תרגול מס' 5: Operator Overloading

friend1

לפעמים נרצה להשתמש בשדות פרטיים של מחלקה מתוך פונקציה חיצונית שהיא לא חברה במחלקה. לשם כך ניתן לעשות שימוש במילת המפתח friend.

כדי לאפשר לפונקציה לגשת לחלקים פרטיים של מחלקה מסויימת נכריז עליה כfriend בתוך המחלקה.

ניתן גם להכריז על מתודה של מחלקה אחרת כחברה בעזרת שמה המלא:

friend void OtherClass::method();

אין חשיבות איפה מכריזים על הfriend במחלקה(private or public).

לתוך המחלקה נכניס את ההצהרה המלאה על הפונקציה, שמתחילה במילת המפתח. משמעות ההצהרה היא שהפונקציה המדוברת יכולה להשתמש בכל השדות והפונקציות הפנימיים של המחלקה (private / protected).

דוגמא:

```
class C
{     private:
              int i;
      friend int foo1(const C& c);
};
int foo1(const C& c){
      return c.i;              //O.K. - friend function
}
int foo2(const C& c){
      return c.i;              //illegal - private member
}
```

בצורה דומה ניתן להגדיר גם מחלקה שלמה כ- friend. זה שקול להגדרת כל הפונקציות החברות במחלקה זו כ- friend.

```
class A
{
      void fooA(..);
      friend class B;
};
class B{
      void fooB(..);
}
```

במקרה הזה הפונקציה fooB תוכל לגשת לשדות פנימיים של A, אבל fooA לא יכולה לגשת לשדות פנימיים של B.

2 חפיפת אופרטורים

מבחינת C++ אופרטורים כמו + או * הם פונקציות רגילות וניתן להעמיס על אופרטורים בדומה לפונקציות רגילות עבור מחלקות.

כדי לתת למחלקות התנהגות דומה לטיפוסים הבסיסיים, שפת C++ מאפשרת להגדיר כמעט את כל סוגי האופרטורים למחלקות חדשות. למשל ניתן לכתוב:

```
1. Date d1(2,8,2001);
2. Date d2 = 4+d1; // create a new date
3. ++d1; // Increment the date by one day
4. d1+=8; // Increment the date by eight days
5. if (d1 == d2) cout <<"equal";
```

כדי לעשות זאת, יש קודם כל להבין כיצד C++ מתייחסת לאופרטורים בקוד.

ניתן להגדיר את האופרטורים כמתודות או כפונקציות חיצוניות, המשמעות היא שאם אופרטור המקבל n פרמטרים מוגדר **כמתודה** אז הפרמטר הראשון הוא this ויש להכריז על n-1 פרמטרים נוספים.

אם האופרטור מוגדר כפונקציה **חיצונית** יש להכריז על כל הn פרמטרים.

לחלק מהאופרטורים קיימת גרסה אונארית וגרסה בינארית, למשל -:

אם נגדיר את הגרסה הבינארית כמתודה - על הפונקציה לקבל שני פרמטרים, אך הראשון מביניהם הוא ה-this. ולכן חתימתה כמתודה תהיה:

```
Complex Complex::operator-(const Complex&) const;
```

אם היינו מגדירים את הפונקציה לא כמתודה, אז צריך לרשום במפורש את שני הפרמטרים ומתקבל:

```
Complex operator-(const Complex&, const Complex&);
```

ואילו עבור האופרטור האונארי, כמתודה מתקבל:

```
Complex Complex::operator-() const;
```

ועבור פונקציה מחוץ למחלקה יש לרשום את הפרמטר היחדי במפורש:

```
Complex operator-(const Complex&);
```

2.1 אופרטור כפונקציה חברה

- כאשר הקומפיילר מזהה אופרטור אונרי (המקבל פרמטר אחד. למשל: ++, כלשהו @): @object; – הוא מפרש זאת כקריאה לפונקציה:

```
object.operator@();
```

- כאשר הקומפיילר מזהה אופרטור בינארי (המקבל שני פרמטרים. למשל: +, &&) כלשהו @: object1 @ object2; – הוא מפרש זאת כקריאה לפונקציה:

```
object1.operator@(object2);
```

לדוגמא, הקוד בשורות 3 ו-4 קורא לפונקציות הבאות:

```
class Date { …
    Date& operator++() { advance(); return *this; }
    Date& Date::operator+=(const unsigned int& rhs){
        for (unsigned int i=0; i< rhs;i++)  ++(*this);
        return *this;
    }…
```

2.2 מימוש כפונקציה חיצונית

אפשרות נוספת לממש אופרטורים, היא כפונקציה חיצונית למחלקה. ניתן לממש חלק מהאופרטורים בצורה אחת וחלק בצורה השנייה.

- כאשר הקומפיילר מזהה אופרטור אונרי (המקבל פרמטר אחד. למשל: ++, כלשהו @): @object; – הוא מפרש זאת כקריאה לפונקציה:

```
operator@(object);
```

- כאשר הקומפיילר מזהה אופרטור בינארי (המקבל שני פרמטרים. למשל: +, &&) כלשהו @: object1 @ object2; – הוא מפרש זאת כקריאה לפונקציה:

```
operator@( object1, object2);
```

לדוגמא, הקוד בשורה 2 קורא לפונקציה הבאה:

```
Date operator+( int lhs, const Date& rhs){
```

```
    Date result(rhs);
    result+=lhs;
    return result;
}
```

את הפונקציה הזו היינו חייבים לממש מחוץ למחלקה, משום שהפרמטר הראשון שהיא מקבלת הוא int.

אופרטור חיצוני מתנהג כמו פונקציה רגילה, ולכן אין לו גישה לפונקציות/משתנים במחלקה שאינם public. כדי לאפשר גישה יש להכריז על האופרטור הנ"ל כ-friend.

2.3 מה לא ניתן לעשות עם אופרטורים

- להוסיף אופרטורים חדשים
- לשנות את טיפוסם (אונרי / בינארי)
- לשנות את העדיפות בין האופרטורים : c1+c2*c3 תמיד יבוצע כ-c1+(c2*c3)

Reference לעומת ערך חדש

נסתכל על שתי הפונקציות הבאות:

אופרטורים הכוללים השמה כמו =+ מאפשרים שרשור ולכן נחזיר reference

```
Date& Date::operator+=(const unsigned int& rhs){
    for (unsigned int i=0; i< rhs;i++)  ++(*this);
    return *this;
}
```

אופרטור + הוגדר בconst כדי שיהיה אפשר להפעילו על אובייקטים קבועים

```
Date Date::operator+(const unsigned int& rhs) const{
    Date result(*this);
    result+=rhs;
    return result;
}
Date d1;
```

השוו בין הפקודות d1+=3 ל d1+3.

הפקודה הראשונה משנה את הערך של d1. שינוי הערך הוא מהות הפונקציה, והחזרת הערך היא "תופעת לוואי" שניתן להשתמש בה. הערך שנחזיר יהיה הערך העצמי – משתמשים ב- *this ולא ב this משום שאנחנו מחזירים reference ולא מצביע. אם היינו מחזירים ערך חדש (במקום רפרנס) ההחזרה הייתה יוצרת copy-constructor מיותר.

הפונקציה השנייה לא משנה את הערך של d1. לא ניתן להחזיר reference למשתנה result משום שהוא משתנה פנימי. יש להחזיר ערך חדש.

בקבלת ערך – נעדיף בדרך כלל לקבל const reference על פני קבלת ערך חדש – כדי לחסוך copy constructor מיותר (לגבי int אין הבדל מהותי בין השניים, אבל במחלקות אחרות copy constructor עלול להיות מסובך). ה- const חשוב מאוד במקרה זה, בשביל שנוכל לבצע פקודה כמו d+3 (3 הוא קבוע).

כמו-כן, הפונקציה השניה הוגדרה כ- const function כדי שניתן יהיה לבצע אותה גם על קבועים מסוג Date.

3פלט ב- C++

הדפסה למסך ב- C++ נעשית בעזרת מחלקה בשם ostream ואופרטור >>.

מימוש האופרטור למחלקה Date יראה כך:

```
class Date
{…
    friend ostream& operator<<(ostream& r, const Date& d);
};

ostream& operator<<(ostream& ro, const Date& d){
   return ro << d.day_ << "/" << d.month_ << "/" << d.year_;
}
```

אם d1 הוא אובייקט של המחלקה Date, נוכל לכתוב:

```
cout << “Date: “ <<  d1 << endl;
```

את הביטוי הזה C++ מפרשת כך:

```
operator<<(operator<<(operator<<(cout, “Date: “), d1), endl);
```

operator<<(cout, “Date: “) מחזיר רפרנס cout
ואז הוא עובר כפרמטר ראשון ל<<operator הבא

הפונקציה הפנימית ביותר מדפיסה ל- cout (אובייקט מסוג ostream שקשור ל- stdout) את המחרוזת ומחזירה את cout. הפונקציה האמצעית מקבלת את cout מהפונקציה הפנימית, מדפיסה אליו את התאריך d1 וכך הלאה.

אופרטור ההדפסה תמיד יקבל כארגומנט ראשון ויחזיר ostream& (לא const ostream&, בגלל שההדפסה משנה אותו)

שימו לב שהגדרנו כאן אופרטור חיצוני שמשתמש בשדות פרטיים של Date. לשם כך היינו צריכים להגדיר את הפונקציה כ- friend.

3.1 הדפסה "וירטואלית"

אופרטור ההדפסה לא יכול להיות פונקציה חברה במחלקה שאנחנו רוצים להדפיס (מדוע?) כי אז הפרמטר הראשון למתודה היה this והוא צריך להיות ostream. מצד שני, יכול להיות שנרצה שההדפסה של מחלקה יורשת תהיה שונה מזו של מחלקת האם – כלומר שהדפסה תתנהג כמו פונקציה וירטואלית.

כדי לממש התנהגות כזו, נממש את אופרטור ההדפסה כך (בדוגמת Person מהתרגול הקודם):

```
ostream& operator<<(ostream& ro, const Person& p){
    p.print(ro);
    return ro;
}

virtual void Person::print(ostream& ro) const {
    ro << "ID: " << id_;
    if (name_) ro << " Name: " << name_;
}

virtual void Customer::print(ostream& ro) const {
    ro << "ID: " << id_;
    if (name_ && title_) ro << " Name: " << title_ << name_;
}
```

אם ()print תוגדר כpublic בPerson אז אין צורך להגדיר את אופרטור >> כfriend

הפונקציה print היא פונקציה וירטואלית חברה ב- Person. אם נגדיר פונקציה זו כ- public (ואין סיבה שלא לעשות כך), לא יהיה יותר צורך בהגדרת האופרטור כ- friend של Date. זהו יתרון נוסף של השיטה.

הערה

אם אופרטור ההדפסה יוגדר כך:

```
ostream& operator<<(ostream& ro, Person p)
```

תאבד ההתנהגות ה"וירטואלית" של האופרטור – ותמיד תקרא פונקציית ההדפסה של Person. מדוע?

מכיוון שלא מועבר Reference – אופרטור ההדפסה ייצור לעצמו עותק מקומי של המשתנה שקיבל. העותק המקומי הזה יהיה מטיפוס Person, ולא מאחד הטיפוסים היורשים (יופעל Copy Constructor של Person).

4המחלקה String

אחת המחלקות הנפוצות ביותר בשפת C++ הינה המחלקה String. מחלקה זו מספקת למשתמשיה גישה נוחה ופונקציונלית לפעולות טקסטואליות. להלן מימוש בסיסי ולא יעיל של מחלקה כזו. כל אובייקט במחלקה מחזיק עותק פרטי של המחרוזת אותה הוא מציג ומטפל בהקצאה דינאמית ושחרור שלה:

```
class String {
 private:
   char* m_sz;
   char* Duplicate(const char* szSource)  {
      char* sz = new char[strlen(szSource) + 1];
      strcpy(sz, szSource);
      return sz;
   }
 public:
   String(const char* sz = "") {                    // C'tors
         if (!sz) sz = "";
         m_sz = Duplicate(sz);
   }
   String(const String& rs) { m_sz = Duplicate(rs.m_sz); }
   String& operator=(const String& rs) {   // Assignment
         if (this != &rs) {
             delete [] m_sz;
             m_sz = Duplicate(rs.m_sz);
         }
        return *this;
```

```
    }
    ~String() { delete [] m_sz; }                    // D'tor
```

האופרטורים +, =+ בעלי משמעות של שרשור. האופרטור [ ] מחזיר את ערך התו מהמקום המתאים במחרוזת (באופן דומה למערך תווים). האופרטור >> כמובן משמש להדפסת המחרוזת.

```
    char operator[](int i) const { return m_sz[i]; }
    bool operator==(const String& rs) const
                                { return !strcmp(m_sz, rs.m_sz); }
    String operator+(const String& rs) const {
        String s(*this);
        return s += rs;
    }
    String& operator+=(const String& rs) {
        char* sz = new char[strlen(m_sz) +
                                strlen(rs.m_sz) + 1];
        sprintf(sz, "%s%s", m_sz, rs.m_sz);
        delete [] m_sz;
        m_sz = sz;
        return *this;
    }
    int GetLen() const { return strlen(m_sz); }

    friend ostream& operator<<(ostream& ro,
                        const String& rs);
};  // End of class String

ostream& operator<<(ostream& ro, const String& rs) {
    ro << rs.m_sz;
    return ro;
}
```

שימוש באופרטור =+ עבור המימוש של אופרטור +

בעזרת אופרטור חיבור שהגדרנו קודם, נוכל להגדיר גם חיבור עם int, שישרשר את המחרוזת המייצגת את המספר. את האופרטורים הללו אנחנו מגדירים מחוץ למחלקה, תוך שימוש באופרטורים שהמחלקה כבר הגדירה (אין גישה לשדות פרטיים):

נגדיר אופרטורים לחיבור מחרוזת ל- int בכל סדר:

אין צורך להכריז על האופרטורים הללו כfriend כי אנו משתמשים באופרטורים שהמחלקה כבר הגדירה

```
String operator+(int i, const String& rs) {
    char szNum[50]; // Should be enough...

    sprintf(szNum, "%d", i);
    String s(szNum);
    return s += rs;
}

String operator+(const String& rs, int i)  {
    char szNum[50]; // Should be enough...

    sprintf(szNum, "%d", i);
    return rs + String(szNum);
}
```

נביט על קוד לדוגמא שמשתמש במחלקה הזו:

```
1.    String s1;
2.    String s2("Radio");
3.    String s3("Blah");
4.    s1 = s2 + String(" ") + s3 + String(" ") + s3;
```

```
5.      cout << s1 << endl;
```

נעבור שורה-שורה על ביצוע התוכנה. נזכור שבסוף כל מחרוזת נרשם התו '\0'.

נזכר מה כל מתודה עשתה: (כדאי לכתוב את המתודות על הלוח)
constructor דיפולטי/שמקבל מצביע לתו- מקצה זיכרון.
copy constructor – מקצה זיכרון.
אופרטור השמה – מוחק זיכרון ישן ומעתיק את המידע החדש.
אופרטור + יוצר עותק לוקלי של המחרוזת המועברת כפרמטר הראשון, משתמש באופרטור += ולאחר מכן מחזיר עותק.
אופרטור =+ מקצה זיכרון חדש, מוחק את הישן, ומכניס את המידע החדש.

שורה 1) קונסטרקטור דיפולטי – מקצה תו אחד בזיכרון ('\0')

שורת 3-2) קונסטרקטורים המקצים בזיכרון מקום למחרוזות Radio ו- Blah.

שורה 4) כאשר אנחנו ניגשים לנתח שורה זו עלינו להבין קודם כל שאופרטור חיבור מתבצע בין שני ארגומנטים בלבד בכל פעם, על פי הסדר.

- תחילה מוקצים שני הערכים הזמניים שנוצרים ע"י String(" ")‏. נקרא להם ערך זמני 1 ו- 2. **<u>שתי הקצאות זיכרון</u>**, של שני תווים בכל פעם.
- נקרא אופרטור חיבור בין s2 לערך זמני 1:
  - אופרטור חיבור מתחיל ביצירת עותק מקומי של s2 למשתנה מקומי (s) – שזו **<u>הקצאת זיכרון</u>**.
  - אחר-כך נקרא אופרטור =+ על המשתנה המקומי s, ש**<u>מקצה זיכרון</u>** למחרוזת החדשה ("Radio ") ו**<u>משחרר זיכרון</u>** שהחזיק את המחרוזת הישנה ("Radio").
  - בסוף האופרטור + מבוצע return by value לערך שהחזיק s – כלומר מבוצע copy constructor לערך, שזו **הקצאת זיכרון** . העותק נוצר לתוך ערך זמני חדש שנקרא לו ערך זמני 3 (מחזיק את המחרוזת "Radio ")
  - לאחר ה- return , נקרא הדיסטרקטור של המשתנה הפרטי s, ש**משחרר זיכרון** שנתפס ע"י המשתנה.
  - סה"כ – שלוש הקצאות זיכרון – שתיים ששוחררו במהלך האופרטור ואחת נוספת לערך זמני 3.

```
s1 = s2 + String(" ") + s3 + String(" ") + s3;
```

- נקרא אופרטור חיבור בין ערך זמני 3 ("Radio ") ל s3.

- כמו בפעם הקודמת, אופרטור זה כולל מספר הקצאות ושחרורים של זיכרון, שבסופם ייווצר ערך זמני 4 שיכיל את המחרוזת "Radio Blah"

- חיבור בין ערך זמני 4 לערך זמני 2. נוצר ערך זמני 5, "Radio Blah "
- חיבור בין ערך זמני 5 ל s3. נוצר ערך זמני 6, "Radio Blah Blah"
- אופרטור השמה מערך זמני 6 לתוך s1
  - שחרור זיכרון של s1 (שהכיל מחרוזת ריקה) והקצאת זיכרון לתכולה החדשה שלו ("Radio Blah Blah")
- בתום השורה – ייקרא הדיסטרקטור של כל אחד מהערכים הזמניים 1-6, כל דיסטרקטור כזה **משחרר זיכרון**.
- בסיכומו של דבר – אין דליפות זיכרון בקוד. בשורה זו 15 הקצאות זיכרון ו- 15 שחרורי זיכרון.

שורה 5) אופרטור הדפסה עובד עם reference בלבד, ללא הקצאות או שחרור זכרון.

4.1 לקחים רלוונטיים

בעזרת כתיבה נכונה של קונסטרקטורים, דיסטרקטור ואופרטורי השמה ניתן ליצור מחלקה שמקצה ומשחררת מקום בזיכרון, מבלי שהמשתמש במחלקה יהיה מודע להקצאות אלה. ערכים זמניים נוצרים ומשוחררים לפי הצורך. תכונה זו נחשבת לחיובית מאוד ב- C++, משום שקל לטעות בשחרור והקצאת זיכרון, וקשה למצוא טעויות כאלו לאחר שנעשו.

שימוש באופרטור אחד עבור מימוש אופרטורים אחרים הוא נכון מבחינות רבות, אך לעיתים הוא יוצר מימוש לא יעיל. כיצד היה ניתן לממש את אופרטור + בצורה יעילה יותר?

בעזרת כתיבה של אופרטורים ניתן ליצור קטעי קוד פשוטים וקריאים מאוד שמחביאים מאחוריהם פונקציות פשוטות פחות. הסתרת קוד מתחת לאופרטורים היא "חרב פיפיות". למשל במקרה הזה, היה ניתן לשרשר את המחרוזות בצורה יעילה הרבה יותר אם היה נעשה שימוש רק באופרטור =+ ללא אופרטור חיבור (מדוע?) כי היינו חוסכים פעולות של הקצאת זיכרון

4.2 הערה אחרונה על יעילות

לעיתים, נעדיף קוד יעיל פחות אבל קריא יותר. רוב התוכנות מבלות כמעט את כל זמן החישוב שלהן בחלק קטן מהקוד שלהן. למשל בתוכנת דחיסה של קובץ נרצה שהדחיסה עצמה תיעשה על-ידי קוד יעיל ככל הניתן, אך אין כל חשיבות לשאלה כמה יעילות שגרות הממשק למשתמש. קוד קריא יותר יחסוך זמן פיתוח ומציאת באגים, ולכן יהיה עדיף. המשקל בקורס הוא חד-משמעית על קריאות הקוד.

מוסף: כל האופרטורים הניתנים לחפיפה ב-++C :

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