

# תכנות מונחה עצמים בשפת C++

אוהד ברזילי

אוניברסיטת תל אביב

# תוצרי לוואי ומצב ממשתי ומופשט

המצגת מכילה קטעים מתוך מצגת של פרופ' עמירם יהודאי ע"פ הספר:

Object-Oriented Software Construction, 2nd edition,  
by Bertrand Meyer (Prentice Hall) .

כל הזכויות שמורות למחברים

# תוצר לוואי (side effect)

■ תוצר לוואי הוא שינוי של תכונה של עצם בעקבות פעולה כלשהי. תוצרי לוואי נוצרים ע"י (הגדרה רקורסיבית):

– השמה לשדה  $x$  כגון:  $x=y$  ;

– הפעלה של רוטינה  $z$  על שדה  $x$ , כאשר  $z$  מייצר תוצרי לוואי:  $x.z()$  ;

– הפעלה מקומית של שרות  $s()$  המייצר תוצר לוואי

– העברה by reference של שדה  $x$  כארגומנט לפונקציה המשנה ארגומנט זה

# פקודות ושאלות (commands and queries)

- פקודות מבוצעות בשל תוצרי הלוואי שלהן
- לשאלות (המחזירות ערך) אסור שיהיו תוצרי לוואי כלשהם
- ומה עם פקודות המחזירות ערך?  
– יש לפרק אותן לשתי פעולות נפרדות

`Max = ++i;`



`++i;`

`Max = i;`

# שקיפות התייחסותית (Referential Transparency)

■ לביטוי  $e$  יש תכונת השקיפות ההתייחסותית אם ניתן להחליף כל תת-ביטוי של  $e$  בערכו, ללא שינוי ערכו של  $e$

■ בשפת C לביטויים רבים אין תכונה זו:

–  $getint () \neq getint ()$

–  $2 * getint () \neq getint () + getint ()$

# שקיפות התייחסותית (Referential Transparency)

■ בשפת Eiffel הפרידו את קריאת הקלט והחזרת ערכו (הקוד הבא "תורגם" לתחביר C++):

*FILE input;*

*input . Advance();*

*input . last\_integer == input . last\_integer;*

*2 \* input . last\_integer ==*

*input . last\_integer + input . last\_integer;*

# תוצרי לוואי לגיטימיים

- המצב הממשי משתנה ללא שינוי המצב המופשט (ערך של שדה משתנה ללא השפעה על ערכן של השאילתות הציבוריות)
- דוגמא: מימוש מסוים של המתודה  $\max$  אולי משנה את המצב זמנית (ריצה עם האיטרטור על המערך) אבל מחזירה בסוף פעולתה את המצב הקודם
- דוגמא: המחלקה `COMPLEX`. שתי אפשרויות ייצוג קארטזית או קוטבית. לכל אחד מהייצוגים יתרונות שונים. רעיון: נשתמש בשני הייצוגים במקביל וניצור שאילתות עם תנאי לוואי לגיטימיים

# COMPLEX

■ עוד לפני המימוש - בעיה - נרצה ליצור בבנאי מספרים בייצוג קוטבי או קרטזי לפי הצורך ואולם:

```
class COMPLEX {
public:
    COMPLEX(float x, float y); // Rectangular coordinates
    COMPLEX(float r, float a); // Polar coordinates
                                // (radius and angle)
    // ERROR: Overload is Ambiguous:
    // COMPLEX::COMPLEX(float,float)
};

int main()
{
    COMPLEX c = COMPLEX(5.7, 1.2); // Ambiguous:
                                // Which coordinate system?
    ...
}
```



# COMPLEX

פתרון: "בנאים עם שם" ■

```
class COMPLEX {
public:
    static COMPLEX rectangular(float x, float y);    // Rectangular coord's
    static COMPLEX polar(float radius, float angle); // Polar coordinates
    // These static methods are the so-called "named constructors"
    ...
private:
    COMPLEX(float x, float y);    // Rectangular coordinates
    float x_, y_;
};

inline COMPLEX::COMPLEX(float x, float y)
    : x_(x), y_(y) { }

inline COMPLEX COMPLEX::rectangular(float x, float y)
{ return COMPLEX(x, y); }

inline COMPLEX COMPLEX::polar(float radius, float angle)
{ return COMPLEX(radius*cos(angle), radius*sin(angle)); }
```

# COMPLEX

■ עכשיו קל ליצור מספרים מרוכבים בשני הייצוגים:

```
int main()
{
    COMPLEX c1 = COMPLEX::rectangular(5.7, 1.2);
    // Obviously rectangular

    COMPLEX c2 = COMPLEX::polar(5.7, 1.2);
    // Obviously polar
    ...
}
```

# לרקוד על שתי החתונות

■ בפתרון הנוכחי כל עצם של מספר מרוכב מיוצג  
**באחד מהייצוגים ולא נהנה מהיתרונות של הייצוג**  
האחר:

– פעולות חיבור\חיסור קל מאוד לבצע בייצוג קארטזי

– כפל\חילוק קל מאוד לבצע בייצוג קוטבי

■ נתחזק את שני הייצוגים במקביל



# COMPLEX

```
/**
 * @inv: cartesian or polar
 * @inv: polar => (0 <= private_theta && private_theta <= Two_pi)
 * @inv: cartesian implies private_x and private_y are meaningful
 * @inv: polar implies private_rho and private_theta are
 *         meaningful
 */
class COMPLEX
{
    bool cartesian, polar;
    float private_x, private_y, private_rho, private_theta;
public:
    float x();
    float y();
    float rho();
    float theta();
    ...
}; // class COMPLEX
```



# COMPLEX



```
/** Abscissa
 */
float COMPLEX::x()
{
    set_cartesian();
    return private_x;
}
```

- אותו דבר גם עבור  $y()$ ,  $\rho()$ , ו- $\theta()$
- נשים לב כי לשאילתות אלו יש תוצרי לוואי לגיטימיים

# COMPLEX

```
/** Add z to current
 * @post: x() == $prev(x()) + z.x()
 * @post: y() == $prev(y()) + z.y()
 * @post: cartesian
 */
void COMPLEX::add(COMPLEX z)
{
    set_cartesian();
    private_x = private_x + z.x();
    private_y = private_y + z.y();
    polar = false;
}
```

■ אותו דבר גם עבור `subtract()`

# COMPLEX

```
/** Divide z to current
 * @post: rho() == $prev(rho()) / z.rho()
 * @post: theta() == ($prev(theta()) - z.theta()) % 2_PI
 * @post: polar
 */
void COMPLEX::divide(COMPLEX z)
{
    set_polar();
    private_rho = private_rho / z.rho();
    private_theta = (private_theta - z.theta()) % TWO_PI;

    cartesian = false;
}
```

multiply() אותו דבר גם עבור ■

# COMPLEX

```
/** sum of current complex and z
 * @post: $result.x() == x() + z.x()
 * @post: $result.y() == y() + z.y()
 * @post: $result.cartesian
 */
```

```
COMPLEX COMPLEX::operator+(COMPLEX z)
{
    return COMPLEX::rectangular(x()+z.x(), y()+z.y());
}
```

- אותו דבר גם עבור " - "
- כנ"ל לגבי "\*" ו- "/" תוך שימוש במערכת הקוטבית



# COMPLEX

```
/** Make cartesian representation available
 * @post: cartesian
 */
void COMPLEX::set_cartesian()
{
    if (!cartesian)
    {
        assert(polar);

        private_x = private_rho * cos(private_theta);
        private_y = private_rho * sin(private_theta);
        cartesian = true;
    }
    assert(polar && cartesian);
}
```

# COMPLEX

```
/** Make polar representation available
 * @post: polar
 */
void COMPLEX::set_polar()
{
    if (!polar)
    {
        assert(cartesian);

        private_rho = sqrt(private_x * private_x +
                           private_y * private_y);

        private_theta = arctan(private_y/private_x);

        polar = true;
    }
    assert(polar && cartesian);
}
```

# תוצר לוואי מופשט, תוצר לוואי ממשי ו `const`

- לשאלות  $r()$ ,  $x()$  ודומותיהן היו תוצרי לוואי לגיטימיים (ממשיים שאינם מופשטים)
- נרצה לבטא זאת ב C++ ע"י שימוש ב `const`

```
float COMPLEX::x() const
{
    set_cartesian();
    return private_x;
}
```

- הדבר גורר שגיאת קומפילציה – `set_cartesian` לא הוגדרה כ `const` ולכן אסור לקרוא לה מתוך מתודה שהוגדרה `const`

# mutable

- כאשר שדה אינו משפיע על המצב המופשט של העצם שבו הוא נמצא ניתן להגדיר שדה זה כ- mutable (מילה שמורה ב C++)
- בדרך זו ניתן להגדיר מתודה שמשנה שדה זה כ- const
- דוגמא שכיחה היא תחזוקה של מטמון (cache) אשר משיקולי ביצועים שומר את תוצאות השאילתות האחרונות כדי להחזירן במקרה של שאילתה זחה

# mutable cache

```
class proxy {  
    mutable string cache;  
  
public:  
    ...  
    string get_new_string() const  
    {  
        cache = expensive_fetch();  
        return cache;  
    }  
    string get_last_fetched() const  
    {  
        return cache;  
    }  
    ...  
};
```



# ארגומנטים למתודה

- מספר הארגומנטים צריך להיות קטן
- נפריד בין `operatnds` ובין `options`
  - על `operand` המתודה פועלת
  - `option` מבטא מצב של הפעולה, בדר"כ קיימת ברירת מחדל
- על מתודה לא להכיל כלל `options` אלא להזין אותן בצורה נפרדת

# גודל של מחלקה

■ מה סופרים (שדות? מתודות? מתודות מיוצאות?)

■ כולל תכונות נורשות?

■ גישת רשימת מכולת – הוסיפי את התכונה אם

היא:

– רלוונטית להפשטה של המחלקה

– מתאימה לשאר התכונות במחלקה

– אינה עושה בדיוק משהו שמתודה אחרת כבר עושה

– שומרת על שמורת המחלקה