג לזה שהיא באמת **تبוצע אחרת על ידי כל אובייקט**, בדומה שהיא שקופה לתוכנת.

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

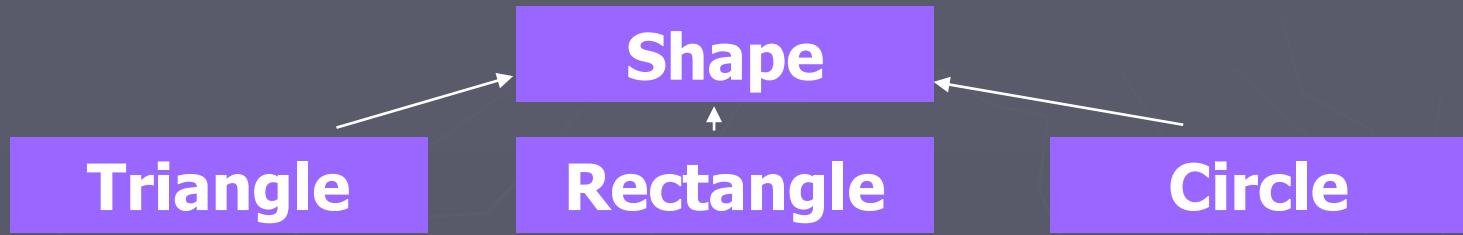
class B : public A
{
    void Do2();
    void Do3();
};
```

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

    A* pa = new B;
    pa->Do1();   V
    pa->Do2();   V??
    pa->Do3();   X
}
```

❖ אנו נוכל להפעיל
METHODS על האובייקט
רק במידה והן מופיעות
בממשק של מחלקה
הבסיסי!

- ❖ הנקודה החשובה והמשמעות היא: **איזה METHOD תופעל (של מחלקה הבסיסי או של המחלקה הנגזרת)?**



```
int main() {  
    Shape* p = GetShape(); ←.....  
    p->Draw();  
    //...  
}
```

Choose randomly
which shape to send
back

For example:
Shape* p = new Circle();

זמן קישור - Binding Time

- ❖ חיבור בין קריאה לפונקציה לגוף הפונקציה שERICAה להבצע
– נקראת קישור (binding).
- ❖ הבעייה: בזמן קומpileציה הקומpileר מקשר את המethodות
לפי הסוג של המצביע! (תקשור מתודת Draw של Shape).
- ❖ הסיבה: בזמן קומpileציה – אין לקומpileר דרך לדעת
שהמצביע מסווג* Shape* יצביע בעצם למחלקה נגזרת. מידע
זה **זמן רק בזמן ריצה!**
- ❖ הפתרון: נדחה את הקישור בין הקריאה לגוף המетодה **זמן
הricao!**

קישור דינמי - Dynamic Binding

- ❖ כאשר הקישור נעשה לפני שהתוכנית רצתה (על ידי הקומפיילר והلينקר) זה נקרא קישור מוקדם (static binding) או קישור סטטי (binding).
❖ כאשר הקישור נעשה בזמן ריצה זה נקרא קישור מאוחר (dynamic binding) או קישור דינמי (late binding).
- ❖ המילה השמורה `virtual` אומرت לקומפיילר לא לבצע קישור מוקדם.
- ❖ במקום, הקומפיילר בונה את המנגנון הנדרש על מנת לאפשר קישור מאוחר (דינמי).
- ❖ אז כאשר קוראים למתודה של המחלקה הנגזרת דרך מצביע (או רפרנס) למחלקה האב – תופעל המתודה המתאימה (של הבן!).

מתודה וירטואלית

- ❖ מתודה המוגדרת וירטואלית **במחלקת הבסיס** היא אוטומטיבית וירטואלית בכל הצעאים (הפורמיים) של אותה מחלקה!
- ❖ כל המתודות של המחלקות הנגזרות שתואמות את החתימה של המתודה במחלקת הבסיס יקרו תור שימוש במנגנון הווירטואלי.
- ❖ שכתב (redefinition) של מתודה וירטואלית במחלקה הנגזרת נקראת overriding.
- ❖ רק מתודות יכולות להיות וירטואליות!
- ❖ קישור מאוחר קורה רק **במתודות וירטואליות**, ורק כאשר משתמשים בכתובות (מצבייע או רפרנס) של מחלקת בסיס בה המתודות האלו קיימות והוגדרו כוירטואליות (במחלקת הבסיס עצמה או באחד האבות הקדמוניים שלה).

Virtual functions cont..

- ❖ מетодה וירטואלית מאפשרת לתוכנה להגדיר מетодות במחלקה האב כך שכל מחלוקת בין יכולה לתת להם שימוש אחר (שכתוב). הקומפיילר וה-loader יודאו שה קישור הנכון מתבצע בין האובייקטים והקריאה למетодה.
- ❖ מетодות וירטואליות עוברות קישור דינמי בזמן ריצה, דבר המאפשר שימוש במערך של מצביעים מסווג האב שבעצם יכול להצביע לכל היצאים של מחלוקת האב והפעלה נcona של METHOD היצא!
- ❖ הבעיות היא קבועה: המצביע הווירטואלי (VPTR).

אין מוממש קישור דינמי בשפת C++?

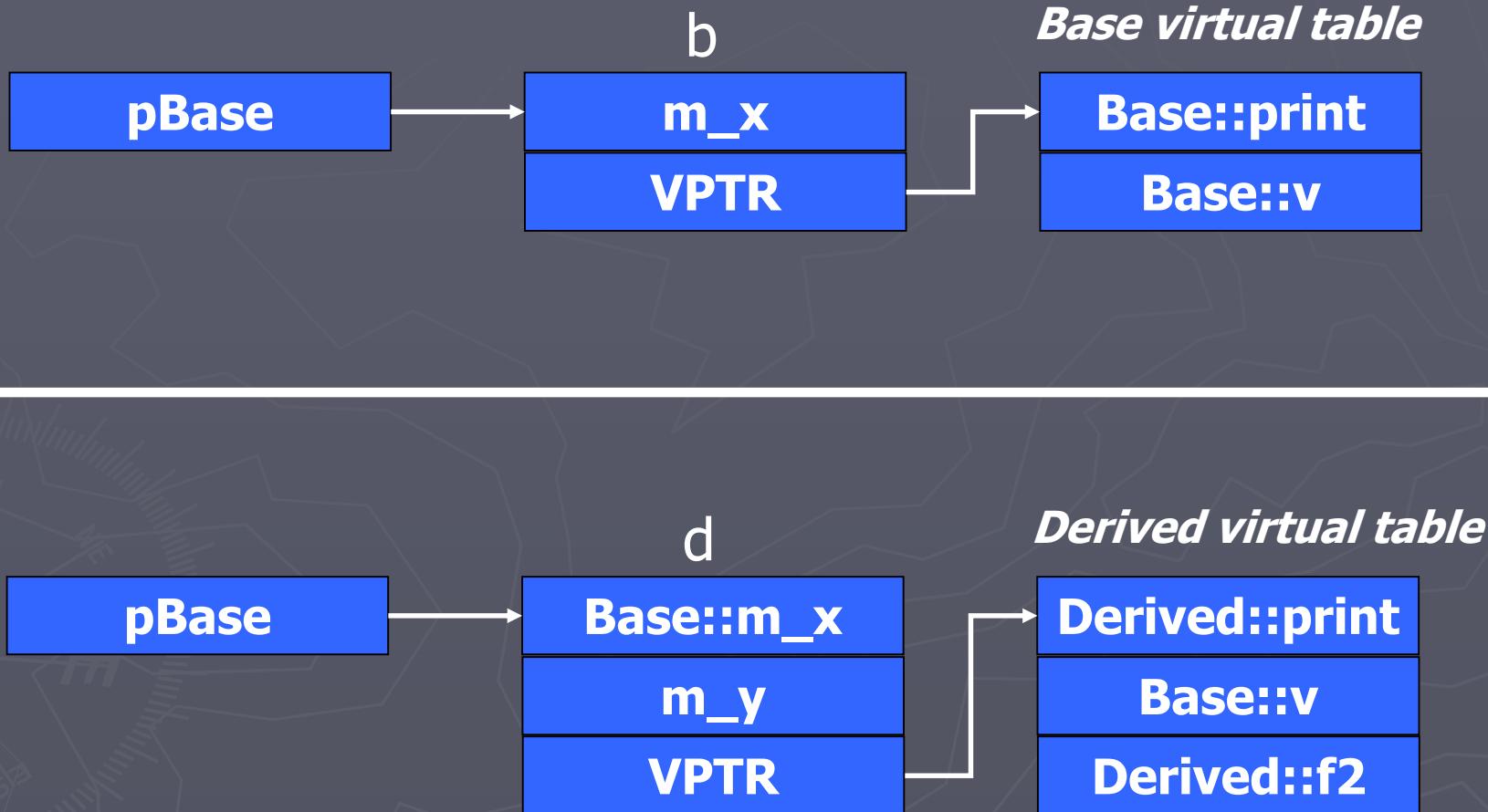
- ❖ עבור כל מחלקה שמכילה מתודה וירטואלית – הקומפיילר בונה טבלה, הנראית: הטבלה הווירטואלית (VTABLE).
- ❖ הטבלה הווירטואלית מכילה את הכתובות של המתודות הווירטואליות (ווק שלה!).
- ❖ לכל אובייקט מחלוקת עם מתודות וירטואליות, הקומפיילר מוסיף (בצורה שקופה למוכנות) מצביע שנקרא **VPTR** (VPTR) שתפקידו הוא להצביע ל-**VTABLE** של אותו אובייקט.
- ❖ כאשר נקרא בנאי (Ctor), אחד הדברים הראשונים שהוא עושה זה אתחול ה-**VPTR**. אך, הוא יודע רק לאתחל אותו להצביע לטליה (VTABLE) של המחלוקת מסווג הבנאי.

How does C++ implement late binding? cont...

```
class Base {  
    int m_x;  
public:  
    void f1();  
    virtual void print();  
    virtual void v();  
};  
  
class Derived : public Base {  
    int m_y;  
public:  
    virtual void f2();  
    void print();  
};
```

```
int main() {  
    Base b;  
    Derived d;  
    d.print();  
    Base* pBase = &b;  
    pBase->print();  
    pBase = &d;  
    pBase->print();  
  
    return 0;  
}
```

How does C++ implement late binding? cont...



How does C++ implement late binding? cont...

```
class NoVirtual {  
    int m_a;  
public:  
    void f() {}  
};
```

```
class YesVirtual {  
    int m_a;  
public:  
    virtual void f() {}  
};
```

```
int main() {  
    cout<<“int” : “<<sizeof(int)<<endl;  
    cout<<“NoVirtual” : “<<sizeof(NoVirtual)<<endl;  
    cout<<“YesVirtual” : “<<sizeof(YesVirtual)<<endl;  
  
    return 0;  
}
```

How does C++ implement late binding? cont...

- ❖ כאשר הקומpileר מעדכן את קוד הקריאה לבנאי (Ctor) הוא מוסיף קוד שקשרו לאוֹתָה מחלוקת עצמה(!) ולא למחוקות הבסיס שלה או לצאצאים שלה (חלוקת לא יודעת מי יורש אותה!). ולכן ה-VPTR שבו הוא משתמש **חייב להצביע על ה-VTABLE של אותהחלוקת!**
- ❖ מצביע VPTR מאותחל ל-VTABLE של אותו אובייקט במשר כל החיים של אותו אובייקט, אלא אם כן יש קראיות נוספות ל-Ctor לשם בנית אותו אובייקט.
- ❖ הטבלה עליה VPTR מצביע נקבעת על ידי הבנאי האחרון שנופעל על האובייקט.
- ❖ נחשו: מה קורה כאשר מפעלים מתודה ורטואלית מתוך בנאי? מדוע?
- ❖ ומה קורה ב-Dtor?

Inheritance and addition virtual functions

```
class Base {  
public:  
    virtual void f1();  
    virtual void f2();  
};  
  
class Derived : public Base {  
public:  
    virtual void f2();  
    virtual void f3();  
};
```

& Base::f1

& Base::f2

& Base::f1

& Derived::f2

& Derived::f3

❖ הקומpileר ממפה את המיקום של הכתובת () f1 בדיק לאותה נקודת גם בטבלה הוירטואלית של Base וגם בטבלה הוירטואלית של Derived.

❖ משומ שהקומPILEר עבד רק עם פוינטר למחוקת הבסיס – ניתן להפעיל רק מתחות שהוגדרו במחוקת הבסיס!

❖ הקומPILEר מגן علينا מפני קריאה לפונקציות וירטואליות שהוגדרו רק בבן (שגיאת קומPILEציה).

Redefinition במחלקה נגזרת

```
class A {  
    virtual void f(int) {...}  
};
```

```
class B : public A {  
    virtual void f() {...}  
};
```

- ❖ רשימה פרטטורים אחרות (hide) את המетодה המקורית של מחלקת הבסיס.
- ❖ במקרה של מתקנות מוגדרות שמן overloaded (אותו שם, אך לא חתימה זהה) ונתן רצים לשכטב אותן במחלקת הבן -> אנחנו מוכרים לשכטב את כלן! אלו שלא ישוכטבו בין - יוחבאו!
- ❖ (הערה: אם אין באמת צורך לשנות את המethodה של האב, ורק רצים למנוע הטעאה – ניתן שימוש של הבן פשוט לקרוא למmethodה של האב).

מדוע המנגנון של ווירטואל?

- ❖ אם מדובר במנגנון כל כך חשוב וחזק, שמאפשר לבצע את המתודה ה'נכונה' כל הזמן, **מדוע זה אופציונלי?** מדוע אנחנו בכלל מודעים למנגנון זהה ויוכלים לבחור האם להשתמש בו או לא?
- ❖ **יעילות:**
 - ❖ זמן ומקום
 - ❖ מethods Inline: למתודה ווירטואלית חיבת להיות כתובת – כי זה מה שנכתב ב-**VTABLE!**
- ❖ "If you don't use it, you don't pay for it" (Stroustrup).
- ❖ מספר שפות מונחות עצמים (Smalltalk, Java, Python) ועוד) אימצו את הגישה שקיים מאוחר כל כך קרייטי לתכנות מונחה עצמים שתמיד צריך להשתמש בו.

Upcasting vs Slicing

- ❖ כאשר יש לנו מצביע (או רפנס) מסווג אב שמצביע לצאצא –
זהו upcasting.
- ❖ מה קורה כאשר אנו שלוחים by value אובייקט מסווג ב-
למקום שמצופה לקבל אב?
 - ❖ Copy ctor! של מי?
 - ❖ של האב!
- ❖ נוצר אובייקט מסווג האב שזיהה בחלק האב לאובייקט מסווג הבן!
- ❖ !Slicing <=
- ❖ מה יקרה בהפעלת מתודה וירטואלית? מדוע?

שלושת התנאים לפולימורפיזם

1. Upcasting – מצביע או רפנס מסוג אב שמצביע לצאצא.
2. המטודה מוגדרת `C-virtual` בחלוקת הבסיס!
3. לצאצא יש מימוש אחר למטודה (שלו, או של אב שלו גם משמש כצאצא שלחלוקת הבסיס).

1 אונליין

```
class A{  
public:  
    virtual void f() {cout<<"fa"<<endl;}  
    void k()      {cout<<"ka"<<endl;}  
};  
  
class B : public A{  
public:  
    virtual void f() {cout<<"fb"<<endl;}  
    void k()      {cout<<"kb"<<endl;}  
};  
  
class C : public B{  
public:  
    virtual void f() {cout<<"fc"<<endl;}  
    virtual void k() {cout<<"kc"<<endl;}  
};
```

```
int main() {  
    C c;  
    c.f();    // fc (regular call)  
    c.k();    // kc (regular call)  
    A& a1 = c;  
    a1.f();   // fc (f virtual in A)  
    a1.k();   // ka (k not virtual in A)  
    B& b1 = c;  
    b1.f();   // fc (f virtual in A)  
    b1.k();   // kb (k not virtual in B)  
    B b2 = c;  
    b2.f();   // fb (slicing)  
    b2.k();   // kb (slicing)  
    A a2 = c;  
    a2.f();   // fa (slicing)  
    a2.k();   // ka (slicing)  
    A *a3 = &c;  
    a3->f(); // fc (f virtual in A)  
    a3->k(); // ka (k not virtual in A)  
    B *b3 = &c;  
    b3->f(); // fc (f virtual in B)  
    b3->k(); // kb (k not virtual in B)  
}
```

```

class A{
public:
    A() {
        cout<<"Ctor A"<<endl;
    }
    void f() {cout<<"af"<<endl;t();}
    void t() {cout<<"at"<<endl;}
    void k() {cout<<"ak"<<endl;}
    ~A() {cout<<"Dtor A"<<endl;}
};

class B : public A{
public:
    B() {
        cout<<"Ctor B"<<endl;
    }
    void k() {cout<<"bk"<<endl;}
    virtual void t() {cout<<"bt"<<endl;k();}
    virtual void m() {cout<<"bm"<<endl;f();}
    ~B() {cout<<"Dtor B"<<endl;}
};

```

```

class C : public B{
public:
    C() {
        cout<<"Ctor C"<<endl;
    }
    virtual void k() {cout<<"ck"<<endl;f();}
    ~C() {cout<<"Dtor C"<<endl;}
};

class E : public C{
public:
    E() {
        cout<<"Ctor E"<<endl;
    }
    virtual void f() {cout<<"ef"<<endl;t();}
    void t() {cout<<"et"<<endl;k();}
    ~E() {cout<<"Dtor E"<<endl;}
};

```

A:	B::VTBL
	B::t()
	B::m()

C::VTBL
B::t()
B::m()
C::k()

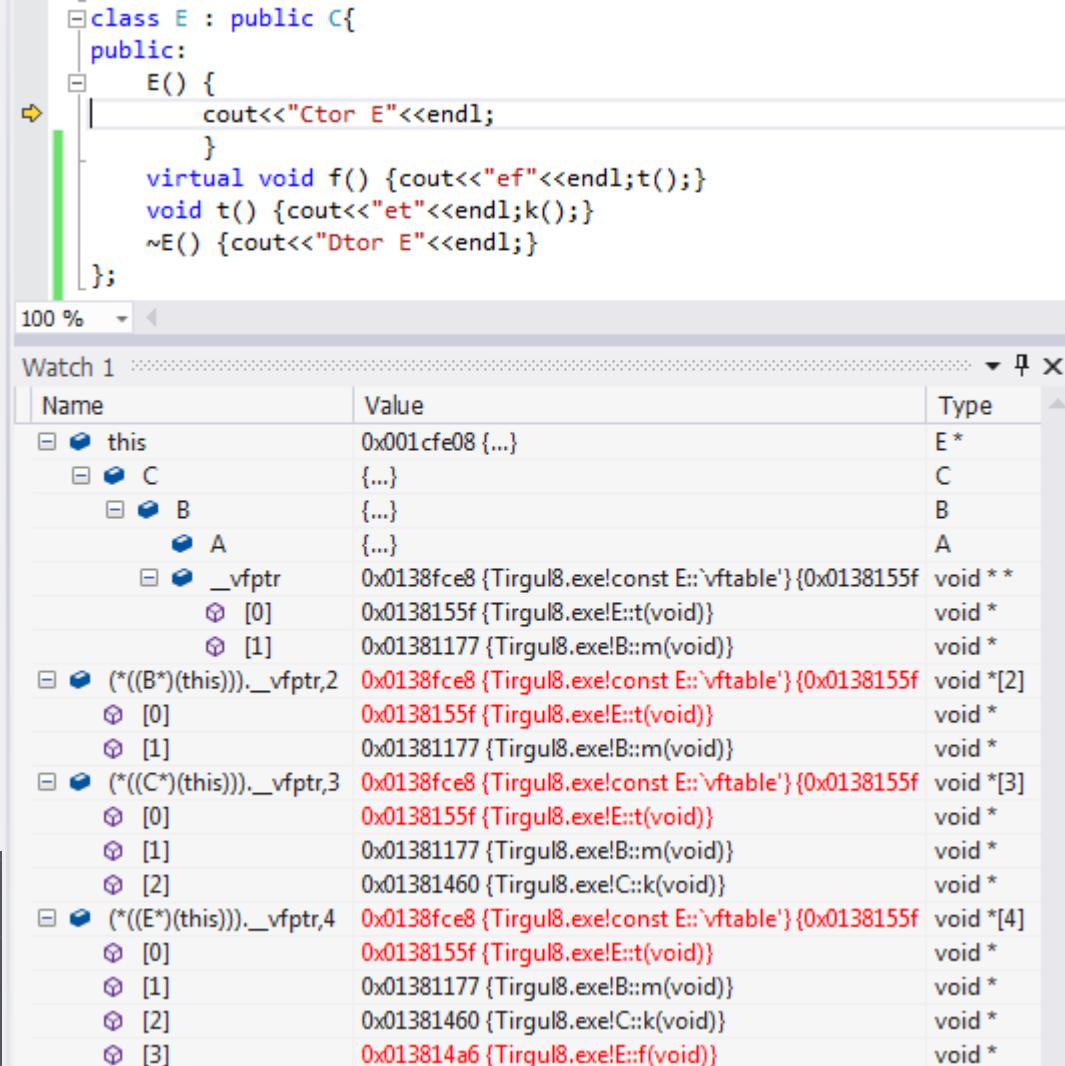
E::VTBL
E::t()
B::m()
C::k()
E::f()

2 אמצעית

```
int main(){  
    A a;  
    B b;  
    C c;  
    E e;  
}
```

Output:

Ctor A
Ctor A
Ctor B
Ctor A
Ctor B
Ctor C
Ctor A
Ctor B
Ctor C
Ctor E



The screenshot shows a debugger interface with two panes. The top pane displays the source code for a class hierarchy:

```
class E : public C{  
public:  
    E() { cout<<"Ctor E"<<endl; }  
    virtual void f() {cout<<"ef"<<endl;t();}  
    void t() {cout<<"et"<<endl;k();}  
    ~E() {cout<<"Dtor E"<<endl;}  
};
```

The bottom pane is a "Watch 1" window showing memory dump details for an object of type E. The object's memory layout is as follows:

Name	Type
this	E *
C	C
B	B
A	A
__vptr	void **
[0]	0x0138155f {Tirgul8.exe!E::t(void)}
[1]	0x01381177 {Tirgul8.exe!B::m(void)}
__vptr,2	void *[2]
[0]	0x0138155f {Tirgul8.exe!E::t(void)}
[1]	0x01381177 {Tirgul8.exe!B::m(void)}
__vptr,3	void *[3]
[0]	0x0138155f {Tirgul8.exe!E::t(void)}
[1]	0x01381177 {Tirgul8.exe!B::m(void)}
[2]	0x01381460 {Tirgul8.exe!C::k(void)}
__vptr,4	void *[4]
[0]	0x0138155f {Tirgul8.exe!E::t(void)}
[1]	0x01381177 {Tirgul8.exe!B::m(void)}
[2]	0x01381460 {Tirgul8.exe!C::k(void)}
[3]	0x013814a6 {Tirgul8.exe!E::f(void)}

המשר - 2 דוגמא

```
int main() {
```

```
    ...
```

```
    a.f();  
    a.k();
```

```
    b.m();
```

```
A* pa=&b;  
pa->t();
```

```
c.k();  
B* pb=&c;
```

```
pb->t();  
pb->k();
```

```
e.f();  
pb = &e;  
pb->m();  
pb->t();
```

```
...  
}
```

Output:

```
af  
at
```

```
ak
```

```
bm  
af  
at
```

```
at
```

```
ck  
af  
at
```

```
bt
```

```
bk
```

```
bk
```

```
ef
```

```
et
```

```
ck
```

```
af
```

```
at
```

```
bm
```

```
af
```

```
at
```

```
et
```

```
ck
```

```
af
```

```
at
```

Recall
funcs that call
other funcs:

A::f(){...t();}

B::t(){...k();}

B::m(){...f();}

C::k(){...f();}

E::f(){...t();}

E::t(){...k();}

B::VTBL

B::t()

B::m()

C::VTBL

B::t()

B::m()

C::k()

E::VTBL

E::t()

B::m()

C::k()

E::f()

ABSTRACT CLASSES

Polymorphism - Abstract Classes

- ❖ ישנים מקרים (במיוחד כשרוצים למשתמש מבני נתונים), שבהם אין לנו שום כוונה באמת לייצר אובייקט מחלוקת הבסיס.
- ❖ במקרים כאלה אנו מעוניינים:
- ❖ **למנוע מהמתכונת את האפשרות לייצר אובייקט מסווג מחלוקת הבסיס.**
- ❖ **להימנע מימוש חלק מהmethodות הוירטואליות של מחלוקת הבסיס** (הבנייה יכולים לשכטב).
- ❖ **להזכיר את המחלוקות הנגזרות לשכטב את methodות שלא מומשו בחלוקת הבסיס.**
- ❖ **פתרונות הוא methodות ווירטואליות טהורות (Functions) ומחלוקות אבסטרקטיות (Classes).**

Polymorphism - Abstract Classes cont

❖ **Pure Abstract Functions** (Classes):
❖ **Pure Abstract Methods** (Functions) וירטואליות טהורות (Methods) מוגדרות וירטואליות טהורות (Methods) ומחלקות אבסטרקטיות (Abstract Classes).

- ❖ מוגדרת טהורה היא מוגדרת וירטואלית שמצוירת במחלקה הבסיס אך ורק לשם תוספת למשך והכרחת היצאים שלה למשוך אותה.
- ❖ לשם הגדרתה – יש להוסיף `=prototype` להגדרה (constructor) של המוגדרת בקובץ ה-`H`.
- ❖ המוגדרת בדרכ"כ לא תוחמש (למעט `Dtor`, פרטיים בהמשך).
- ❖ מחלקה עם מוגדרות `virtual` pure (פחות אחת) היא אוטומטית – מחלקה אבסטרקטית! \rightarrow לא ניתן לייצר אובייקטים מסווג מחלקה זאת (שגיאת קומpileציה).

Abstract classes & Pure virtual methods

- ❖ אנו רוצים את מחלקת הבסיס רק בשביל ליצג ממשק (interface) משותף לכל עצאייה!
- ❖ למחלקת Employee יש שימושות וגם למחלקה הנגזרת ממנה Manager. לעומת זאת למחלקת Shape אין שם שימושות, אלא רק עצאים שלה!
- ❖ במצב זה (Shape) אנו לא רוצים שימושו באמת יצר אובייקט מסווג האב, אלא רק משתמש במצביעים (או רפנס) של המחלקה על מנת לעשות upcasting – כדי שייהי לנו אפשרות לכתוב קוד כללי שימושת בכל העצאים (דרך הממשק המשותף).
- ❖ ברגע שהגדכנו לפחות מетодה pure virtual אחת בתוך מחלקה – המחלקה אוטומטית נהפכת למחלקה אבסטרקטית (abstract class).
- ❖ ניתן לזהות מетодה pure virtual מטעמו שהוא מוגדרת כוירטואלית ולאחר החתימה שלה מופיע `=0`.

Abstract classes & Pure virtual functions cont...

- ❖ אי אפשר ליצור אובייקט ממחלקה אבסטרקטית! (שגיאת קומPILEZA!).
- ❖ שימוש לב שאי אפשר לשלוח אובייקט אבסטרקטי `by value!` למה? כאשר יורשים ממחלקה אבסטרקטית חיבים למש (implement) את כל המתודות הוירטואליות הטהורות, או שגם המחלקה הירושת תהיה ממחלקה אבסטרקטית.
- ❖ יוצרים ממחלקה אבסטרקטית כאשר אנו רוצים ליצור מספר מחלקות בעלות ממתק זהה ולתפעל את כולם דרך הממתק המשותף, אך למעשה המשותף אין משמעות בפני עצמו או שאין צורך למש אותו (מיושן מלא).

Abstract classes & Pure virtual functions cont...

Shape

```
virtual void Draw()=0;
```

```
virtual void Rotate(float)=0;
```

```
virtual void Enlarge(float)=0;
```

Rectangle

```
Draw() {...}
```

```
Rotate(float) {...}
```

```
Enlarge(float) {...}
```

Circle

```
Draw() {...}
```

```
Rotate(float) {...}
```

```
Enlarge(float) {...}
```

Virtual Destructor

- ❖ נניח שאנו רוצים למחוק אובייקט מחלקה נוצרת שהוקצה דינמית.
- ❖ אנו עשו זאת על ידי `delete` לצביע שצביע עליו.
- ❖ אם הצביע הוא מסוג מחלקת האב, הקומפילר, בזמן קומpileציה יכול לקשר אותו רק עם ה-`Dtor` של מחלקת האב.
- ❖ הבעיה: אנחנו רוצים לקרוא להורס של הבן, כדי שינקה הכל כמו צריך.
- ❖ **נשמע מוכר? זאת אותה בעיה שבשבילה השתמשנו בMETHODS וירטואליות!**
- ❖ **פתרון: נגדיר את ההורס כוירטואלי!**

האם ניתן למשוך פונקציה pure ?virtual

```
class A {  
public:  
    A() {cout << "A's CTOR\n";}  
    ~A() {cout << "A's DTOR\n";}  
    virtual void f() =0;  
};
```

```
void A::f() {cout << "A's f()\n";}
```

```
class B : public A {  
public:  
    B() {}  
    ~B() {}  
    void f() {cout << "B's f()\n";}  
};
```

```
int main() {  
    B b;  
    b.f();  
  
    b.A::f();  
    //...  
}
```

//output:
A's CTOR
B's f()
A's f()
A's DTOR

בעיה:

- ❖ נסתכל על המחלקה הבאה:

```
class Pet {  
public:  
    Pet() { //... }  
    ~Pet() { //... }  
    virtual void eat() { //default eat }  
    virtual void sleep() { //default sleep }  
    virtual void clean() { //default clean }  
    virtual void makeSound() { //default makeSound }  
    //...  
};
```

- ❖ נרצה שלא ניתן יהיה ליצור אובייקטים מסוג Pet אלא רק מסוג הילדיים שלו.

- ❖ ← נוצר פונקציה שתהיה pure virtual.
❖ איזו פונקציה זו תהיה?

פתרונות:

```
Pet(),  
~Pet()  
virtual void eat()  
virtual void sleep()  
virtual void clean()  
virtual void makeSound()
```

- ❖ איזו פונקציה זו תהיה?
- ❖ נמלה לא צריך להאכיל, דג לא ישן ולא משמע קול, וחזיר לא מנקים
(גם דג ונמלה לא ממש..)
- ❖ מה קורה אם אין פונקציה מתאימה להיות pure virtual (לכל פונקציה קיים בן שלא ימשח אותה)?
- ❖ ← ה-DTOR יהיה pure virtual
- ❖ בעיה - DTOR חייב מימוש...
- ❖ פתרון – מימושים את ה-DTOR למחרת שהוא pure virtual

חובה למשח DTOR שהוא pure virtual

Dynamic vs. static binding

❖ אנו משתמשים בקשרי סטטי כאשר:

❖ אנו בטוחים שכל תת-סוג שלנו (צאצא עתידי) ירצה להשתמש
בגרסתה שלנו לMETHOD ולא ירצה לשכט (override) אותה דינמית
(אלא רק redefine או hide).

❖ אנו משתמשים בכך או יוצרים "concrete data types" (פתרונות
פרטיאי מאוד לבעה שלא ניתן להשתמש או להרחיב למינים
אחרים).

❖ אנו משתמשים בקשר דינמי כאשר:

❖ יתכן כי לאחד מצאצאי המחלקה יהיהימוש אחר (יותר פונקציונלי
או יעיל) שעדיף לבחור בזמן ריצה.

❖ אנו בונים היררכיות הורשה אבסטרקטית (abstract data type).

Dynamic vs. static binding cont...

- ❖ **קישור סטטי יותר יעיל מקשרור דינמי.**
- ❖ **עלות overhead נמוכה יותר של זמן וזיכרון.**
- ❖ **אפשר שימוש ב-Inline.**
- ❖ **עשויה יותר בזמן קומpileציה ופחות בזמן ריצה.**

- ❖ **קישור דינמי הוא יותר גמיש מקשרור סטטי.**
- ❖ **אפשר למפתחים להרחיב את התנהלות המערכת בלי שהמשתמש ירגיש.**
- ❖ **אפשר הרחבות מערכת בלי לגעת בקוד של המערכת.**
- ❖ **הוספת תכונות חדשות (על ידי הוספה בן) בלי לשנות את הקוד הקיים.**

Polymorphism

General Containers

- ❖ **מכילים כללים** תומכים במבנה מבני נתונים שמכילים מצביעים – *
שיצבעו לאובייקטים מחלוקת שנגזרות ממחלוקת Data.
- ❖ **דוגמאות:** *Humus, MFC objects (CObArray), Graphic objects...*

Generic Algorithms

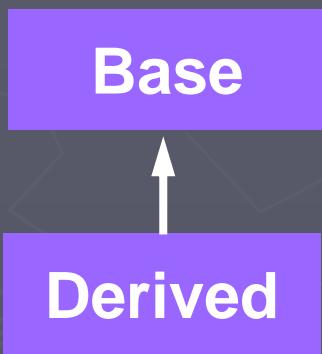
- ❖ **אלגוריתמים כללים** יוצרים שלד אלגוריתמי על ידי אלגוריתם שעבוד עם מצביע לחלוקת אב וקורא למתקודות שימושו אחרת אצל כל צאצא.
- ❖ **דוגמאות:** *Print, Draw Object, Calc3DVolume.*

Polymorphism - Additional notes

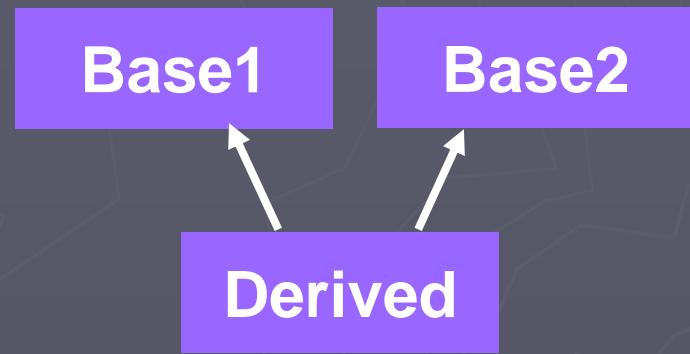
- ❖ התכונה הווירטואלית הוא מורשת! זאת אומרת שם מתודה הוגדרה כוירטואלית במחלקה בסיס היא תהיה וירטואלית **בכל היצאים שלה**. ועודין מומלץ לרשום את המילה **virtual** לפני הצהרות המתודות ביצאים (לשם קריאות).
- ❖ אם אובייקטים מכילים דברים לא טריויאליים או שיש חשש שהיצאים שלהם ייכלו דברים שכאלו – ההורס חייב **להיות וירטואלי!** (מדוע?)
- ❖ **כלל אבעע:** אם מדובר בהורשה וירטואלית, אז חובה ממש **Dtor** וירטואלי במחלקת הבסיס (אפילו ריק!).
- ❖ **בניי (Ctor)** לא יכול להיות וירטואלי!
 - ❖ **למה?**
- ❖ **ומה לגבי בניי העתקה (CCtor)?** איך ניתן להשיג את האפקט הנ"ל?

Inheritance types

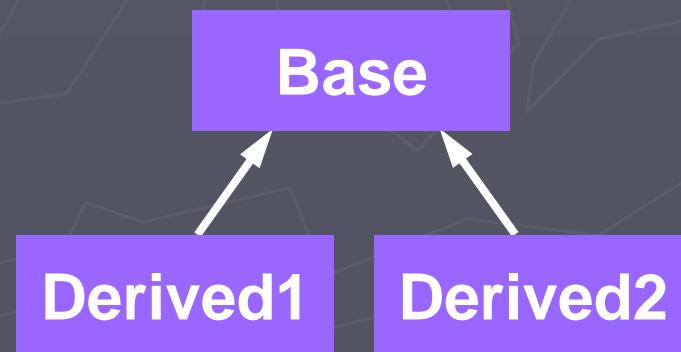
Simple Inheritance



Multiple Inheritance



Polymorphism



שאלות