

עוד על העמסת אופרטורים

אוהד ברזילי
אוניברסיטת תל אביב

אינטואיטיבי

- העמסת אופרטורים נותנת למחלקות שהוגדרו על ידנו "look and feel" של טיפוסים סטנדרטים
- אופרטורים עושים פעולה טבעית, אינטואיטיבית,

תכופה

- לדוגמא:

```
CPoint p1(2,3), p2(1,6);
```

```
→ CPoint p3 = p1+p2;
```

```
p3.Print();
```

- אילו פעולות מתבצעות בשורה זו?
- 1. `operator+` בנאי עבור עצם זמני `(temp)`
- 2. בנאי העתקה (שמעתיק את `temp` ל `p3`)
- 3. מפרק (עבור `temp`)
- 4.

העברת נתונים

- פרמטרים גדולים יועברו by reference
- פרמטרים אשר הפונקציה או המתודה אינן משנות את ערכם יוגדרו const
- אם הערך המוחזר מקומי, הוא יועבר by value אפשר אפילו ע"י יצירת עצם זמני במשפט ה return
- כאשר העצם המוחזר הוא העצם המפעיל או אחד הארגומנטים, ההחזרה תעשה by reference
- פקודות אשר בדר"כ אינן מחזירות ערך, כאשר הן מוגדרות כאופרטורים, יחזירו הפנייה לעצם המפעיל (הדבר מאפשר הפעלה חוזרת של אופרטורים בצורה 'טבעית')

אופרטור החיבור

```
class CPoint
{
    int m_x, m_y;
public:
    CPoint(int ax, int ay):m_x(ax), m_y(ay){}
    CPoint operator+(const CPoint& p) const;
};

CPoint CPoint::operator+(const CPoint& p) const
{
    return CPoint(m_x + p.m_x, m_y + p.m_y);
}
```

פונקציות או מתודות

- לפעמים יש למתכנת בחירה בין 2 צורות מימוש של האופרטור: פונקציה גלובלית (בדור"כ friend של טיפוס הארגומנטים) או מתודה
- למשל $p1+p2$ עשוי להיות ממומש כ-
 - `p1.operator+(p2)`
 - או כ:
 - `operator+(p1,p2)`
- האופרטורים: `operator=`, `operator []`, `operator ()`, `operator->` חייבים להיות ממומשים כמתודות
- תזכורת: מחלקה אשר מכילה מצביעים תרצה לממש בעצמה את אופרטור ההשמה, להבטיח התנהגות נאותה

פונקציות ומתודות

```
class X {
public:
    void operator+(int);
    X(int);
};

void operator+(X ,X);
void operator+(X ,double);

void f(X a)
{
    a+1;      // a.operator+(1)
    1+a;     // ::operator+(X(1),a)
    a+1.0;   // ::operator+(a,1.0)
}
```

עקביות והתנהגות ברירת מחדל

- זוהי אחריות המתכנת לשמור על עקביות במימוש.
למשל ש ++a יהיה שקול ל: a+=1 שיהיה שקול ל
a=a+1
- האופרטורים היחידים בעלי משמעות עבור כל טיפוס הם אופרטור ההשמה(=), אופרטור הכתובת של(&) ואופרטור הביצוע הסדרתי (,)
- המשתמש רשאי לשנות התנהגות זו או לחסום אותה

חסימת התנהגות ברירת מחדל

```
class X {  
private :  
    void operator=(const X&);  
    void operator&();  
    void operator,(const X&);  
    // ...  
};  
  
void f(X a ,X b)  
{  
    a = b; // error: operator= private  
    &a; // error: operator& private  
    a ,b ; // error: operator, private  
}
```



מימוש + ע"י +=

```
class complex {
    double re , im;
public:
    // needs access to representation
    complex &operator+=(complex a );

    // ...
};
complex operator+(complex a , complex b)
{
    complex r = a;
    return r += b ; // access representation through +=
}
```

מימוש + ע"י +=

```
inline complex &complex::operator+=(complex a)
{
    re += a.re;
    im += a.im;
    return *this;
}
```



המרות טיפוסים

- הופעת הסימן '=' בעת הצהרה על משתנה הוא סוכר תחבירי לבנאי המתאים לדוגמא:

```
class complex {
    double re , im;
public:
    complex(double r) :re(r), im (0) { }
    // ...
};

complex b = 3 ;// same as complex b = complex(3);
complex x(2); // initialize x by 2. no temporary
```

המרות טיפוסים

- כאשר לא הוגדר אופרטור עבור טיפוס מסוים המהדר מחפש המרה לטיפוסים שעבורם כן הוגדר אופרטור

- לדוגמא:

```
bool operator==(complex ,complex);
void f(complex x , complex y)
{
    x == y; // means operator==(x,y)
    x == 3; // means operator==(x,complex(3))
    3 == y; // means operator==(complex(3),y)
}
```

אופרטור המרה

```
class Tiny {
    char v;
    void assign (int i)
    { if(i&~077) throw Bad_range (); v =i ; }

public:
    class Bad_range { };

    Tiny (int i) { assign (i);}

    Tiny& operator=(int i)
    { assign(i); return *this; }

    operator int () const
    { return v ; } // conversion to int function
};
```



אופרטור המרה

```
int main()
{
    Tiny c1 = 2;
    Tiny c2 = 62;
    Tiny c3 = c2 - c1; // c3 = 60
    Tiny c4 = c3; // no range check (not necessary)
    int i = c1 + c2; // i = 64
    c1 = c1 + c2; // range error: c1 can't be 64
    i = c3 - 64 ; // i = - 4
    c2 = c3 - 64 ; // range error: c2 can't be - 4
    c3 = c4 ; // no range check (not necessary)
}
```

אופרטור המרה או העמסת אופרטורים?

- או זה או זה אבל לא גם וגם, אחרת:

```
int operator+(Tiny ,Tiny);

void f(Tiny t , int i)
{
    t+i;    // error, ambiguous:
            // operator+(t,Tiny(i))
            // or
            // int(t)+i ?
}
```

המרה מפורשת

- מה עושה הקוד הבא?

```
// initialize z with complex(2)
complex z = 2 ;

// make s a string with int('a') elements
string s = 'a';
```

- כאשר הבנאי נקרא "מאחורי הקלעים" ועושה דברים לא אינטואיטיביים, נרצה להגביל את הקריאה לבנאי לזימונים מפורשים בלבד

- הדבר מושג ע"י המילה השמורה explicit

המרה מפורשת

```
class String {
    // ...
    explicit String(int n); // preallocate n bytes
    String (const char *p); // initial value is the Cstyle string p
};

String s1 = 'a'; // error: no implicit char->String
                // conversion
String s2(10); // ok: String with space for 10
                // characters
String s3 = String(10); // ok: String with space for 10
                        // characters
String s4 = "Brian"; // ok: s4 = String("Brian")

String s5("Fawlty");
```

המרה מפורשת

```
void f (String);

String g()
{
    f(10); // error: no implicit int->String conversion
    f(String(10));
    f("Arthur"); // ok: f(String("Arthur"))
    f(s1);
    String *p1 = new String("Eric");
    String *p2 = new String(10);

    return 10; //error: no implicit int->String conversion
}
```

אופרטור [] אינדקס לא מספרי

```
class Assoc {
    struct Pair {
        string name;
        double val;
        Pair(string n="", double v=0) : name(n), val(v) { }
    };

    vector<Pair> vec;
    Assoc(const Assoc &); // private to prevent copying
    Assoc& operator=(const Assoc &); // private to prevent
    // copying

public:
    Assoc() {}
    double& operator[](const string &);
    void print_all() const;
};
```

אופרטור [] אינדקס לא מספרי

```
/** search for s; return its value if found;
    otherwise, make a new Pair and return the default
    value 0
*/
double& Assoc::operator[](const string& s)
{
    for(vector<Pair>::const_iterator p = vec.begin();
        p!=vec.end(); ++p)
        if (s == p->name) return p->val;
    vec.push_back(Pair(s,0)); // initial value: 0
    return vec.back().val ; // return last element
}
```

אופרטור ההפעלה ()

- הקוד הבא מפעיל את הפונקציה `negate` על כל אחד מאברי המכלים:

```
void negate(complex& c) { c = -c; }

void f(vector<complex>& aa , list<complex>& ll)
{
    // negate all vector elements
    for_each(aa.begin(), aa.end(), negate);

    // negate all list elements
    for_each(ll.begin(), ll.end(), negate);
}
```

אופרטור ההפעלה ()

- בדומה הקוד הבא מוסיף את המספר המרוכב `complex(2,3)` לכל אחד מאברי המכלים:

```
void add23(complex& c) { c += complex(2,3); }

void f(vector<complex>& aa , list<complex>& ll)
{
    // negate all vector elements
    for_each(aa.begin(), aa.end(), add23);

    // negate all list elements
    for_each(ll.begin(), ll.end(), add23);
}
```

אופרטור ההפעלה ()

- נרצה להכליל רעיון זה, נרצה לבצע פעולה כללית על כל אברי המערך ע"י שימוש בעצם שימש את פעולת ההפעלה ע"י אופרטור (). הדבר מאפשר שימוש בעצם ממש כמו שם של פונקציה

```
class Add {
    complex val;
public:
    Add(complex c) { val = c ; } // save value
    Add(double r , double i) { val = complex(r ,i); }

    // add value to argument
    void operator()(complex& c) const { c += val; }
};

void f(vector<complex>& aa , list<complex>& ll , complex z)
{
    // negate all vector elements
    for_each(aa.begin(), aa.end(), Add(2,3));

    // negate all list elements
    for_each(ll.begin(), ll.end(), Add(z));
}
```

אופרטור החץ

- האופרטור אונרי

```
class Ptr {
    // ...
    X *operator->();
};

void f(Ptr p)
{
    p->m = 7 ; // (p.operator->())-> m = 7
}

void g(Ptr p) {
    X *q1 = p->; // syntax error
    X *q2 = p.operator->(); // ok
}
```



מצביעים חכמים ++

```
void f1(T a) // traditional use
{
    T v[200];
    T* p = &v[0];
    p--;
    *p = a; // Oops: 'p' out of range, uncaught
    ++p;
    *p = a; // ok
}
```

- נרצה להחליף את המצביע ל T במצביע חכם שידע לדווח על חריגה מגבולות המערך

מצביעים חכמים ++

```
class Ptr_to_T {
    // ...
};

void f2(T a) // checked
{
    T v[200];
    Ptr_to_T p(&v[0], v, 200);
    p--;
    *p = a; // runtime error: 'p' out of range
    ++p;
    *p = a; // ok
}
```



מצביעים חכמים ++

```
class Ptr_to_T {
    T* p;
    T* array;
    int size;
public:
    // bind to array v of size s, initial value p
    Ptr_to_T(T *p , T *v , int s);

    // bind to single object, initial value p
    Ptr_to_T(T * p);

    Ptr_to_T& operator++();           // prefix
    Ptr_to_T operator++(int);        // postfix
    Ptr_to_T& operator();             // prefix
    Ptr_to_T operator(int);          // postfix
    T& operator*();                  // prefix
};
```

תכנות מונחה עצמים בשפת C++
אוניברסיטת תל אביב

27

אופרטורים ב ++C

- שם של אופרטור יכול להיות רק אחד מהאופרטורים הסטנדרטים של ++C:

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new []	delete	delete []

- לא ניתן להעמיס את האופרטורים:
 - :: (אופרטור השיוך)
 - . (גישה לשדה)
 - * . (גישה לשדה דרך מצביע לפונקציה)

תכנות מונחה עצמים בשפת C++
אוניברסיטת תל אביב

28