נושאים ב C++

מאת: אילן אמויאל



נושאים ב C++

(מהדורה דיגיטלית)

מאת אילן אמויאל

זכויות יוצרים

2010 © כל הזכויות שמורות לאילן אמויאל

הספר מוגש לציבור ללא תשלום, לשימוש אישי ולמטרות לימוד.

אין בהפצתו ללא תשלום משום ויתור על זכויות היוצרים. אין להעתיק, לשכפל, למכור או לעשות שימוש מסחרי בספר, בשלמותו או בחלקו, ללא אישור מראש ובכתב מאת בעל זכויות היוצרים.

הגבלת אחריות

המידע, ההסברים ודוגמאות הקוד בספר נועדו למטרות לימוד והעשרה בלבד.

המחבר וההוצאה עשו כמיטב יכולתם להבטיח את נכונות המידע המובא בספר, אולם אין הם מתחייבים לשלמותו, לדיוקו או להתאמתו למטרה מסוימת.

דוגמאות הקוד נכתבו ונבדקו בסביבת Visual C++ על מערכת ההפעלה Windows. ייתכנו הבדלים בהתנהגותן בסביבות פיתוח אחרות ובגרסאות שונות של כלי הפיתוח.

השימוש במידע ובדוגמאות הקוד נעשה באחריות הקורא בלבד. המחבר וההוצאה לא יישאו באחריות לנזק ישיר או עקיף העלול להיגרם כתוצאה מהשימוש בהם.

הוצאת מויאל

אתר ההוצאה: https://moyal.es

אתר "כיתה ברשת": https://kita.co.il

תוכן העניינים

הקדמה 7

שיעור 1: Constructors 9

1.1 Default Constructor (Parameterless Constructor) 9

1.2 Copy Constructor 10

1.2.1 Shallow Copying - העתקה רדודה 12

1.2.2 Deep Copying - העתקה עמוקה 14

1.2.3 Memberwise Copy - העתקת חברי מחלקה 15

1.2.4 מתי מופעל ה-Copy Constructor 15

1.3 Explicit Constructor 15

1.4 אתחול קבועים ב-Constructor 18

שיעור 2: Inheritance 21

2.1 הרשאות גישה 21

2.2 סוגי הירושה 21

2.2.1 ירושה פרטית 21

2.2.2 ירושה מוגנת 24

2.2.3 ירושה ציבורית 25

2.3 ירושה מרובה 27

2.3.1 פריסת הזיכרון 27

2.3.2 Name Ambiguity 31

2.4 ירושה וירטואלית 37

2.5 סדר בנייה ופירוק של אובייקטים 41

שיעור 3: Polymorphism 49

3.1 מבוא 49

3.2 פונקציות וירטואליות 52

3.3 מחלקות אבסטרקטיות 61

3.3.1 Pure Function - פונקציה טהורה 61

3.3.2 לשם מה מחלקות אבסטרקטיות? *62*
3.4 Virtual Destructor 63
3.4.1 אופן ההגדרה של Virtual Destructor *63*
3.4.2 לשם מה Virtual Destructor? *63*
3.6 Static Binding vs. Dynamic Binding 74
3.6.1 Static Binding *74*
3.6.2 Dynamic Binding (Late Binding) *74*
3.6.3 ביצועים *77*
3.7 הגדרת Interfaces 78
שיעור 4: נושאים נוספים **87**
4.1 משתנים מסוג mutable 87
4.2 האופרטור new 90
4.2.1 זריקת exception *90*
4.2.2 הקבוע nothrow *91*
שיעור 5: Templates **93**
5.1 פונקציות תבנית 93
5.2 Template Specializations 96
5.2.1 Overloaded Functions vs. Specialized Functions *96*
5.3 Template Classes 98
5.4 העברת פרמטרים לתבנית 103
5.4.1 ברירת מחדל של פרמטרים של תבנית *105*
5.5 שימוש - אובייקט פונקציה 105
5.6 Template ממספר סוגים 106
5.7 המילה השמורה typename 109
שיעור 6: RTTI & Casting **113**
6.1 Runtime Type Information - RTTI 113
6.1.1 המחלקה type_info *113*
6.1.2 האופרטור typeid *114*

6.2 CASTING - המרת סוגים ב-C++ 117
6.2.1 static_cast *118*
6.2.2 dynamic_cast *121*
6.2.3 const_cast *126*
6.2.4 reinterpret_cast *129*
שיעור 7: STL - THE STANDARD LIBRARY **131**
7.1 מבוא לאיטרטורים 131
7.2 האוסף VECTOR - מערך דינמי 132
7.2.1 הגדרת הווקטור *132*
7.2.2 אתחול וקטור *132*
7.2.3 גודל הווקטור וניצול הזיכרון *133*
7.2.4 מחיקת כל איברי המערך *133*
7.2.5 הוספת איברים *134*
7.2.6 מחיקת איברים במערך *134*
7.2.7 גודל מקסימלי של מערך *134*
7.2.8 איטרטורים לתחילת המערך וסוף המערך *135*
7.2.9 אחזור איבר *136*
7.2.10 התייחסויות לאיבר ראשון ואיבר אחרון *136*
7.3 אחסון אובייקטים ממחלקה 136
7.3.1 האלגוריתם sort *137*
7.4 המחלקות PHONEENTRY ו-PHONEENTRYPREDICATE 139
7.5 האוסף LIST - רשימה מעגלית דו כיוונית 143
7.5.1 פונקציות של list *144*
7.5.2 האלגוריתם find_if *145*
7.6 האוסף DEQUE - תור כפול (DOUBLE ENDED QUEUE) 149
7.6.1 פונקציות של deque *150*
7.6.2 האלגוריתם stable_sort *151*

הקדמה

ספר זה מיועד למתכנתים המכירים את יסודות שפת C++ ומעוניינים להרחיב את ידיעותיהם ולהעמיק את הבנתם בנושאים שונים בשפה.

אני עוסק בתכנות משנת 1998. במהלך שנות עבודתי בחברות שונות נתקלתי לא פעם במצבים שבהם היכרות עם התחביר של השפה בלבד אינה מספיקה. כדי להשתמש נכון ביכולותיה של C++ , חשוב להבין גם כיצד פועלים המנגנונים שמאחורי הקלעים, כיצד ה-Compiler מתייחס לקוד שאנו כותבים ומהן ההשלכות של השימוש במנגנונים השונים.

בספר זה בחרתי להתמקד במספר נושאים חשובים בשפת C++, בהם בנייה והעתקה של אובייקטים (Constructors), ירושה (Inheritance), רב צורתיות (Polymorphism), תבניות (Templates), זיהוי סוגים בזמן ריצה (RTTI), המרות סוגים ושימוש בספריית STL.

הנושאים מוסברים באמצעות דוגמאות קוד, תוך התייחסות לא רק לאופן השימוש במנגנונים השונים, אלא גם להתנהגותם, למגבלותיהם ולמקרים שבהם שימוש לא נכון עלול לגרום לתוצאות בלתי צפויות.

דוגמאות הקוד נבדקו בסביבת Visual C++ על מערכת ההפעלה Windows. מומלץ לקמפל ולהריץ את הדוגמאות, לבצע בהן שינויים ולבחון את התוצאות. בדרך זו ניתן להבין טוב יותר את הנושאים הנלמדים.

הספר אינו מתיימר להקיף את כל שפת C++ או ללמד אותה מן היסוד, אלא להרחיב ולהעמיק את הידע בנושאים שנבחרו.

אני מקווה שהספר יהיה לעזר למתכנתים המעוניינים להכיר טוב יותר את יכולותיה של השפה ולעשות בהן שימוש נכון ויעיל.

קוד המקור לדוגמאות והתרגילים שבספר זמין במאגר ה-GitHub:

https://github.com/IlanAmoyal/advanced_cpp_2010

אילן אמויאל



מוזמנים להתחבר
אליי גם ב־LinkedIn

שיעור 1:
Constructors

שים לב: קוד התוכניות של שיעור זה נמצא בספרייה Demos\Chapter 01.

בשיעור זה נסקור מספר היבטים חשובים הנוגעים ל-Constructors ואשר לעתים קרובות גורמים לאי הבנת התהליך של בניית אובייקטים.

1.1 Default Constructor (Parameterless Constructor)

זהו Constructor שאינו מקבל פרמטרים. אם לא הוגדר אף Constructor במחלקה, ה-Compiler ייצור באופן אוטומטי Default Constructor למחלקה. לעומת זאת, אם הוגדר במחלקה Constructor כלשהו המקבל פרמטרים, לא ייוצר באופן אוטומטי Default Constructor. במקרה כזה - אם ברצוננו ב-Constructor שאינו מקבל פרמטרים - עלינו להגדירו באופן מפורש.

דוגמה

להלן קטע קוד הממחיש זאת (Ex01):

```
class ClassA
{
    /* A default constructor is generated automatically by the
compiler. */
};

class ClassB
{
public:
    ClassB(int dummy){ }
    /* This class has NO default constructor */
};

class ClassC
{
public:
    ClassC(){ }
    ClassC(int dummy){ }
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    /* This class has a default constructor because it was
    declared explicitly. */
};

int main(int argc, char *argv[])
{
    ClassA a;
    // ClassB b; this line executes a compilation error
    ClassC c;
}
```

כפי שאפשר לראות בקטע קוד זה, המחלקה היחידה שאי אפשר ליצור ממנה אובייקט ללא העברת פרמטרים היא המחלקה ClassB שבה מוגדר Constructor המקבל פרמטר, ולכן ה-Compiler לא יוצר עבורנו Default Constructor.

Copy Constructor 1.2

זהו Constructor האחראי על העתקת שדות מאובייקט אחד לאובייקט חדש שנוצר. אם לא נגדיר Copy Constructor, ה-Compiler ייצור עבורנו באופן אוטומטי Copy Constructor המבצע **העתקת חברי מחלקה (Memberwise Copy)** של השדות מאובייקט אחד לאובייקט חדש שנוצר (עוד על העתקה רדודה, העתקה עמוקה ו-Memberwise בהמשך).

חשוב להדגיש, בניגוד ל-Default Constructor, ה-Copy Constructor נוצר בכל מקרה, בין אם הגדרנו Constructor אחר ובין אם לאו.

Copy Constructor מקבל כפרמטר מופע של המחלקה, ומעתיק ממופע זה את השדות למופע הנוכחי.

דוגמה

להלן קטע קוד הממחיש זאת (Ex02):

```
#include <iostream>

using namespace std;

class ClassA
{
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
private:
    int m_value;

public:
    ClassA(int value = 0){ m_value = value; }
    /* A copy constructor is automatically created! */

    void Show() { cout << "ClassA: Value = " << this->m_value <<
endl; }
};

class ClassB
{
private:
    int m_value;

public:
    ClassB(int value = 0 ){ m_value = value; }
    ClassB(const ClassB &obj) { m_value = obj.m_value; }

    void Show() { cout << "ClassB: Value = " << m_value << endl; }
};

int main(int argc, char *argv[])
{
    cout << "a...\n";
    ClassA a(7);
    a.Show();

    cout << "\ncopy of a...\n";
    ClassA aCopy(a); /* The compiler copy constructor is executed */
    aCopy.Show();

    cout << "\nb...\n";
    ClassB b(9);
    b.Show();

    cout << "\ncopy of b...\n";
    ClassB bCopy(b);
    bCopy.Show();
}
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פלט התוכנית הוא:

```
a...
ClassA: Value = 7

copy of a...
ClassA: Value = 7

b...
ClassB: Value = 9

copy of b...
ClassB: Value = 9
```

רואים שגם עבור ClassA, אשר עבורו לא הוגדר Copy Constructor באופן מפורש, ה-Compiler יצר Copy Constructor כך שיכולנו ליצור את aCopy כהעתק של a.

הערה: ישנן כמה דרכים להגדיר Copy Constructor, למשל:

```
1. MyClass(const MyClass &c) { }
2. MyClass(MyClass &c) { }
3. MyClass(const MyClass &c, int value = 0) { }
                                                ועוד...
```

הדרך הנפוצה להגדיר Copy Constructor היא כמו בשורה 1. הסיבה לכך היא שבהעברת האובייקט כפרמטר מסוג const אנו מבטיחים שה-Copy Constructor יוכל לקבל גם אובייקטים שהוגדרו כ-constant. לעומת זאת, שורה 2 מגדירה Copy Constructor הניתן לאתחול רק על ידי אובייקט שאינו constant.

חשוב לשים לב שיש להשתמש באופרטור & (אופרטור ההתייחסות). ללא אופרטור זה נקבל שגיאת קומפילציה, שכן Copy Constructor אינו יכול לקבל אובייקט מהמחלקה שלו כפרמטר המועבר By Value (ראה גם #1.2.3 מתי מופעל ה-Copy Constructor). אם הדבר היה אפשרי, העברת הפרמטר לפי ערך הייתה מחייבת הפעלה חוזרת של אותו Copy Constructor, וכך הייתה נוצרת רקורסיה אינסופית.

1.2.1 Shallow Copying - העתקה רדודה

זוהי העתקה של ערכי המשתנים החברים באובייקט המקור כמו שהם למשתנים החברים באובייקט החדש. העתקה מעין זו אינה מבחינה בין טיפוסים שבמהותם הם Value Types,

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

כמו למשל: int, float וכולי, לבין משתנים המתייחסים לאזור אחר בזיכרון, כלומר משתנים מסוג Reference Type (כלומר Pointers), כמו למשל: int *, char * וכן הלאה. ברור כי העתקה של ערך מצביע יכולה להיות מסוכנת כיוון שייוצרו שני אובייקטים המצביעים לאותו בלוק זיכרון (שבמקרים רבים הוקצה באופן דינמי). במצב כזה אובייקט אחד עלול לשחרר את הזיכרון, ואילו אובייקט שני ימשיך להתייחס לאותו אזור זיכרון שכבר שוחרר.

כאמור, ה-Copy Constructor הנוצר באופן אוטומטי על ידי ה-Compiler מבצע העתקה של חברי המחלקה (Memberwise Copy, מוסבר בהמשך).

דוגמה

להלן קטע קוד הממחיש זאת (Ex04)

```
#include <iostream>
#include <cstdlib>


using namespace std;

class ClassA
{
private:
    char *m_strValue;
    int m_nId;
public:
    ClassA(const char *strValue) { m_strValue = strdup(strValue); }
    void SetId(int id) { m_nId = id; }
    ~ClassA()
    {
        cout << "Object ID: " << m_nId << ", trying to release the
address: " << (int)m_strValue << "\n";
        std::free(m_strValue);
        cout << "success\n";
    }
};

int main(int argc, char **argv)
{
    ClassA a("Hello world.");
    a.SetId(1);
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    ClassA aCopy(a);
    aCopy.SetId(2);
}
```

פלט התוכנית הוא:

```
Object ID: 2, trying to release the address: 202032
success
Object ID: 1, trying to release the address: 202032
```

כפי שרואים, שחרור הזיכרון הדינמי של אובייקט מספר 1 אינו מסתיים בהצלחה (יש לזכור שאובייקטים בבלוק משוחררים בסדר הפוך לסדר הצהרתם).

הערה: אם למחלקה די בהעתקה רדודה, אין צורך להגדיר Copy Constructor.

1.2.2 Deep Copying - העתקה עמוקה

כדי להימנע ממצב כזה מגדירים Copy Constructor באופן מפורש וממשים בו **העתקה עמוקה (Deep Copying)**. בהעתקה עמוקה נדאג לכך שבלוק זיכרון ישוכפל כך שלא ייוצר מצב בו שני אובייקטים מתייחסים לאותו אזור זיכרון.

על מנת לתקן את התוכנית הקודמת, נגדיר במחלקה את ה-Copy Constructor באופן מפורש (Ex02 בספריית Demos\Lesson 01), באופן הבא:

```
ClassA(const ClassA &c) { m_strValue = strdup(c.m_strValue); }
```

לחילופין, אפשר לדאוג במקרה של העתקה למנות את מספר ההתייחסויות לבלוק הזיכרון, ולשחרר את בלוק הזיכרון רק כאשר מספר ההתייחסויות יורד לאפס (שיטה זו נקראת Reference Counting, והיא נהוגה למשל בטכנולוגיית COM).

הערה: הגדרת Copy Constructor אינה מספקת פתרון להעתקה עמוקה בכל המצבים, שכן אופרטור ההשמה (=) מבצע Memberwise Assignment. משתנים בסיסיים ומצביעים מועתקים לפי ערכם, ואילו עבור חברים שהם אובייקטים מופעל אופרטור ההשמה שלהם. לכן, במחלקות המנהלות זיכרון דינמי באמצעות מצביעים, יש לממש גם את אופרטור ההשמה באופן מתאים.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

1.2.3 Memberwise Copy - העתקת חברי מחלקה

כאשר ה-Compiler יוצר באופן אוטומטי Copy Constructor, הוא מבצע העתקה של חברי המחלקה, תהליך הנקרא Memberwise Copy. משתנים מסוגים בסיסיים ומצביעים מועתקים לפי ערכם. לעומת זאת, עבור משתנים שהם אובייקטים של מחלקות אחרות מופעל ה-Copy Constructor של כל אחד מהם.

לכן, מידת העומק של ההעתקה תלויה במימוש ה-Copy Constructors של המחלקות המרכיבות את האובייקט, והיא עשויה לכלול העתקה רדודה או עמוקה.

1.2.4 מתי מופעל ה-Copy Constructor

ישנם כמה מצבים בהם מופעל ה-Copy Constructor:

- כאשר מעבירים אובייקט לפונקציה By Value.
- כאשר אובייקט מוחזר מפונקציה By Value.
- כאשר אובייקט נזרק כ-Exception.
- כאשר מאתחלים אובייקט באמצעות אובייקט אחר - בשורת ההצהרה עצמה, למשל:
  Object obj2 = obj1

שים לב: הפקודה obj1=obj2 לא מפעילה את ה-Copy Constructor אלא מפעילה את ה-Assignment Operator (האופרטור =).

1.3 *Explicit Constructor*

Constructor המקבל פרמטר אחד מגדיר גם המרה מרומזת (Implicit Conversion) מאותו סוג של פרמטר למופע של המחלקה.

דוגמה

להלן קטע קוד הממחיש זאת (Ex05):

```
#include <iostream>

using namespace std;

class Complex
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
{
private:
    double m_real, m_imaginary;
public:
    Complex(double real) { m_real = real; m_imaginary = 0.0; }
    Complex(double real, double imaginary) { m_real = real;
m_imaginary = imaginary; }
    void Show() { cout << "(" << m_real << ", " << m_imaginary <<
")\n"; }
};

int main(int argc, char **argv)
{
    Complex z1(3);
    z1.Show();

    Complex z2 = 4;
    z2.Show();
}
```

פלט התוכנית הוא:

```
(3, 0)
(4, 0)
```

כפי שאפשר לראות, האובייקט z2 אותחל על ידי ערך מסוג double מבלי שהגדרנו המרה כזו. למעשה, ה-Compiler משתמש ב-constructor הראשון כאופרטור להמרה מרומזת מ-double לאובייקט מסוג Complex.

במקרים רבים זוהי התנהגות רצויה, אם כי ישנם מקרים בהם לא נרצה שתתבצע המרה מרומזת אלא אם נבקש זאת באופן מפורש.

למשל, נניח שהגדרנו מחלקה בשם String הניתנת לאתחול באמצעות מספר שלם הגורם להקצאה מוקדמת של מספר תווים במחרוזת, באופן הבא:

```
class String
{
    ...
public:
    String(size_t bufferSize) { m_buffer = new char[m_bufferSize =
bufferSize]; }
    ...
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
};
```

השורה הבאה תגרום להמרה ממספר שלם למחרוזת:

```
String s = 'A';
```

בשורה זו תתבצע הקצאה של כמות תווים לפי הערך ה-ASCII של התו 'A' (הערך 65), אם כי ייתכן מאוד כי המתכנת שכתב את השורה הנ"ל התכוון ליצור מחרוזת המכילה את התו 'A' בלבד.

כדי למנוע המרה מרומזת כזו נקדים להגדרת ה-constructor את המילה explicit. באופן כזה לא נוכל לבצע המרה אלא אם נרשום אותה באופן מפורש.

דוגמה

להלן קטע קוד הממחיש זאת (Ex06):

```
#include <iostream>
#include <cstring>

using namespace std;

class String
{
private:
    char *m_buffer;
    int m_bufferSize;
public:
    explicit String(size_t bufferSize)
    { m_buffer = new char[m_bufferSize = bufferSize]; }

    String(const char *str)
    {
        m_bufferSize = std::strlen(str) + 1;
        m_buffer = new char[m_bufferSize];
        std::memcpy(m_buffer, str, m_bufferSize);
    }

    void Set(const char *str)
    { strncpy(m_buffer, str, m_bufferSize); m_buffer[m_bufferSize -
1] = '\0'; }

    void Show() { cout << m_buffer << endl; }
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    ~String() { delete[] m_buffer; }
};

int main(int argc, char **argv)
{
    // String s = 5; ERROR!!
    String s1(5); // OK!
    s1.Set("Hello world");
    s1.Show();

    String s2 = (String)5;
    s2.Set("Oh yehh");
    s2.Show();
}
```

פלט התוכנית הוא:

```
Hell
Oh y
```

כפי שאפשר לשים לב, ניסיון להמיר באופן מרומז את המספר 5 למופע של String יגרום לשגיאה. לעומת זאת, אם נציין המרה מפורשת (כפי שמאותחל s2), הפעולה תצליח.

1.4 אתחול קבועים ב-Constructor

ניתן להצהיר על משתנים קבועים במחלקה שאינם סטטיים. במקרה כזה, אפשר לאתחל את אותם קבועים אך ורק ב-Initialization List שב-Constructor.

דוגמה

קוד הדוגמה הבאה ממחיש זאת היטב (Ex07):

```
#include <iostream>

using namespace std;

class ConstClass
{
private:
    const int m_relative;
    int m_value;

public:
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    ConstClass(int relative, int value) : m_relative(relative),
m_value(value) { }
    void WhoIsGreater()
    {
        if(m_relative > m_value)
        {
            cout << "relative > value" << endl;
        }
        else if(m_relative < m_value)
        {
            cout << "relative < value" << endl;
        }
        else
        {
            cout << "relative == value" << endl;
        }
    }
};

int main(int argc, char **argv)
{
    ConstClass(7, 9).WhoIsGreater();
    ConstClass(9, 7).WhoIsGreater();
    ConstClass(8, 8).WhoIsGreater();
}
```

פלט התוכנית הוא:

```
relative < value
relative > value
relative == value
```

כפי שרואים, אתחול המשתנה m_relative יכול להתבצע רק ב- Constructor Initialization List.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שיעור 2:
Inheritance

שים לב: קוד התוכניות של שיעור זה נמצא בספרייה Demos\Chapter 02.

2.1 הרשאות גישה

ישנן שלוש הרשאות גישה אפשריות לחברי מחלקה:

- **private** - חבר המחלקה הינו פרטי למחלקה ואין אליו גישה, לא מקוד חיצוני ולא ממחלקות נגזרות. הגישה לחבר מסוג private היא רק לחברי המחלקה עצמה או למחלקות או פונקציות שהוכרו בתוך המחלקה כ-friends.
- **public** - חבר מחלקה מסוג public נגיש הן לקוד חיצוני והן למחלקות נגזרות.
- **protected** - חבר מחלקה מסוג protected אינו נגיש מקוד חיצוני אך נגיש למחלקות נגזרות. כלומר כלפי קוד חיצוני הוא מתנהג כ-private ואילו כלפי קוד של מחלקות נגזרות הוא מתנהג כ-public. גם כאן, כמו ב-private, קוד חיצוני (מחלקות או פונקציות) יוכלו לגשת לחבר אם הן הוכרו בתוך המחלקה כ-friends.

2.2 סוגי הירושה

C++ מגדיר שלושה סוגים של ירושה: ירושה פרטית, ירושה מוגנת וירושה ציבורית.

2.2.1 ירושה פרטית

- בירושה פרטית כל ה-Public Members וה-Protected Members הופכים להיות פרטיים במחלקה הנגזרת. כל אותם חברים יהיו נגישים מתוך קוד שבמחלקה הנגזרת, אך לא יהיו נגישים לעולם החיצון.
- ירושה פרטית דומה להכלה של משתנה מסוג מחלקת הבסיס במחלקה הנגזרת.
- במקרה של ירושה פרטית, מצביע למחלקת הבסיס לא יכול להצביע על אובייקטים של המחלקה הנגזרת. מדוע? כי אז הוא ישבור את הירושה הפרטית בכך שיאפשר לקוד חיצוני לגשת ל-Public Members של מחלקת הבסיס.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- ניתן לשנות את הרשאות הגישה של חברי מחלקה יורשת על ידי הצהרת using. בכל מקרה, לא ניתן לשנות רמת גישה של חבר מחלקת בסיס פרטי לרמה מתירה יותר (protected או public), אם כי, חבר מחלקת בסיס שניתן לגשת אליו (כלומר שרמת ההרשאה שלו היא Protected או Public), ניתן לחשוף לקוד חיצוני (להפוך אותו ל-Public).
- אם סוג ההורשה לא צוין - ברירת המחדל עבור class היא הורשה פרטית (עבור struct ברירת המחדל היא הורשה ציבורית).

דוגמה

הדוגמה הבאה ממחישה ירושה פרטית (Ex01):

```
#include <iostream>

using namespace std;

class base
{
private:
    int a, b;

protected:
    int c, d;

public:
    int e, f;
    base() { a = 1; b = 2; c = 3; d = 4; e = 5; f = 6; }
};

class derived : private base
{
public:
    int g, h;
    derived() : base() {g = 7; h = 8; }
    void func()
    {
        // 'base::a' : cannot access private member declared in class
'base'
        // a = 1;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
        // OK
        c = 12;
    }
};

class anotherDerivation : private base
{
public:
    int g, h;
    anotherDerivation() : base() {g = 70; h = 80; }
    using base::e;
    using base::c; /* OK */

    // 'base::a' is not accessible from 'base'
    //using base::a;
};

int main(int argc, char **argv)
{
    derived d;

    // OK
    d.g = 10;

    // 'type cast' : conversion from 'derived *' to 'base *' exists,
but
    // is inaccessible
    // base *pB = &d;

    // 'base::e' not accessible because 'derived' uses 'private' to
inherit
    // from 'base'
    // d.e = 6;

    anotherDerivation ad;

    // OK
    ad.e = 10;
    ad.c = 9;

    /* Oh, we can cheat the compiler */
    reinterpret_cast<base*>(&d)->e = 11;
}
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הערות

- כפי שאפשר לראות ירושה פרטית מסתירה את חברי המחלקה הציבוריים מקוד חיצוני (main).
- לא ניתן להצביע באמצעות מחלקת הבסיס לאובייקט מהמחלקה הנגזרת.
- כפי שרואים במחלקה anotherDerivation, אפשר להחזיר הרשאת גישה ציבורית לחבר אפילו אם הירושה הייתה פרטית. עושים זאת באמצעות הצהרת using.
- השורה הממוסגרת הראשונה: מרגע שיש לנו גישה במחלקה הנגזרת לחבר ממחלקת הבסיס, אנו יכולים לחשוף אותו לקוד חיצוני באמצעות המילה using. זה מה שעשינו לחבר c של מחלקת הבסיס.
- השורה הממוסגרת השנייה - אנו יכולים לרמות את ה-Compiler בכך שנמיר באמצעות reinterpret_cast את המצביע למחלקה d, למצביע מחלקת b. באופן כזה אנו יכולים לשנות את החבר e הציבורי של base, שבינתיים הפך לפרטי ב-derived

הערה: בהמשך הספר מוסברים אופרטורי ה-casting השונים, ביניהם גם reinterpret_cast.

שים לב: החברים הפרטיים של מחלקת הבסיס נשארים פרטיים של מחלקת הבסיס ולמחלקה הנגזרת בכל מקרה אין גישה אליהם.

2.2.2 ירושה מוגנת

ירושה מוגנת פועלת על פי עקרונות דומים של ירושה פרטית, אך עם הבדלים מסוימים:

- בירושה מוגנת כל ה-Public Members וה-Protected Members הופכים להיות Protected במחלקה הנגזרת. כל אותם חברים יהיו נגישים מתוך קוד שבמחלקה הנגזרת, אך לא יהיו נגישים לעולם החיצון.
- במקרה של ירושה מוגנת, מצביע למחלקת הבסיס לא יכול להצביע על אובייקטים של המחלקה הנגזרת. מדוע? כי אז הוא ישבור את הירושה המוגנת בכך שיאפשר לקוד חיצוני לגשת ל-Public Members של מחלקת הבסיס.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- ניתן לשנות לחברי מחלקה נורשים את הרשאות הגישה על ידי הצהרת using. בכל מקרה, לא ניתן לשנות רמת גישה של חבר מחלקת בסיס פרטי לרמה מתירה יותר (protected או public), אם כי, חבר מחלקת בסיס שניתן לגשת אליו (כלומר שרמת ההרשאה שלו היא Protected או Public), ניתן לחשוף לקוד חיצוני (להפוך אותו ל-Public).
- ניתן להשתמש ב-using כדי להפחית את רמת הגישה של חבר, למשל מ-protected ל-private.

הדוגמה הקודמת תעבוד באותה הצורה ותזרוק את אותן שגיאות, אם נסיר הערות ונחליף את הירושה הפרטית בירושה מוגנת.

2.2.3 ירושה ציבורית

בירושה ציבורית נשמרים עקרונות הירושה, אך קיימים מספר הבדלים חשובים לעומת ירושה פרטית ומוגנת:

- בירושה ציבורית ה-Public Members נשמרים כ-Public במחלקה הנגזרת, ה-Protected Members נשמרים כ-Protected, ואילו ל-Private Members כלל אין גישה.
- במקרה של ירושה ציבורית, מצביע למחלקת הבסיס יכול להצביע על אובייקטים של המחלקה הנגזרת.
- ניתן להשתמש ב-using כדי להפחית את רמת הגישה של חבר, למשל מ-protected ל-private, או מ-public ל-protected או private.
- ב-struct ברירת המחדל היא ירושה ציבורית (ב-class ברירת המחדל היא ירושה פרטית).

דוגמה

הדוגמה הבאה ממחישה ירושה ציבורית (Ex02):

```
#include <iostream>

using namespace std;
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
class base
{
private:
    int a, b;

protected:
    int c, d;

public:
    int e, f;
    base() { a = 1; b = 2; c = 3; d = 4; e = 5; f = 6; }
};

class derived : public base
{
private:
    /* makes c private */
    using base::c;

protected:
    /* making e protected */
    using base::e;

public:
    int g, h;
    derived() : base() {g = 7; h = 8; }
};

int main(int argc, char **argv)
{
    derived d;

    // 'derived::e' : cannot access protected member declared in
class 'derived'
    //d.e = 10;

    /* OK */
    base *pB = &d;
    pB->e = 10; /* bypasses the privacy! */
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הערות

- כפי שרואים, אפשר להפחית את רמת הגישה של חברים מ-base כך שיהיו פחות נגישים ב-derived.
- לא ניתן לגשת למשתנה e באמצעות d, אך זה אפשרי מהמצביע pB. ה-Compiler לא מונע יצירת מצביע מסוג base ל-derived, למרות שהפחתנו את רמת הגישה של החבר הציבורי e במחלקה base כאשר ירשנו אותו במחלקה derived. ה-Compiler מונע יצירת מצביע רק כאשר הירושה היא פרטית או מוגנת.

2.3 ירושה מרובה

C++ מאפשרת ירושה ממספר מחלקות בו זמנית. תכונה זו היא מהחשובות בשפה, אך גם מהמסוכנות שבהן, אם לא משכילים להשתמש בה באופן תקין. שפות מונחות עצמים כמו Java ו-C# אשר פותחו לאחר C++, הסירו את האפשרות לגזור ממספר מחלקות לאחר שהתברר ששימוש בתכונה זו הוא מקור לבאגים רבים.

2.3.1 פריסת הזיכרון

כאשר ה-Compiler בונה את בלוק הזיכרון של האובייקט, בתחילת הבלוק הוא ממקם את בלוקי הזיכרון ממחלקות הבסיס, ורק לאחר מכן נוספים החברים של המחלקה הנגזרת עצמה, שהאובייקט הוא מופע שלה. למשל:

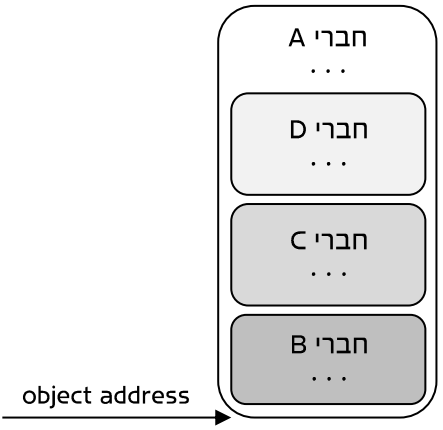
```
class A : public B, public C, protected D { ... };
```

בקטע קוד זה המחלקה A יורשת באופן ציבורי מהמחלקות B ו-C, ובאופן מוגן מהמחלקה D.

הערה: פריסת הזיכרון המתוארת מתייחסת למימוש שנבדק בסביבת Visual C++. תקן C++ אינו מחייב פריסת זיכרון זו.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פריסת הזיכרון עבור אובייקט מסוג A היא כדלקמן:



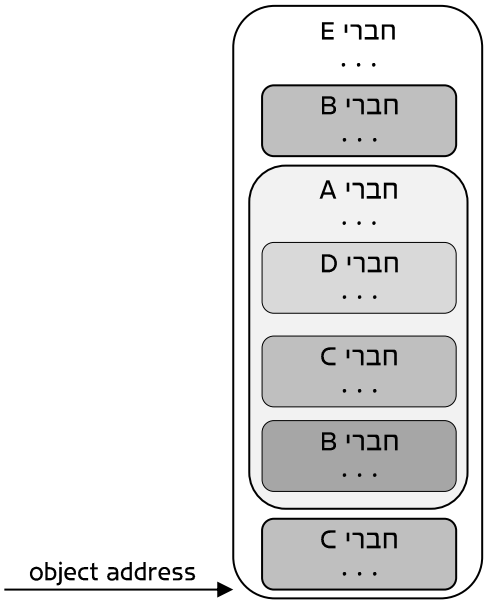
אובייקט מסוג A בנוי כך שבתחילת בלוק הזיכרון שלו נמצאים חברי האובייקט B, אחריהם נמצאים חברי C, לאחר מכן חברי D ורק לאחר מכן נוספים החברים החדשים של A. כל קטע הזיכרון הזה מייצג אובייקט של A.

נניח שאנחנו מגדירים מחלקה חדשה בשם E, באופן הבא:

```
class E : public C, public A, public B { ... };
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פריסת הזיכרון של אובייקט מסוג E היא כדלקמן:



דוגמה

קטע הקוד הבא ממחיש את פריסת הזיכרון במקרה של ירושה מרובה (Ex03):

```
#include <iostream>

using namespace std;

void DumpMemory(const char *name, int *pBase, size_t size)
{
    cout << name << " (" << hex << (int)pBase << ") " << ":[ ";
    if(size > 0)
    {
        cout << hex << *pBase;
    }
    for(int i = 1; i < size / sizeof(int); i++)
    {
        cout << " | " << hex << *(pBase + i);
    }
    cout << " ]" << endl;
}
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
class A
{
public:
    int a;
    A() { a = 0xAAAAAAAA;}
};

class B
{
public:
    int b;
    B() { b = 0xBBBBBBBB; }
};

class C : public A, public B
{
public:
    int c;
    C() { c = 0xCCCCCCCC; }
};

class D : public A, public C
{
public:
    int d;
    D() { d = 0xDDDDDDDD; }
};

class E : public A, public D, public C
{
public:
    int e;
    E() { e = 0xEEEEEEEE; }
};

int main(int argc, char **argv)
{
    A a;
    B b;
    C c;
    D d;
    E e;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    DumpMemory("a", (int*)&a, sizeof(a));
    DumpMemory("b", (int*)&b, sizeof(b));
    DumpMemory("c", (int*)&c, sizeof(c));
    DumpMemory("d", (int*)&d, sizeof(d));
    DumpMemory("e", (int*)&e, sizeof(e));
}
```

פלט התוכנית הוא:

```
a (38f8ac) :[ aaaaaaaa ]
b (38f8a0) :[ bbbbbbbb ]
c (38f88c) :[ aaaaaaaa | bbbbbbbb | cccccccc ]
d (38f870) :[ aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc | dddddddd ]
e (38f840) :[ aaaaaaaa | aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc |
dddddddd |
              aaaaaaaa | bbbbbbbb | cccccccc | eeeeeeee ]
```

הערות

- לצורך הנוחות, בכל מחלקה אתחלנו את ערך המשתנה בערך הקסדצימלי שקל לזהותו.
- לצורך הנוחות הגדרנו רק משתנים מסוג int.
- הפונקציה DumpMemory מדפיסה מספר int החל מהכתובת המועברת אליה לפי ה-size שהועבר.
- אפשר לראות איך כל מחלקת בסיס מוסיפה את החברים שלה לפריסת הזיכרון, ורק לבסוף נוספים חברי המחלקה המוגדרת.
- כמו כן, רואים כי מחלקות הבסיס מסודרות בזיכרון בדיוק על פי הסדר בו הן נרשמו ברשימת מחלקות הבסיס של המחלקה המוגדרת.

Name Ambiguity 2.3.2

נתבונן בהיררכיית המחלקות הבאה:

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


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
class B : public A { };

class C : public A { };

class D : public B, public C { };

class E : public B, public C
{
public:
    int a; /* Hides the inherited a */
}

void main()
{
    D d;
    // d.a = 5; /* Error - ambiguity */

    d.B::a = 5; /* Changes the a that was inherited from B */
    d.C::a = 6; /* Changes the a that was inherited from C */

    E e;
    e.a = 11; /* OK! */
}
```

הסבר

- בלוק הזיכרון של D כולל בתוכו שני מופעים של a, אחד מתקבל מהירושה של B והשני מתקבל מהירושה של C.
- כאשר מנסים לגשת למשתנה a מתוך אובייקט d, מתקבלת שגיאה. ה-Compiler לא יודע בדיוק לאיזה a לגשת.
- הפתרון, כפי שרואים, הוא שימוש ב-Full Qualified Name בגישה לחבר a.
- לעומת זאת, עבור e לא קיימת בעיה כי החבר a שמוגדר ב-E מסתיר את אלה שנורשים.

הערה: כשאומרים שהחבר a של E מסתיר את החבר a הנורש מ-B ומ-C, אין הכוונה שחברים אלה נעלמו מבלוק הזיכרון, אלא שהוקצה מקום לעוד משתנה a, השייך ישירות למשתנה E.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הכלל הוא פשוט, ה-Compiler קובע לאיזה חבר מתייחס השם לפי מידת הקרבה ב-Scope. במקרה שיש שני חברים או יותר בעלי אותו שם נמצאים באותו מרחק ברמת ה-Scope תתקבל שגיאת Ambiguity.

דוגמה

הדוגמה הבאה ממחישה באופן מקיף יותר עניין זה (Ex04):

```
#include <iostream>

using namespace std;

void DumpMemory(const char *name, int *pBase, size_t size)
{
    cout << name << " (" << hex << (int)pBase << ") " << ":[ ";
    if(size > 0)
    {
        cout << hex << *pBase;
    }
    for(int i = 1; i < size / sizeof(int); i++)
    {
        cout << " | " << hex << *(pBase + i);
    }
    cout << " ]" << endl;
}

class A
{
public:
    int a;
    A() { a = 0xAAAAAAAA;}
};

class B
{
public:
    int b;
    B() { b = 0xBBBBBBBB; }
};

class C : public A, public B
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
{
public:
    int c;
    C() { c = 0xCCCCCCCC; }
};

class D : public A, public C
{
public:
    int d;
    D() { d = 0xDDDDDDDD; }
};

class E : public A, public C
{
public:
    int a; /* hides the one inherited from a and c */
    int e;
    E() { a = 0xEEEEEEEE; e = 0xEEEEEEEE; }
};

class F : public E
{
};

int main(int argc, char **argv)
{
    cout << "Inspect D..." << endl;

    D d;
    DumpMemory("d", (int*)&d, sizeof(d));

    // ambiguous access of 'a'
    // d.a = 0x;

    d.A::a = 0x11111111;
    DumpMemory("d", (int*)&d, sizeof(d));

    d.C::a = 0x22222222;
    DumpMemory("d", (int*)&d, sizeof(d));

    d.C::A::a = 0x33333333;
    DumpMemory("d", (int*)&d, sizeof(d));
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    cout << "\nInspect E..." << endl;
    E e;
    DumpMemory("e", (int*)&e, sizeof(e));

    e.a = 0x11111111; /* works! */
    DumpMemory("e", (int*)&e, sizeof(e));

    e.A::a = 0x22222222;
    DumpMemory("e", (int*)&e, sizeof(e));

    e.C::a = 0x33333333;
    DumpMemory("e", (int*)&e, sizeof(e));

    e.C::A::a = 0x44444444;
    DumpMemory("e", (int*)&e, sizeof(e));

    cout << "\nInspect F..." << endl;
    F f;
    DumpMemory("f", (int*)&f, sizeof(f));

    f.a = 0x11111111;
    DumpMemory("f", (int*)&f, sizeof(f));

    f.E::a = 0x22222222;
    DumpMemory("f", (int*)&f, sizeof(f));

    f.A::a = 0x33333333;
    DumpMemory("f", (int*)&f, sizeof(f));

    f.E::A::a = 0x44444444;
    DumpMemory("f", (int*)&f, sizeof(f));

    f.C::a = 0x55555555;
    DumpMemory("f", (int*)&f, sizeof(f));

    f.E::C::a = 0x66666666;
    DumpMemory("f", (int*)&f, sizeof(f));
}
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פלט התוכנית הוא:

```
Inspect D...
d (35f878) :[ aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc | dddddddd ]
d (35f878) :[ 11111111 | aaaaaaaa | bbbbbbbb | cccccccc | dddddddd ]
d (35f878) :[ 11111111 | 22222222 | bbbbbbbb | cccccccc | dddddddd ]
d (35f878) :[ 11111111 | 33333333 | bbbbbbbb | cccccccc | dddddddd ]

Inspect E...
e (35f858) :[ aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc | eeeeeeee |
eeeeeeee ]
e (35f858) :[ aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc | 11111111 |
eeeeeeee ]
e (35f858) :[ 22222222 | aaaaaaaa | bbbbbbbb | cccccccc | 11111111 |
eeeeeeee ]
e (35f858) :[ 22222222 | 33333333 | bbbbbbbb | cccccccc | 11111111 |
eeeeeeee ]
e (35f858) :[ 22222222 | 44444444 | bbbbbbbb | cccccccc | 11111111 |
eeeeeeee ]

Inspect F...
f (35f838) :[ aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc | eeeeeeee |
eeeeeeee ]
f (35f838) :[ aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc | 11111111 |
eeeeeeee ]
f (35f838) :[ aaaaaaaa | aaaaaaaa | bbbbbbbb | cccccccc | 22222222 |
eeeeeeee ]
f (35f838) :[ 33333333 | aaaaaaaa | bbbbbbbb | cccccccc | 22222222 |
eeeeeeee ]
f (35f838) :[ 44444444 | aaaaaaaa | bbbbbbbb | cccccccc | 22222222 |
eeeeeeee ]
f (35f838) :[ 44444444 | 55555555 | bbbbbbbb | cccccccc | 22222222 |
eeeeeeee ]
f (35f838) :[ 44444444 | 66666666 | bbbbbbbb | cccccccc | 22222222 |
eeeeeeee ]
```

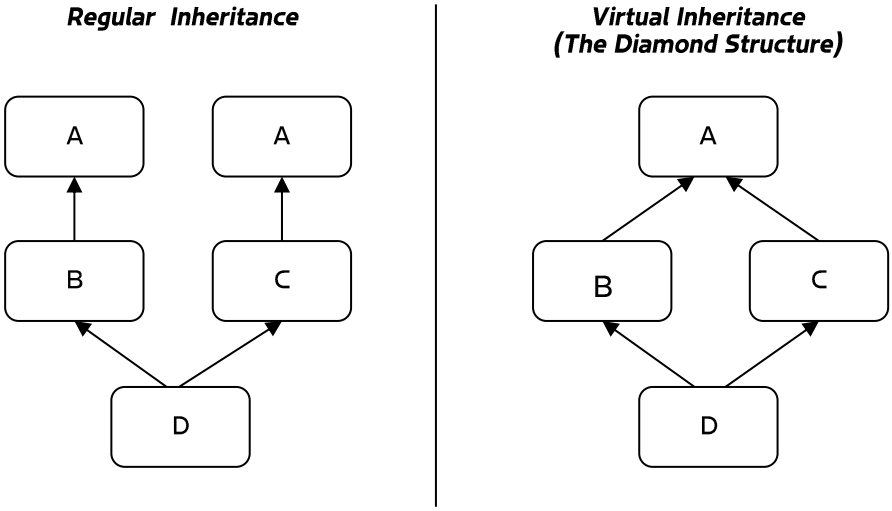
© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הערות

- לאחר כל שינוי הדפסנו את בלוק הזיכרון מחדש.
- השינוי שבוצע בבלוק הזיכרון מודגש בכל שורה (מלבד השורה הראשונה).
- התוכנית מציגה אפשרויות שונות להתייחסויות למשתנה חבר של מחלקה, וכיצד ה-Compiler פותר (Resolves) את כתובות החברים.

2.4 ירושה וירטואלית

כדי לפתור את בעיית הכפילות של חבר הנורש דרך שתי מחלקות בסיס (כמו בדוגמה הקודמת), C++ מספק את מנגנון הירושה הווירטואלית. שימוש בירושה וירטואלית יבטיח מופע אחד של מחלקת הבסיס שממנה יורשים באופן וירטואלי. באופן גרפי אפשר לתאר את הירושה הרגילה והירושה הווירטואלית באופן הבא:



כפי שאפשר לראות על פי התרשים, בירושה וירטואלית האובייקט A יופיע פעם אחת בתוך האובייקט D.

דוגמה

קוד הדוגמה הבא ממחיש לעומק את עניין הירושה הווירטואלית (Ex05):

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
#include <iostream>

using namespace std;

void DumpMemory(const char *name, int *pBase, size_t size)
{
    cout << name << " (" << hex << (int)pBase << ") " << ":[ ";
    if(size > 0)
    {
        cout << hex << *pBase;
    }
    for(int i = 1; i < size / sizeof(int); i++)
    {
        cout << " | " << hex << *(pBase + i);
    }
    cout << " ]" << endl;
}

class A
{
public:
    int a;
    A() { a = 0xAAAAAAAA;}
};

class B : virtual public A
{
public:
    int b;
    B() { b = 0xBBBBBBBB; }
};

class C : virtual public A
{
public:
    int c;
    C() { c = 0xCCCCCCCC; }
};

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

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
int main(int argc, char **argv)
{
    cout << "Inspect B..." << endl;
    B b;
    DumpMemory("b", (int*)&b, sizeof(b));

    cout << "\nInspect C..." << endl;
    C c;
    DumpMemory("c", (int*)&c, sizeof(c));

    cout << "\nInspect D..." << endl;
    D d;
    DumpMemory("d", (int*)&d, sizeof(d));

    d.a = 0x11111111;
    DumpMemory("d", (int*)&d, sizeof(d));

    d.B::a = 0x22222222;
    DumpMemory("d", (int*)&d, sizeof(d));

    d.C::a = 0x33333333;
    DumpMemory("d", (int*)&d, sizeof(d));

    B *pB = &d;
    pB->a = 0x44444444;
    DumpMemory("d", (int*)&d, sizeof(d));
}
```

פלט התוכנית הוא:

```
Inspect B...
b (20fe48) :[ 11a7818 | bbbbbbbb | aaaaaaaa ]

Inspect C...
c (20fe34) :[ 11a7840 | cccccccc | aaaaaaaa ]

Inspect D...
d (20fe18) :[ 11a7970 | bbbbbbbb | 11a78b8 | cccccccc | aaaaaaaa ]
d (20fe18) :[ 11a7970 | bbbbbbbb | 11a78b8 | cccccccc | 11111111 ]
d (20fe18) :[ 11a7970 | bbbbbbbb | 11a78b8 | cccccccc | 22222222 ]
d (20fe18) :[ 11a7970 | bbbbbbbb | 11a78b8 | cccccccc | 33333333 ]
d (20fe18) :[ 11a7970 | bbbbbbbb | 11a78b8 | cccccccc | 44444444 ]
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הערות

- כפי שרואים, בפריסת הזיכרון של B ושל C נוספה כתובת לתחילת הבלוק. כתובת זו היא כתובת לטבלה וירטואלית המסייעת בחישוב כתובות של חברי מחלקת הבסיס הווירטואלית בזמן ריצה. נציין כי במימוש של Visual C++ נעשה שימוש בטבלאות וירטואליות בשני מקרים: כאשר יורשים ממחלקה באמצעות המילה virtual, וכאשר במחלקה מוגדרת לפחות פונקציה וירטואלית אחת. עם זאת, מדובר במנגנונים שונים המותאמים למטרה של כל אחד מהמקרים. בהמשך הספר, בשיעור העוסק בפולימורפיזם, נרחיב בנושא הטבלאות המשמשות להפעלת פונקציות וירטואליות.
- אפשר להבחין בשני הבדלים מהתוכניות הקודמות:
  - כל שינוי ב-a פועל על אותו מקום בזיכרון, בלי קשר לסוג המצביע שבאמצעותו הוא נעשה.
  - a בניגוד לירושה רגילה ממוקם בסוף הבלוק.
  - שני מספרים מוזרים (מסומנים בקו תחתי) מופיעים במקום שבו אמורים היו להיות תאי הזיכרון של B::a ו-C::a.
- שני המספרים המסומנים בקו תחתון מסייעים ל-Compiler להכניס קוד המסייע לחשב את ה-Offset של החבר a כאשר מנסים לגשת אליו באמצעות מצביע מסוג B או מצביע מסוג C.
- להלן קוד האסמבלי ש-Visual Studio יוצר לפעולות ההשמה המודגשות בתוכנית:

```
    B *pB = &d;
011A4609 lea         eax,[d]
011A460C mov         dword ptr [pB],eax
    pB->a = 0x44444444;
011A460F mov         eax,dword ptr [pB]
011A4612 mov         ecx,dword ptr [eax]
011A4614 mov         edx,dword ptr [ecx+4]
011A4617 mov         eax,dword ptr [pB]
011A461A mov         dword ptr [eax+edx],44444444h
```

- הקוד הנ"ל מאתר באמצעות גישה לטבלה הווירטואלית את ההיסט מתחילת הבלוק של D שבו ממוקם המשתנה A.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

2.5 סדר בנייה ופירוק של אובייקטים

בבניית אובייקט, הבנייה מתחילה במחלקה הבסיסית ביותר ומסתיימת במחלקה הנגזרת ביותר. הפירוק נעשה בסדר הפוך.

דוגמה

הדוגמה הבאה מציגה את סדר הפעלת ה-Constructors של אובייקטים, וסדר הפירוק שלהם (Ex06):

```
#include <iostream>

using namespace std;

class A
{
private:
    int a;
public:
    A(int _a)
    {
        cout << "A( a = " << _a << ")" << endl;
        a = _a;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

class B : public A
{
private:
    int b;
public:
    B(int _a, int _b) : A(_a)
    {
        cout << "B( a = " << _a << ", b = " << _b << ")" << endl;
        b = _b;
    }
    ~B()
    {
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
        cout << "~B()" << endl;
    }
};

class C : public A
{
private:
    int c;
public:
    C(int _a, int _c) : A(_a)
    {
        cout << "C( a = " << _a << ", c = " << _c << ")" << endl;
        c = _c;
    }
    ~C()
    {
        cout << "~C()" << endl;
    }
};

class D : public B, public C
{
private:
    int d;
public:
    D(int _a, int _b, int _c, int _d) : B(_a, _b), C(_a, _c)
    {
        cout << "D( a = " << _a << ", b = " << _b << ", c = "
            << _c << ", d = " << _d << ")" << endl;
        d = _d;
    }
    ~D()
    {
        cout << "~D()" << endl;
    }
};

int main(int argc, char **argv)
{
    {
        cout << "Inspect a..." << endl;
        A a(1);
    }
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    {
        cout << "\nInspect b..." << endl;
        B b(2, 3);
    }

    {
        cout << "\nInspect c..." << endl;
        C c(4, 5);
    }

    {
        cout << "\nInspect d..." << endl;
        D d(6, 7, 8, 9);
    }
}
```

פלט התוכנית הוא:

```
Inspect a...
A( a = 1)
~A()

Inspect b...
A( a = 2)
B( a = 2, b = 3)
~B()
~A()

Inspect c...
A( a = 4)
C( a = 4, c = 5)
~C()
~A()

Inspect d...
A( a = 6)
B( a = 6, b = 7)
A( a = 6)
C( a = 6, c = 8)
D( a = 6, b = 7, c = 8, d = 9)
~D()
~C()
~A()
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
~B()
~A()
```

כפי שאפשר לראות, סדר הבנייה הוא מהמחלקה הבסיסית ביותר למחלקה הנגזרת ביותר, וסדר הפירוק הפוך.

נתבונן כעת בבנייה ופירוק של היררכיה הכוללת ירושה וירטואלית (Ex07):

```
#include <iostream>

using namespace std;

class A
{
private:
    int a;
public:
    A(int _a)
    {
        cout << "A( a = " << _a << ")" << endl;
        a = _a;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

class B : virtual public A
{
private:
    int b;
public:
    B(int _a, int _b) : A(_a)
    {
        cout << "B( a = " << _a << ", b = " << _b << ")" << endl;
        b = _b;
    }
    ~B()
    {
        cout << "~B()" << endl;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    }
};

class C : virtual public A
{
private:
    int c;
public:
    C(int _a, int _c) : A(_a)
    {
        cout << "C( a = " << _a << ", c = " << _c << ")" << endl;
        c = _c;
    }
    ~C()
    {
        cout << "~C()" << endl;
    }
};

class D : public B, public C
{
private:
    int d;
public:
    D(int _a, int _b, int _c, int _d) : B(_a, _b), C(_a, _c), A(_a)
    {
        cout << "D( a = " << _a << ", b = " << _b << ", c = " << _c
<< ", d = " << _d << ")" << endl;
        d = _d;
    }
    ~D()
    {
        cout << "~D()" << endl;
    }
};

int main(int argc, char **argv)
{
    {
        cout << "Inspect a..." << endl;
        A a(1);
    }
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    {
        cout << "\nInspect b..." << endl;
        B b(2, 3);
    }

    {
        cout << "\nInspect c..." << endl;
        C c(4, 5);
    }

    {
        cout << "\nInspect d..." << endl;
        D d(6, 7, 8, 9);
    }
}
```

פלט התוכנית הוא:

```
Inspect a...
A( a = 1)
~A()

Inspect b...
A( a = 2)
B( a = 2, b = 3)
~B()
~A()

Inspect c...
A( a = 4)
C( a = 4, c = 5)
~C()
~A()

Inspect d...
A( a = 6)
B( a = 6, b = 7)
C( a = 6, c = 8)
D( a = 6, b = 7, c = 8, d = 9)
~D()
~C()
~B()
~A()
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הערות

- רואים שביצירת האובייקט d, ה-Constructor של A מופעל פעם אחת בלבד.
- ה-Constructor של d חייב להפעיל את ה-Constructor של A, כיוון שה-Compiler מתעלם מאתחול A באמצעות B ו-C. לכן הוספנו ב-Initialization List קריאה ל-Constructor של A.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שיעור 3:
Polymorphism

שים לב: קוד התוכניות של שיעור זה נמצא בספרייה Demos\Chapter 03.

פולימורפיזם (רב-צורתיות) הוא אחת התכונות החשובות ביותר הנתמכות על ידי שפות מונחות עצמים.

3.1 מבוא

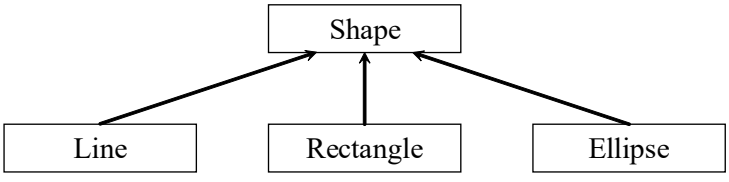
בשיעור הקודם ראינו כי מצביע למחלקת בסיס יכול להצביע על מחלקה נגזרת. במצב זה, כפי שראינו ה-Compiler יקשר לפונקציה על פי המצביע ולא על פי האובייקט שמוצבע בפועל. לעתים זו התנהגות רצויה, אך על פי רוב אין זה כך.

נסביר מהו פולימורפיזם:

- Polymorphism הוא היכולת של אובייקט ללבוש צורות שונות, או במילים אחרות לתפעל את אותו member באופנים שונים.
- Polymorphism נובע מהשימוש בהורשה.

קל יותר להבין מהו Polymorphism באמצעות דוגמה:

נניח שברצוננו ליצור תוכנית גרפית פשוטה המאפשרת ציור של שלוש צורות: Line (קו), Rectangle (מלבן) ו-Ellipse (אליפסה). ברור שנרצה לאחסן את הצורות באמצעות מבנה נתונים מסוים. הבעיה הראשונה שתתעורר היא שלא נוכל להשתמש במבנה נתונים הכולל מצביעים מסוג Line-ים, למשל, לאחסון Rectangle-ים או Ellipse-ות. כדי לפתור בעיה זו נגדיר את כל המחלקות הללו כנגזרות של מחלקה מופשטת בשם Shape. היררכיית המחלקות תיראה, אם כן, כמו בציור הבא (כאשר החץ פירושו "יורש מ..."):



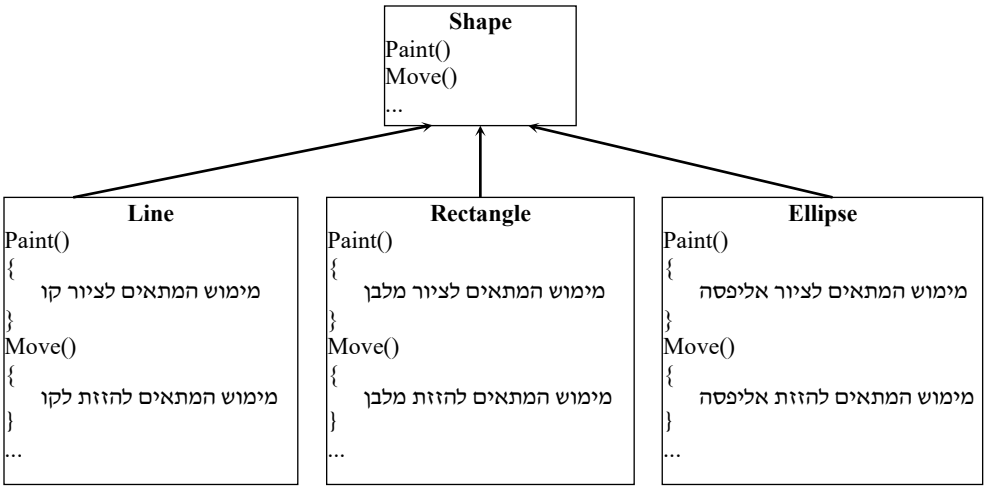
© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

את הבעיה הראשונה פתרנו, כיוון שבשפות מונחות עצמים מצביע למחלקת בסיס (במקרה זה Shape) יכול להצביע לסוגים נגזרים (במקרה זה Line, Rectangle ו-Ellipse). כלומר סוג המצביע במבנה הנתונים יהיה מסוג Shape. הבעיה השנייה שמתעוררת היא, כיצד נבצע סריקה על כל האובייקטים במבנה הנתונים לצורך ביצוע פעולה כלשהי (למשל ציור על המסך), הבעיה נובעת מכך שבפועל בכל תא במבנה יש מצביע לסוג נתונים לא ידוע (Line, Rectangle או Ellipse). בעיה זו אינה בעיה כלל ועיקר, כיוון שנוכל להשתמש במצביע מחלקת בסיס כדי לבצע סריקה במבנה הנתונים ובאמצעות מצביע זה נפעיל את הפונקציה הרצויה. השאלה הנשאלת כעת היא איזו פונקציה בדיוק תופעל, האם זו של המחלקה Shape ממנה מוגדרים המצביעים במבנה הנתונים, או אולי הפונקציה של האובייקט המוצבע בפועל, כאשר זה יכול להיות שונה בין תא לתא במבנה הנתונים. התשובה היא:

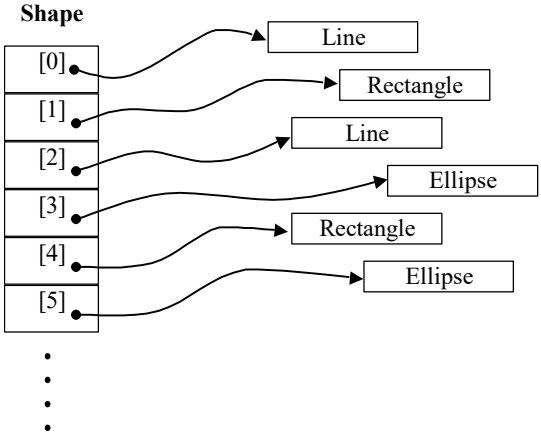
- אם הפונקציה הוגדרה כפונקציה וירטואלית במחלקה Shape (באופן שנראה בהמשך), אז תופעל הפונקציה של האובייקט הקונקרטי (כלומר האובייקט שמאוחסן בפועל) ולא זה של סוג המצביע.
- אם הפונקציה אינה וירטואלית, תופעל הפונקציה של Shape, כלומר, תופעל הפונקציה על פי סוג המצביע.

כדי להשלים את התמונה, וכדי להפוך את העניין למוחשי יותר, נגדיר פונקציות המשותפות לכל הצורות כחלק מהממשק של Shape. פונקציות לדוגמה יהיו Paint, Move. פונקציות אלו עוברות בירושה לכל אחת מהמחלקות הנגזרות (כלומר Line, Rectangle ו-Ellipse) ומקבלות בכל מחלקה משמעות שונה. כך למשל, Paint עבור Line פירושו שימוש בפונקציות מערכת המיועדות לציור קווים על גבי המסך. Paint עבור Rectangle ימומש באמצעות קריאה לפונקציה מערכת המתמחה בציור מלבנים. היררכיית המחלקות, אם כן, תיראה כך:

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il



כעת נניח שברשותנו מערך של Shape-ים שכל תא בו מצביע על צורה מסוג כלשהו:



כעת נוכל להצהיר על משתנה מסוג Shape ולבצע סריקה על המערך. מתוך המצביע הזה נפעיל את הפונקציה Print. מה שיופעל למעשה אלו הן הפונקציות של האובייקטים המוצבעים, ובאופן יותר ספציפי עבור האיבר עם האינדקס [0], תופעל הפונקציה Paint של Line, עבור האיבר עם האינדקס [1] תופעל הפונקציה Print של Rectangle, וכן הלאה.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

3.2 פונקציות וירטואליות

כדי להגדיר פונקציה כווירטואלית יש להשתמש במילה virtual באופן הבא:

```
virtual return-type function-name(parameter-list...);
```

כלומר יש להוסיף את המילה virtual.

שים לב: מחלקות המממשות פונקציות וירטואליות מכנים לעתים מחלקות פולימורפיות.

דוגמה

בדוגמה זו נממש את המתואר במבוא של שיעור זה. ניצור "תוכנית ציור" המגדירה מספר צורות ומוסיפה אותן באופן אקראי למערך מתאים. לאחר מכן נסרוק את כל הרשימה באמצעות מצביע מסוג Shape ונפעיל את הפונקציה המציירת.

להלן קטעי הקוד של הדוגמה (Ex01):

◄ המחלקה Size מאחסנת בתוכה נתוני גודל של צורה. להלן קובץ הכותרת:

```
File: size.h

#ifndef SIZE_H
#define SIZE_H

#include <iostream>

namespace Drawing
{
    class Size
    {
    private:
        int m_cx, m_cy;

    public:
        Size() : m_cx(0), m_cy(0) {}
        Size(int s) : m_cx(s), m_cy(s) {}
        Size(int cx, int cy) : m_cx(cx), m_cy(cy) {}
        friend std::ostream &operator<<(std::ostream &o, const Size
&sz);

    };
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
#endif // SIZE_H
```

להלן קובץ המימוש:

```
File: size.cpp

#include "size.h"

namespace Drawing
{
    std::ostream &operator<<(std::ostream &o, const Size &sz)
    {
        o << "{" << sz.m_cx << ", " << sz.m_cy << "}";
        return o;
    }
}
```

- המחלקה נקודה מייצגת נקודה כלשהי באמצעות שתי קואורדינטות (שמאל, למעלה).

להלן קובץ הכותר:

```
File: point.h

#ifndef POINT_H
#define POINT_H

#include <iostream>
#include "size.h"

namespace Drawing
{
    class Point
    {
    private:
        int m_left;
        int m_top;

    public:
        Point() : m_left(0), m_top(0) {}
        Point(int x) : m_left(x), m_top(x) {}
        Point(int left, int top) : m_left(left), m_top(top) {}
        friend std::ostream &operator<<(std::ostream &o, const Point
&point);
    };
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
}

#endif // POINT_H
```

להלן קובץ המימוש:

```
File: point.cpp

#include "point.h"

namespace Drawing
{
    std::ostream &operator<<(std::ostream &o, const Point &point)
    {
        o << "(" << point.m_left << ", " << point.m_top << ")";
        return o;
    }
}
```

- המחלקה Shape מייצגת הפשטה של צורה כלשהי. במחלקה זו מוגדרות שתי פונקציות וירטואליות. האחת Paint תפקידה הוא לצייר את הצורה (אנו נסתפק בכתיבת שם הצורה למסך ונתוניה), ובנוסף פונקציה וירטואלית MoveTo שתפקידה הוא להזיז את הצורה. להלן קובץ הכותר:

```
File: shape.h

#ifndef SHAPE_H
#define SHAPE_H

#include "point.h"

namespace Drawing
{
    class Shape
    {
    public:
        virtual void Paint()const;
        virtual void MoveTo(const Point &newLocation);
    };
}

#endif // SHAPE_H
```

להלן קובץ המימוש:

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
File: shape.cpp

#include "shape.h"

namespace Drawing
{
    void Shape::Paint()const { /* do nothing */ }
    void Shape::MoveTo(const Point &point) { /* do nothing here */ }
}
```

◄ המחלקה Line מייצגת את הצורה קו. כיוון שקו הוא סוג של צורה, גזרנו את המחלקה מהמחלקה Shape והגדרנו מימושים עבור הפונקציה Paint ועבור הפונקציה MoveTo, בהתאם למה שהיינו מצפים מקו. להלן קובץ הכותר:

```
File: line.h

#ifndef LINE_H
#define LINE_H

#include "shape.h"
#include "point.h"

namespace Drawing
{
    class Line : public Shape
    {
    private:
        Point m_topLeft, m_bottomRight;

    public:
        Line() : m_topLeft(), m_bottomRight() {}
        Line(const Point &topLeft, const Point &bottomRight) :
            m_topLeft(topLeft), m_bottomRight(bottomRight) {}
        Line(int left, int top, int right, int bottom) :
            m_topLeft(left, top), m_bottomRight(right, bottom) {}
        virtual void MoveTo(const Point &newLocation);
        virtual void Paint()const;
    };
}

#endif // LINE_H
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

להלן קובץ המימוש:

```
File: line.cpp

#include <iostream>
#include "line.h"

namespace Drawing
{
    void Line::MoveTo(const Point &newLocation)
    {
        m_topLeft = newLocation;
    }

    void Line::Paint()const
    {
        std::cout << "Line[" << m_topLeft << "; " << m_bottomRight <<
"]\n";
    }
}
```

◄ המחלקה Rectangle מייצגת מלבן. כיוון שמלבן הוא סוג של צורה, גם את מחלקה זו נגזור מ-Shape, ונתאים את המימושים של הפונקציות הווירטואליות כך שיתאימו למלבן. להלן קובץ הכותר:

```
File: rectangle.h

#ifndef RECTANGLE_H
#define RECTANGLE_H

#include <iostream>
#include "shape.h"
#include "point.h"
#include "size.h"

namespace Drawing
{
    class Rectangle : public Shape
    {
    private:
        Point m_location;
        Size m_size;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    public:
        Rectangle() : m_location(), m_size() {}
        Rectangle(const Point &location, const Size &size) :
                m_location(location), m_size(size) {}
        Rectangle(int left, int top, int cx, int cy) :
                m_location(left, top), m_size(cx, cy) {}
        virtual void MoveTo(const Point &newLocation);
        virtual void Paint()const;
    };
}

#endif // RECTANGLE_H
```

להלן קובץ המימוש:

```
File: rectangle.cpp

#include "rectangle.h"

namespace Drawing
{
    void Rectangle::MoveTo(const Point& newLocation)
    {
        m_location = newLocation;
    }

    void Rectangle::Paint()const
    {
        std::cout << "Rectangle[" << m_location << "; " << m_size <<
"]\n";
    }
}
```

◄ גם את המחלקה Circle נגזור מ-Shape ונתאים את המימושים של הפונקציות הווירטואליות, כך שישקפו את הצורה מעגל. להלן קובץ הכותר:

```
File: circle.h

#ifndef CIRCLE_H
#define CIRCLE_H

#include "shape.h"
#include "point.h"
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
namespace Drawing
{
    class Circle : public Shape
    {
    private:
        Point m_location;
        int m_radius;

    public:
        Circle() : m_location(0), m_radius(0) {}
        Circle(int radius) : m_location(), m_radius(radius) {}
        Circle(const Point &location, int radius) :
               m_location(location), m_radius(radius) {}
        Circle(int left, int top, int radius) :
               m_location(left, top), m_radius(radius) {}
        virtual void MoveTo(const Point &newLocation);
        virtual void Paint()const;
    };
}

#endif // CIRCLE_H
```

להלן קובץ המימוש:

```
File: circle.cpp

#include <iostream>
#include "circle.h"

namespace Drawing
{
    void Circle::MoveTo(const Point &newLocation)
    {
        m_location = newLocation;
    }

    void Circle::Paint() const
    {
        std::cout << "Circle[" << m_location << "; " << m_radius <<
"]\n";
    }
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- קובץ הכותר drawing.h נועד לרכז במקום אחד את כל המחלקות שב-namespace: Drawing. להלן תוכנו:

```
File: drawing.h
#ifndef DRAWING_H
#define DRAWING_H

namespace Drawing
{
    class Size;
    class Point;
    class Shape;
    class Line;
    class Rectangle;
    class Circle;
}

#include "size.h"
#include "point.h"
#include "shape.h"
#include "line.h"
#include "rectangle.h"
#include "circle.h"

#endif // DRAWING_H
```

- להלן הפונקציה הראשית Main:

```
File: main.cpp

#include <iostream>
#include "drawing.h"

using namespace std;
using namespace Drawing;

int main(int argc, char **argv)
{
    Line l = Line(2,3,4,5);
    Rectangle r = Rectangle(5,6,7,8);
    Circle c = Circle(20, 30, 5);

    cout << "using pointers...\n";
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    Shape *pShapes[3] = {&l, &r, &c};
    for(int i = 0; i < 3; i++)
    {
        pShapes[i]->Paint();
        pShapes[i]->MoveTo(Point(100,200));
        pShapes[i]->Paint();
    }

    cout << "\n\nusing references...\n";
    Shape &shape1 = l;
    Shape &shape2 = r;
    Shape &shape3 = c;
    shape1.Paint();
    shape2.Paint();
    shape3.Paint();

    cout << "\n\nusing objects...\n";
    Shape shapes[] = {l, r, c};
    for(int i = 0; i < 3; i++)
    {
        shapes[i].Paint();
    }
}
```

פלט התוכנית הוא:

```
using pointers...
Line[(2, 3); (4, 5)]
Line[(100, 200); (4, 5)]
Rectangle[(5, 6); {7, 8}]
Rectangle[(100, 200); {7, 8}]
Circle[(20, 30); 5]
Circle[(100, 200); 5]

using references...
Line[(100, 200); (4, 5)]
Rectangle[(100, 200); {7, 8}]
Circle[(100, 200); 5]

using objects...
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הערות

- כפי שאפשר לראות, הפונקציות Paint ו-MoveTo נקראות על פי האובייקט בפועל למרות שהמצביעים שמהם מתבצעת הקריאה הם מסוג Shape.
- אנו השתמשנו במילה virtual גם בהגדרת המחלקות Line, Circle ו-Rectangle. למעשה, די בהגדרת הפונקציה כוירטואלית במחלקת האב - כלומר Shape, הוירטואליות היא תכונה שנורשת.
- אפשר להפעיל פונקציות וירטואליות גם ישירות באמצעות אובייקט, אולם היתרון העיקרי של מנגנון הפולימורפיזם מתבטא בקריאה באמצעות מצביע או ייחוס למחלקת הבסיס, המאפשרת להפעיל את המימוש של המחלקה הנגזרת בהתאם לסוג האובייקט בפועל.

3.3 מחלקות אבסטרקטיות

מחלקה אבסטרקטית היא מחלקה שבה מוצהרת לפחות פונקציה טהורה אחת.

3.3.1 Pure Function - פונקציה טהורה

פונקציה טהורה היא פונקציה וירטואלית ללא מימוש. חובת המימוש מוטלת על המחלקה היורשת. כדי לסמן פונקציה וירטואלית כטהורה, נשווה אותה ל-0 באופן הבא:

```
class MyAbstractClass
{
    ...
    virtual void SetName(const char *name) = 0;
    ...
};
```

המחלקה MyAbstractClass המתוארת בקוד היא מחלקה אבסטרקטית כיוון שיש לה פונקציה טהורה:

- אי אפשר ליצור מופעים של המחלקה
- מחלקה נגזרת חייבת לממש את SetName (אלא אם היא בעצמה גם אבסטרקטית).

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הערה: ניתן לספק מימוש גם לפונקציה וירטואלית טהורה במחלקת הבסיס, באמצעות הגדרת המימוש מחוץ להגדרת המחלקה. עם זאת, הפונקציה נשארת טהורה והמחלקה נשארת אבסטרקטית, כך שלא ניתן ליצור ממנה מופעים ישירים.

3.3.2 לשם מה מחלקות אבסטרקטיות?

נתאר לעצמנו את קטע הקוד הבא:

```
...
Shape shape;
shape.Paint();
...
```

זוהי המחלקה Shape מהדוגמה הקודמת. מה בדיוק נצייר? ברור שאין משמעות לקטע הקוד והיינו שמחים להימנע ממנו. בתוכנית הקודמת, קטע קוד זה יעבור קומפילציה.

ישנם מקרים שבתהליך ניתוח ועיצוב מחלקות התוכנה מגלים מחלקות המשמשות כמחלקות אב למחלקות אחרות, אך אין להן קיום עצמאי. דוגמה אחת, שבה כבר נתקלת, היא המחלקה Shape. מחלקה זו היא מחלקת אב לכל מחלקה שתייצג צורה. היא מגדירה מהן הפונקציות שעל כל צורה לממש על מנת שתוכלל במערכת. לאובייקטים מסוג המחלקה Shape אין משמעות כיוון שהם מייצגים צורה לא ידועה. מאידך מצביעים מסוג Shape כן רלוונטיים (כפי שראינו בדוגמה הקודמת) כיוון שהם מאפשרים להפעיל את הפונקציה של האובייקט הקונקרטי.

אם כך, אפשר היה לוותר על המימושים של הפונקציות MoveTo ו-Paint במחלקה Shape כיוון שאין להם שום משמעות.

אם כן, נשנה את הגדרת המחלקה Shape כך שזו תהיה מחלקה אבסטרקטית, ובכך לא רק נמנע ייצור אובייקטים מסוג Shape שאינו מוגדר, אלא גם המערכת תאופיין בצורה נכונה יותר.

אם כן, הקוד החדש של המחלקה יהיה:

```
#ifndef SHAPE_H
#define SHAPE_H
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
#include "point.h"

namespace Drawing
{
    class Shape
    {
    public:
        virtual void Paint()const = 0;
        virtual void MoveTo(const Point &newLocation) = 0;
    };
}

#endif // SHAPE_H
```

הערות

- קוד המחלקה כבר אינו כולל קובץ מימוש.
- אם ננסה ליצור מופע של Shape ב-main, נקבל שגיאת קומפילציה.

Virtual Destructor 3.4

כל פונקציות המחלקה, מלבד ה-Constructors והפונקציות הסטטיות, ניתנות להגדרה כפונקציות וירטואליות. אם כן, מה זה בדיוק virtual destructor ולשם מה צריך אותו?

3.4.1 אופן ההגדרה של Virtual Destructor

כדי להגדיר virtual destructor מוסיפים את המילה virtual לפני ה-destructor. למשל:

```
virtual ~Shape(){}
```

3.4.2 לשם מה Virtual Destructor?

virtual destructor מבטיח שבכל מקרה ייקרא ה-destructor של המחלקה הקונקרטית (כלומר ה-destructor של האובייקט בפועל). למה זה חשוב כל כך?

נניח שיש לנו שתי מחלקות, A ו-B, כאשר B יורשת מ-A. כמו שכבר ידוע לנו, מצביע ממחלקה A יכול להצביע על אובייקט מסוג B. מה קורה אם זה המצב וקראנו לאופרטור delete על מנת למחוק את המופע של B באמצעות מצביע מסוג A, כלומר:

```
A *pA = new B(1024);
delete pA;
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

במקרה כזה, כאשר ה-Destructor של מחלקת הבסיס אינו וירטואלי, התנהגות התוכנית אינה מוגדרת (Undefined Behavior). בפועל, בקומפיילרים רבים התוצאה היא הפעלה של ה-Destructor של מחלקת הבסיס A בלבד (ולא לפי הסדר הנורמלי המצופה - קודם של B ואח"כ של A).

עכשיו מה קורה אם ב-Constructor של B מתבצעת הקצאה דינמית של זיכרון, למשל:

```
B::B(int size) { m_buffer = new int[size]; }
```

וב-destructor מתבצע השחרור:

```
B::~B() { delete[] m_buffer; }
```

במצב עניינים כזה, נגרום לזליגת זיכרון - כלומר למצב בו מוקצה באופן דינמי בלוק זיכרון אך אותו בלוק לא משתחרר באופן מסודר.

הגדרת ה-destructor כווירטואלי במחלקה A פותר בעיה זו.

דוגמה א

נמחיש את שאמרנו בדוגמה פשוטה (Ex03):

```
#include <iostream>

using namespace std;

class A
{
public:
    ~A(){ cout << "~A()\n"; }
};

class B : public A
{
public:
    ~B(){ cout << "~B()\n"; }
};

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

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
};

class D : public C
{
public:
    ~D(){ cout << "~D()\n"; }
};


int main(int argc, char **argv)
{
    cout << "new B()\n";
    A *pA = new B();
    delete pA;

    {
        cout << "\nblock B()\n";
        B();
    }

    cout << "\n new D()\n";
    C *pC = new D();
    delete pC;

    {
        cout << "\nblock D()\n";
        D();
    }
}
```

פלט התוכנית הוא:

```
new B()
~A()

block B()
~B()
~A()

 new D()
~D()
~C()

block D()
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
~D()
~C()
```

הערות

- כפי שרואים כאשר מצהירים על משתנה באופן רגיל בבלוק, השחרור מתבצע כשורה: ה-destructor-ים נקראים מהמחלקה הנגזרת מעלה לכיוון מחלקת הבסיס.
- בהקצאה דינמית של אובייקטים, שחרור באמצעות delete ייקרא על פי המצביע. virtual destructor מבטיח קריאה ל-destructor של האובייקט בפועל.

דוגמה ב

בדוגמה הבאה ניצור מעין מיני Framework:

- המחלקה הבסיסית ביותר ב-Framework תהיה המחלקה Object, ועל כל מחלקה להיגזר ממנה.
- על כל מחלקה לממש את הפונקציה ()ToString המחזירה String (שגם הוא מוגדר ב-Framework) המייצג טקסטואלית את האובייקט.
- למחלקה Object ישנה פונקציה ידידה היודעת להדפיס כל אובייקט באמצעות האופרטור >> (פלט), פונקציה זו משתמשת ב-()ToString כדי לקבל את הייצוג הטקסטואלי של האובייקט. באופן כזה אנחנו מרוויחים את האופרטור הזה לכל המחלקות הנגזרות מ-Object. כל שעלינו לעשות הוא לממש את הפונקציה ()ToString בכל מחלקה שתיגזר מ-Object.
- Object מגדיר Virtual Destructor, לפיכך תמיד ייקרא ה-Destructor של האובייקט בפועל.
- בקוד התוכנית הוספנו מעקב אחר מספר ה-Strings שנוצרים ונהרסים. כדי לא לגרור את הקוד המיותר הזה איתנו, הגבלנו אותו לקונפיגורציית _DEBUG.

להלן קוד התוכנית (Ex04):

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

3.4.2.1 המחלקה Object:

```
File: object.h

#ifndef OBJECT_H
#define OBJECT_H

#include <iostream>

class String;

class Object
{
public:
    virtual String ToString()const = 0;
    virtual ~Object() {};
    friend std::ostream &operator<<(std::ostream &o, const Object
&obj);
};

#endif // OBJECT_H
```

```
File: object.cpp

#include "object.h"
#include "string.h"

std::ostream &operator<<(std::ostream &o, const Object &obj)
{
    o << (char *)(&obj)->ToString();
    return o;
}
```

3.4.2.2 המחלקה String:

```
File: string.h

#ifndef STRING_H
#define STRING_H

#include "object.h"
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
class String : public Object
{
private:
    char *m_str;
    String(const char *str, bool copy);
    void CreateEmptyString();
    void CopyFrom(const char *str, bool copy);
    void ClearMemory();

public:
    String();
    String(const char *str);
    String(const String &str);
    String ToString()const;
    const String& operator=(const String &str);
    operator char*();
    String operator+(const String &str)const;
    ~String();

#ifdef _DEBUG
private:
    static int m_stringCouter;
    static int IncrementStringCounter();
    static int DecrementStringCounter();
public:
    static int GetStringCounter();
#endif

};

#endif STRING_H
```

```
File: string.cpp

#include "string.h"
#include <cstring>
#include <cstdlib>

#if _DEBUG

int String::m_stringCouter = 0;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
int String::IncrementStringCounter()
{
    ++m_stringCouter;
    std::cout << "\nstring counter = " << m_stringCouter <<
std::endl;
    return m_stringCouter;
}

int String::DecrementStringCounter()
{
    --m_stringCouter;
    std::cout << "\nstring counter = " << m_stringCouter <<
std::endl;
    return m_stringCouter;
}

int String::GetStringCounter()
{
    return m_stringCouter;
}

#endif // DEBUG

void String::CreateEmptyString()
{
    m_str = new char[1];
    m_str[0] = '\0';
}

void String::CopyFrom(const char *str, bool copy)
{
    if(copy)
    {
        m_str = new char[std::strlen(str) + 1];
        std::strcpy(m_str, str);
    }
    else
    {
        m_str = const_cast<char *>(str);
    }
}

void String::ClearMemory()
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
{
    delete[] m_str;
}

String::String(const char *str, bool copy)
{
#ifdef _DEBUG
    IncrementStringCounter();
#endif // _DEBUG
    if(str == NULL)
    {
        CreateEmptyString();
    }
    else
    {
        CopyFrom(str, false);
    }
}

String::String()
{
#ifdef _DEBUG
    IncrementStringCounter();
#endif // _DEBUG
    CreateEmptyString();
}

String::String(const char *str)
{
#ifdef _DEBUG
    IncrementStringCounter();
#endif // _DEBUG
    CopyFrom(str, true);
}

String::String(const String &str)
{
#ifdef _DEBUG
    IncrementStringCounter();
#endif // _DEBUG
    CopyFrom(str.m_str, true);
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
String String::ToString()const
{
    return *this;
}

String String::operator+(const String &str)const
{
    char *tmp = new char[strlen(m_str) + strlen(str.m_str) + 1];
    tmp[0] = '\0';
    strcat(tmp, m_str);
    strcat(tmp, str.m_str);
    return String(tmp, false);
}

const String &String::operator=(const String &str)
{
    if(this == &str) // self assinment
        return *this;

    ClearMemory();
    CopyFrom(str.m_str, true);
    return *this;
}

String::operator char*()
{
    return m_str;
}

String::~String()
{
#ifdef _DEBUG
    DecrementStringCounter();
#endif
    ClearMemory();
}
```

הערות

- נשים לב כי הפונקציות CreateEmptyString() וה-CopyFrom() הן למעשה הפונקציות שמאתחלות את ה-String, והן נקראות על ידי כל ה-Constructors.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- הפונקציה ClearMemory() משחררת את הזיכרון הדינמי שהוקצה כאשר ה-String() נהרס.
- האופרטור + משרשר שתי מחרוזות. כיוון שהמחרוזת המשורשרת לא שייכת בפועל לאף אובייקט String, פונקציית האופרטור קוראת לפונקציה CopyFrom ומעבירה את הערך false עם הפרמטר copy. באופן כזה, הפונקציה CopyFrom משייכת את המחרוזת ל-String במקום לשכפל פעם שנייה את אותו String, דבר שהוא למעשה מיותר.

3.4.2.3 המחלקה Worker

```
File: worker.h

#ifndef WORKER_H
#define WORKER_H

#include "object.h"
#include "string.h"

class Worker : public Object
{
private:
    String m_name, m_phone;

public:
    Worker(const char *name, const char *phone) : m_name(name),
m_phone(phone) {}
    Worker(const String &name, const String &phone) : m_name(name),
m_phone(phone) {}
    String ToString()const;
};

#endif // WORKER_H
```

```
File: worker.cpp

#include "worker.h"

String Worker::ToString()const
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
{
    return m_name + ": " + m_phone;
}
```

הערות

- המחלקה Worker מממשת את הפונקציה ()ToString ובאופן כזה היא מרוויחה את האופרטור >> הממומש במחלקה Object.
- המחלקה Object מנצלת את המחלקה String שהוגדרה כבר על מנת לשרשר מחרוזות בקוד יותר אינטואיטיבי.

3.4.2.4 התוכנית הראשית

```
File: main.cpp

#include <iostream>
#include "object.h"
#include "string.h"
#include "worker.h"

int main(int argc, char **argv)
{
    Object *pObj = new String("Hello world!");
    std::cout << *pObj << std::endl;

    Object *pWorker = new Worker("Ilan Amoyal", "12345678");
    std::cout << *pWorker << std::endl;

    delete pObj;
    delete pWorker;
}
```

3.4.2.5 פלט התוכנית

כאשר מריצים את התוכנית בקונפיגורציית Debug ב-Visual Studio, מתקבל הפלט הבא:

```
string counter = 1
string counter = 2
Hello world!
string counter = 1
string counter = 2
string counter = 3
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
string counter = 4
string counter = 5
string counter = 6
string counter = 5
string counter = 4
Ilan Amoyal: 12345678
string counter = 3
string counter = 2
string counter = 1
string counter = 0
```

אם מריצים את התוכנית בקונפיגורציית Release מתקבל הפלט הבא:

```
Hello world!
Ilan Amoyal: 12345678
```

3.6 Static Binding vs. Dynamic Binding

מנגנון הפונקציות הווירטואליות הוא מנגנון חזק, אך צריך לדעת כיצד הוא פועל ומה המחיר.

3.6.1 Static Binding

עבור פונקציות שאינן וירטואליות, ה-Compiler קובע בזמן הקומפילציה איזו פונקציה תופעל, לפי הסוג הסטטי של המצביע (או הייחוס). זו הסיבה ש-Static Binding מהיר מאוד, כיוון שאין צורך להחליט איזה קוד יש להריץ בזמן ריצת התוכנית.

3.6.2 Dynamic Binding (Late Binding)

ב-Dynamic Binding ההחלטה על הכתובת של הפונקציה הווירטואלית שאותה יש להפעיל מתבצעת בזמן ריצת התוכנית. כל מה שה-Compiler מבצע הוא רק להכין את הקרקע לכך שבמהלך ריצת התוכנית, אכן אפשר יהיה להחליט מהי הפונקציה שתופעל. עבור הפעלה של פונקציות וירטואליות, ה-Compiler בונה מנגנון מתוחכם המבוסס על טבלת מצביעים לפונקציות. כיצד?

שלב א: מחלקת הבסיס הראשונה בהיררכיה בה מוגדרת פונקציה וירטואלית, גורמת ל-Compiler ליצור טבלת סטטית של מצביעים לפונקציות (כזו שתשמש את כל האובייקטים ממחלקה זו). בטבלה זו מספר הכניסות (האיברים) הוא כמספר הפונקציות

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

הווירטואליות המוגדרות במחלקה. כל כניסה מצביעה על פונקציה וירטואלית אחת. אנו נכנה טבלה זו בשם [vftable], היינו Virtual Function Table.

שלב ב: לכל אובייקט מהמחלקה ה-Compiler מוסיף משתנה חבר נוסף (שמוסתר מאיתנו), שנקרא לו [vftable-pointer], היינו Virtual Function Table Pointer, חבר זה יצביע בהמשך לטבלה [vftable].

שלב ג: ב-Constructor של המחלקה, ה-Compiler מוסיף פקודה המאתחלת את [vftable-pointer] כך שיצביע לטבלה הסטטית של המחלקה כולה [vftable].

שלב ד: עבור כל מחלקה הנגזרת מהמחלקה הבסיסית, ה-Compiler חוזר על שלבים א ו-ג'. כלומר, הוא יוצר טבלה [vftable] עבור המחלקה החדשה, ובה כתובות הפונקציות הווירטואליות. את ה-[vftable-pointer] שאותו המחלקה ירשה ממחלקת הבסיס, ה-Compiler יאתחל לפני תחילת ביצוע גוף ה-Constructor של המחלקה הנגזרת.

שלב ה: כל קריאה לפונקציה וירטואלית, מוחלפת על ידי ה-Compiler בהפעלה עקיפה של הפונקציה שכתובתה נמצאת בכניסה המתאימה בטבלת הפונקציות הווירטואליות. את הכתובת לטבלה ה-Compiler לוקח מהמשתנה [vftable-pointer] אותו הוא עצמו יצר.

שלב ו: בזמן ריצת התוכנית, כל בנאי מאתחל באופן מתאים את מצביע הפונקציות הווירטואליות שלו, באופן כזה כל הקריאות לפונקציות וירטואליות שקושרו על ידי ה-Compiler, יפעילו את הפונקציה המתאימה.

דוגמה

נניח שיש ברשותנו שתי מחלקות, Base ו-Derived, המוגדרות כדלקמן:

```
class Base
{
public:
    void Function_01() {};
    void Function_02() {};
    virtual void Function_03() {};
    virtual void Function_04() {};
    virtual void Function_05() { };
};
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

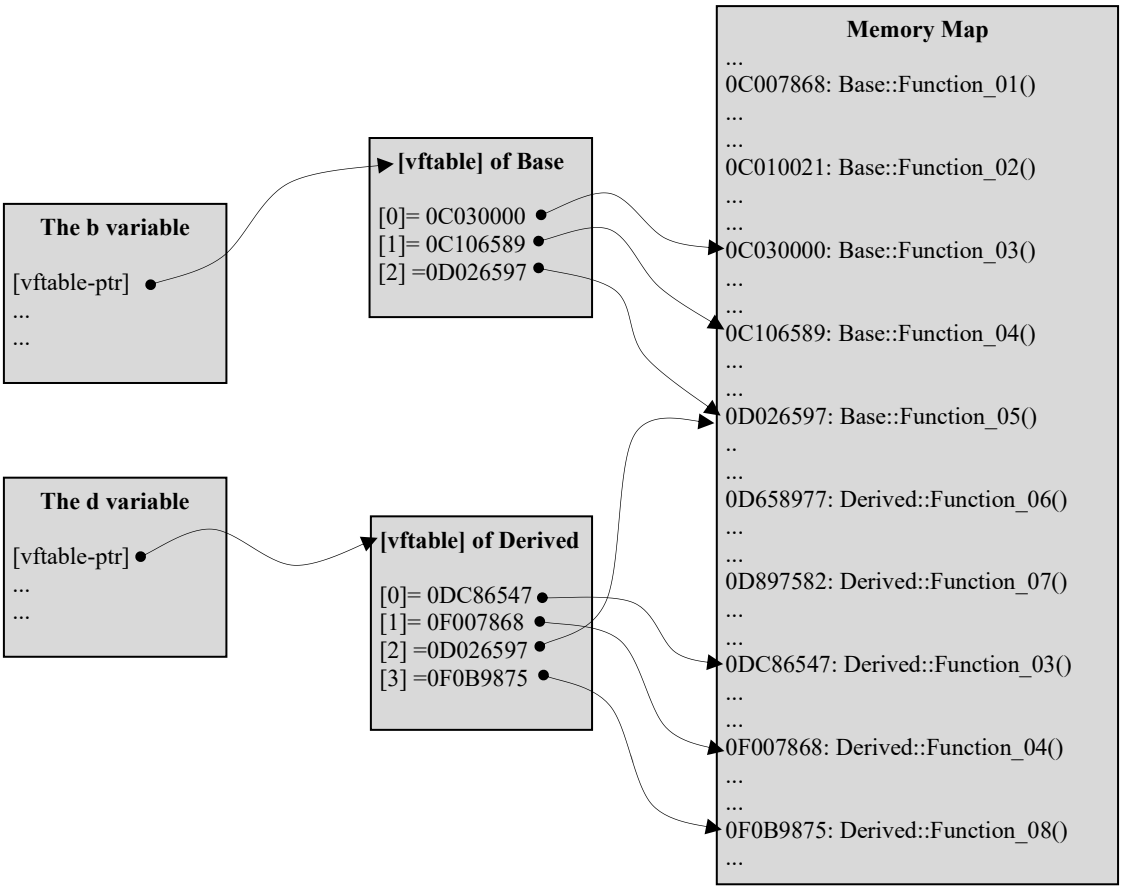
```
class Derived : public Base
{
public:
    void Function_06() {};
    void Function_07() {}
    virtual void Function_03() { } // overrides base implementation
    virtual void Function_04() { } // overrides base implementation
    virtual void Function_08() { } // new virtual function
};

void main()
{
    Base b;
    Derived d;
}
```

כפי שרואים, Derived דורס מימוש של שתי פונקציות וירטואליות (Function_03 ו-Function_04), ומוסיף פונקציה וירטואלית אחת (Function_08).

תמונת הזיכרון תיראה בערך כך:

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il



3.6.3 ביצועים

אם כן, נובע כי Static Binding מהיר יותר וחסכוני יותר מבחינת מקום. צריך לזכור שאם מגדירים פונקציה וירטואלית אחת במחלקת בסיס כלשהי, עבור אותה מחלקה - ועבור כל נגזרותיה, ייווצרו טבלאות לפונקציות וירטואליות. בנוסף, לכל אובייקט מההיררכיה הזו יתוסף מצביע בגודל Word (2, 4 או 8 בתים - בהתאם לסביבה), אשר יצביע על הטבלה הווירטואלית השייכת לאותו אובייקט. בהתקנים בהם חסכון בזיכרון הוא עניין מהותי, ייתכן שזה יהווה שיקול מרכזי. בכל אופן, ברוב היישומים הרגילים (למעט אלה שהם Real Time

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

או כאלה שרצים על התקנים זעירים הקצרים בזיכרון), ביצועי פונקציות וירטואליות הם מעולים ולא מהווים שום בעיה, הן מבחינת מקום והן מבחינת מהירות.

3.7 הגדרת Interfaces

Interface הוא ממשק פונקציות שמחלקה מממשת. ה-Interface עצמו אינו מממש אף אחת מהפונקציות של עצמו, אלא רק מגדיר פונקציונליות. בדרך כלל מגדירים Interface כשרוצים לאפשר לקוד חיצוני לקבל שירותים ממחלקה או לספק לה שירותים מסוימים. על מנת להבטיח שהמחלקה אכן מספקת תמיכה בשירותים הללו, מגדירים את השירותים בתוך Interface ואז כל מחלקה שמממשת את הפונקציות שהוגדרו ב-Interface תוכל לפעול באינטראקציה עם הקוד.

דוגמה

בשפת C# אפשר להגדיר interface באופן הבא:

```
public interface PlayerMixer
{
    int Version { get; }
    void Mix(Stream s);
}
```

כאן למשל, הגדרנו interface אותו אמורים לממש Plugins המתיימרים לספק שירותי Mix ל-Player. ה-Player יטען את אותם שמתיימרים להיות Plugins, יבדוק שאכן הם מממשים את ה-Interface, ואם זה המצב - יפעיל את הפונקציונליות שלהם במידת הצורך.

נהוג שב-Interfaces כל הפונקציות הן ציבוריות. שפת C++ אינה תומכת באופן ישיר ב-Interfaces ולכן הדרך שלנו להשיג את הדבר הדומה ביותר ל-Interface היא באמצעות Abstract Class שבו כל הפונקציות טהורות ומוגדר בו Virtual Destructor.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

דוגמה א

```
class IComparable
{
public:
    virtual int CompareTo(const Object &obj) const = 0;
    ~virtual IComparable() {}
};
```

בדוגמה זו הגדרנו מחלקה אבסטרקטית הדורשת מימוש של פונקציה אחת CompareTo. כמו כן הגדרנו Virtual Destructor על מנת להבטיח פירוק נכון במקרה שמחלקה יורשת מנהלת משאבים המחייבים שחרור מסודר.

דוגמה ב

נרחיב מעט ונשנה את הדוגמה הקודמת, כך שתתאפשר תמיכה ב-IComparable שתואר לעיל. להלן נציג רק את קטעי הקוד שנוספו, ונסביר אותם (Ex05):

```
File: __baseObject

#ifndef BASE_OBJECT_H
#define BASE_OBJECT_H

class __baseObject
{
protected:
    virtual void __dummyMethod(){};
};

#endif // BASE_OBJECT_H
```

הערות

- מחלקה זו תהיה מחלקת הבסיס החדשה לכל הסביבה. מטרתה של מחלקה זו היא לאפשר קשר בין מחלקות שאינן נגזרות מ-Object. למשל, IComparable לא נגזר מ-Object ו-Object לא נגזר ממנו. לפיכך כדי להשיג מצביע ממחלקה Object ל-IComparable נהיה חייבים לעבור דרך מחלקה משותפת. (מעבר דרך * void לא יפעל היטב).

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- יצרנו פונקציה וירטואלית שאין בה שום צורך, אלא רק להבטיח שההיררכיה תהיה פולימורפית.

```
File: IComparable.h

#ifndef ICOMPARABLE_H
#define ICOMPARABLE_H

#include "object.h"
#include "base_object.h"

class IComparable : virtual public __baseObject
{
public:
    virtual int CompareTo(const Object &obj)const = 0;
    virtual ~IComparable(){}
};

#endif // ICOMPARABLE_H
```

הערות

- זהו ה-Interface שהוספנו לפרויקט.
- מחלקה המעוניינת לתת תמיכה בהשוואה, כל שעליה לעשות הוא לממש את IComparable.

```
File: object.h

#ifndef OBJECT_H
#define OBJECT_H

#include <iostream>
#include "base_object.h"

class String;
class IComparable;

class Object : virtual public __baseObject
{
private:
    template<typename T>
    T *GetInterface(T *dummy)const;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
public:
    virtual String ToString()const = 0;
    virtual ~Object() {};

    bool operator<(const Object &obj)const;
    bool operator<=(const Object &obj)const;
    bool operator==(const Object &obj)const;
    bool operator>=(const Object &obj)const;
    bool operator>(const Object &obj)const;
    bool operator!=(const Object &obj)const;

    friend std::ostream &operator<<(std::ostream &o, const Object
&obj);
};

#endif // OBJECT_H
```

הערות

- עכשיו המחלקה Object נגזרת מ-__baseObject. דרך מחלקה זו יצרנו "קשר משפחתי" בין Object לבין מחלקות אחרות שאיננו מעוניינים לגזור מ-Object, אותן נגזור מ-__baseObject. כך למשל Interfaces.
- אופרטורי ההשוואה ממומשים במחלקה Object ועושים שימוש ב-IComparable (אם הוא נמצא). להלן קוד המימוש:

```
File: object.cpp

#include "object.h"
#include "string.h"

template<typename T>
T *Object::GetInterface(T *dummy)const
{
    try
    {
        return dynamic_cast<T*>(

    dynamic_cast<__baseObject*>(const_cast<Object*>(this)));
    }
    catch(std::bad_cast &bd)
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    {
        return NULL;
    }
    catch(...)
    {
        return NULL;
    }
}

bool Object::operator<(const Object &obj)const
{
    IComparable *pComparable = GetInterface<IComparable>(NULL);
    if(pComparable)
    {
        return pComparable->CompareTo(obj) < 0;
    }
    throw "Not implemented";
}

bool Object::operator<=(const Object &obj)const
{
    IComparable *pComparable = GetInterface<IComparable>(NULL);
    if(pComparable)
    {
        return pComparable->CompareTo(obj)<= 0;
    }
    throw "Not implemented";
}

bool Object::operator==(const Object &obj)const
{
    IComparable *pComparable = GetInterface<IComparable>(NULL);
    if(pComparable)
    {
        return pComparable->CompareTo(obj)== 0;
    }
    throw "Not implemented";
}

bool Object::operator>=(const Object &obj)const
{
    IComparable *pComparable = GetInterface<IComparable>(NULL);
    if(pComparable)
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    {
        return pComparable->CompareTo(obj)>= 0;
    }
    throw "Not implemented";
}

bool Object::operator>(const Object &obj)const
{
    IComparable *pComparable = GetInterface<IComparable>(NULL);
    if(pComparable)
    {
        return pComparable->CompareTo(obj)> 0;
    }
    throw "Not implemented";
}

bool Object::operator!=(const Object &obj)const
{
    IComparable *pComparable = GetInterface<IComparable>(NULL);
    if(pComparable)
    {
        return pComparable->CompareTo(obj)!= 0;
    }
    throw "Not implemented";
}

std::ostream &operator<<(std::ostream &o, const Object &obj)
{
    o << (char *)(&obj)->ToString();
    return o;
}
```

הערות

- הפונקציה GetInterface משיגה מצביע ל-Interface המבוקש. פונקציה זו עושה שימוש בשתי טכניקות:
  - RTTI ו-Casting: טכניקה המאפשרת זיהוי סוג בזמן ריצה וביצוע המרות מסוג לסוג.
  - Templates: טכניקה המאפשרת לכתוב קוד גנרי שאינו צמוד למחלקה מסוימת.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שתי טכניקות אלה נלמדות בהמשך הספר.

- מימושי האופרטורים, לאחר שהשיגו מצביע ל-IComparable, משווים את האובייקט עם האובייקט שהועבר, ועל פי הערך המוחזר קובעים את תוצאת ההשוואה.

```
File: string.h

#include "icomparable.h"

class String : public Object, virtual public IComparable
{
...
...
/* IComparable members */
public:
    int CompareTo(const Object &obj)const;
...
}
```

הערות

- כאן מוצגות רק השורות שנוספו ל-string.h
- כל שנוסף זה הצהרת ירושה מ-IComparable, והצהרת הפונקציה שיש לממש.

```
File: string.cpp

...
int String::CompareTo(const Object &obj) const
{
    return strcmp(this->m_str, obj.ToString().m_str);
}
...
```

הערות

- זהו קוד הפונקציה שנוסף למחלקה String
- הפונקציה משתמשת ב-strcmp על מנת לבצע את ההשוואה בין המחרוזת הנוכחית לבין הייצוג הטקסטואלי של האובייקט שהועבר אליה.

```
File: main.cpp

#include <iostream>
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
#include "object.h"
#include "string.h"
#include "worker.h"

int main(int argc, char **argv)
{
    Object *pObj = new String("Hello world!");
    std::cout << *pObj << std::endl;

    Object *pWorker = new Worker("Ilan Amoyal", "12345678");
    std::cout << *pWorker << std::endl;

    if(*pObj > String("Hello"))
    {
        std::cout << "yes" << std::endl;
    }

    delete pObj;
    delete pWorker;
}
```

פלט התוכנית הוא:

```
Hello world!
Ilan Amoyal: 12345678
yes
```

הערות

- בשורות הכהות יותר, רואים שקל לבצע השוואה בין שתי מחרוזות.
- כל זה מבוצע בתיווכו של Object המפעיל את הפונקציה CompareTo של IComparable הממומשת במחלקה string.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שיעור 4:
נושאים נוספים

שים לב: קוד התוכניות של שיעור זה נמצא בספרייה Demos\Chapter 04.

4.1 משתנים מסוג mutable

לעתים נרצה שפונקציית const מסוימת במחלקה תוכל לשנות משתנה חבר. למרות שפעולה כזו חורגת מההגדרה של פונקציית const, לעתים יש צורך באפשרות הזו.

דוגמה

נניח שיש ברשותנו מחלקה המייצגת Collection (אוסף) כלשהו. שינוי ה-Collection פירושו הוספה, הסרה, או שינוי מיקום של אחד האלמנטים ב-Collection. עם זאת, ברור כי סריקת המערך אינה משנה את תוכן ה-Collection. כעת, אם ברצוננו להגדיר מספר פונקציות שיאפשרו איטרציה על איברי ה-Collection, ודאי נרצה, למשל, שהפונקציה ()next תקדם את האובייקט הנוכחי (current) וכמו כן תשנה את השדה הפנימי ב-Collection המחזיק את האיבר הנוכחי. כמו כן ברור כי פונקציה זו אמורה להיות פונקציית const כיוון שאינה משנה את תוכן ה-Collection. כדי שנוכל לשנות, מתוך פונקציית const, את השדה המחזיק באיבר הנוכחי באיטרציה, נגדיר איבר זה באמצעות המילה mutable.

הקוד שלהלן מדגים מימוש של Stack המשתמש בתכונה זו כדי לאפשר איטרציה על ה-Stack (Ex01):

```
#include <iostream>
#include <cstring>

using namespace std;

class Stack
{
private:
    int *m_data;
    size_t m_size;
    size_t m_actualSize;
    mutable int m_currentIteratorIndex;
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
public:
    Stack()
    {
        m_data = NULL;
        m_size = 0;
        m_currentIteratorIndex = -1;
        m_actualSize = 0;
    }

    void Push(int value)
    {
        if(m_size == m_actualSize)
        {
            // reallocated the block
            int *newData = new
                int[m_actualSize = (m_actualSize == 0 ? 2 :
m_actualSize * 2)];
            if(m_data != NULL)
            {
                memcpy(newData, m_data, m_size * sizeof(int));
                delete[] m_data;
            }
            m_data = newData;
        }
        m_data[m_size++] = value;
    }

    int Pop()
    {
        if(m_size > 0)
        {
            return m_data[--m_size];
        }
        else
        {
            throw "Invalid operation";
        }
    }

    /* Iteration functions */
    void Start()const { m_currentIteratorIndex = -1; }
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    bool Next()const
    {
        if(m_currentIteratorIndex < (int)m_size)
        {
            m_currentIteratorIndex++;
        }
        return m_currentIteratorIndex < (int)m_size;
    }

    bool Previous()const
    {
        if(m_currentIteratorIndex >= 0)
        {
            m_currentIteratorIndex--;
        }
        return m_currentIteratorIndex >= 0;
    }

    int Current()const
    {
        return *(m_data + m_currentIteratorIndex);
    }

    ~Stack()
    {
        if(m_data != NULL)
        {
            delete[] m_data;
        }
    }
};

void DisplayStack(const Stack &stack)
{
    stack.Start();
    while(stack.Next())
    {
        cout << stack.Current() << " ";
    }
    cout << endl;
}

int main(int argc, char **argv)
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
{
    Stack stack;
    stack.Push(2);
    stack.Push(6);
    stack.Push(-9);
    stack.Push(13);
    stack.Push(-14);
    stack.Push(104);
    DisplayStack(stack);
}
```

כפי שאפשר לראות, הפונקציה DisplayStack מקבלת אובייקט const Stack ומפעילה פונקציות const שלו. פונקציות האיטרציה למרות היותן פונקציות const יכולות לשנות את ערכו של m_currentIteratorIndex כיוון שהוא מוגדר באמצעות המילה mutable.

הערה: המחלקה Stack מנהלת זיכרון דינמי, אך לצורך פשטות הדוגמה לא הוגדרו בה Copy Constructor ואופרטור השמה. לכן, העתקת אובייקט מסוג Stack תגרום להעתקה רדודה של המצביע לזיכרון, כך ששני אובייקטים יתייחסו לאותו בלוק זיכרון ועלולים לנסות לשחררו פעמיים. במימוש מלא יש לדאוג להעתקה תקינה של הנתונים.

4.2 האופרטור new

4.2.1 זריקת exception

תקן C++ קובע כי במקרה של הקצאת זיכרון דינמית שנכשלת, יש לזרוק exception מסוג bad_alloc. אם כן, כדי לטפל במצב שגיאה זה, נכתוב את הקוד הבא:

```
#include <new>
...
...
...
try
{
    char *str = new char[10];
}
catch(bad_alloc &ba)
{
    cerr << "Out of memory..." << endl;
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

4.2.2 הקבוע nothrow

כאמור, הקצאת זיכרון דינמית ב-C++ באמצעות האופרטור new זורקת Exception במקרה של כישלון. בשפת C הקצאה כושלת מחזירה מצביע NULL, דבר שהשתמר ב-C++ בגרסתו הראשונה, אך התקן העדכני קובע שיש לזרוק Exception. לעתים תרצה לעבוד בסגנון הישן, היינו לקבל מצביע NULL. במקרה כזה יש להעביר את הקבוע **nothrow** לאופרטור new באופן הבא:

```
new(nothrow) type[size]
```

דוגמה

```
char *str = new(nothrow) char[128];
if(str == NULL)
{
    cerr << "Error: Out of memory" << endl;
}
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שיעור 5:
Templates

שים לב: קוד התוכניות של שיעור זה נמצא בספרייה Demos\Chapter 05.

5.1 פונקציות תבנית

נתבונן בפונקציה swap שלפנינו, המחליפה בין ערכי שני המשתנים השלמים המועברים אליה:

```
void swap(int *n, int *m)
{
    int tmp = *n;
    *n = *m;
    *m = tmp;
}
```

החיסרון שבמימוש זה הוא בכך שהוא מוגבל למשתנים מסוג Integer בלבד. מה אם נרצה ליצור פונקציית swap עבור משתנים מסוג double למשל, או פונקציית swap עבור אובייקטים, וכן הלאה.

כמובן שאפשר ליצור פקודת (#define) מאקרו שתעשה זאת, אך לפקודות מאקרו מספר חסרונות:

- לא ניתן לעבור עם ה-Debugger על פקודות מאקרו
- לא ניתן לבדוק את סוגי הנתונים בזמן העברתם לפקודות מאקרו.
- בפקודות מאקרו לעתים משכפלים קטעי קוד שלא רוצים לשכפל (כך למשל - הגדלה ב-1).

הצורה בה מגדירים פונקציית תבנית היא:

```
template <class T> function definition...
```

במקרה זה הביטוי template<class T> מגדיר את T כסוג נתונים גנרי שנקבע בעת השימוש בפונקציה. נשים לב אפשר לכתוב את אותה ההצהרה גם באופן הבא:

```
template<typename T> function definition
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פה השתמשנו במילה typename במקום class. זה דבר אפשרי, אם כי למילה השמורה typename יש שימוש ייחודי נוסף כפי שנראה בהמשך.

דוגמה א

נמיר את קטע הקוד הנ"ל לפונקציית תבנית באופן הבא (Ex01):

```
#include <iostream>

using namespace std;

template<class T>
void Swap(T &obj1, T &obj2)
{
    T tmp = obj1;
    obj1 = obj2;
    obj2 = tmp;
}

int main(int argc, char **argv)
{
    int a = 7, b = 9;
    Swap<int>(a, b);
    cout << "a = " << a << ", b = " << b << endl;

    double d1 = 2.0, d2 = 6.7;
    Swap<double>(d1, d2);
    cout << "d1 = " << d1 << ", d2 = " << d2 << endl;
}
```

פלט התוכנית הוא:

```
a = 9, b = 7
d1 = 6.7, d2 = 2
```

כפי שאפשר לראות, מגדירים פונקציה בשם swap המקבלת סוג כללי של פרמטר. לאחר מכן קוראים לפונקציה זו עם פרמטרים מסוגים שונים. ה-Compiler מזהה קריאות אלו, ויוצר קטעי קוד עבור כל אחד מהמימושים הנדרשים (במקרה שלנו מימוש עבור int ומימוש עבור double).

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

דוגמה ב

נכתוב פונקציית תבנית המשווה בין שני אובייקטים ומחזירה מספר חיובי אם הראשון מביניהם הוא הגדול, מספר שלילי אם השני הוא הגדול ו-0 אם שניהם שווים (Demos\Chapter 07\Ex02):

```
#include <iostream>

using namespace std;

template<typename T>
int Compare(const T obj1, const T obj2)
{
    return obj1 > obj2 ? 1 : (obj1 == obj2 ? 0 : -1);
}

int main(int argc, char **argv)
{
    int a = 2, b = 6;
    cout << "Compare(a, b) = " << Compare(a, b) << endl;
    cout << "Compare(b, a) = " << Compare(b, a) << endl;

    const char *str1 = "aaa";
    const char *str2 = "bbb";
    cout << "Compare(str1, str2) = " << Compare(str1, str2) << endl;
}
```

פלט התוכנית הוא:

```
Compare(a, b) = -1
Compare(b, a) = 1
Compare(str1, str2) = 1
```

פלט התוכנית מראה שבוצעה השוואה נכונה בין שני מספרים מסוג int, אך השוואת המחרוזות נכשלה. הסיבה היא שההשוואה בוצעה על ערכי המצביעים ולא על תוכן המחרוזות עצמן. כלומר ההשוואה תלויה במיקום בו ה-Compiler מיקם את שתי המחרוזות הללו. על מנת לפתור בעיות מעין אלה, כלומר מצב בו ההגדרה הכללית טובה לסוג רחב של סוגי נתונים אך אינה מתאימה לסוגים ספציפיים, C++ מאפשרת לנו להגדיר Specializations (התמחויות) של פונקציות תבנית.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

Template Specializations 5.2

כדי לפתור את הבעיה שהוצגה, נוסיף לתוכנית פונקציה המטפלת בדיוק במקרה של פרמטר מסוג char *, באופן הבא (Ex03):

```
template<>
int Compare(const char *str1, const char *str2)
{
    return strcmp(str1, str2);
}
```

בקטע קוד זה הגדרנו בעצמנו את המימוש של פונקציית התבנית למקרה שסוג הנתונים הוא char *.

Overloaded Functions vs. Specialized Functions 5.2.1

יש להבחין בין Function Overloading לבין Function Specialization. כאשר קיימות פונקציות רגילות ופונקציות תבנית בעלות אותו שם, ה-Compiler בוחר את הפונקציה המתאימה ביותר על פי סוגי הפרמטרים. במקרה של התאמה שווה, ניתנת עדיפות לפונקציה רגילה. עם זאת, כאשר מציינים במפורש את סוג הנתונים באמצעות סוגריים זוויתיים, הבחירה מוגבלת לפונקציות תבנית ולהתמחויות שלהן. קטע הקוד הבא ממחיש זאת (Ex04):

```
#include <iostream>

using namespace std;

template<class T>
void DoSomething(T t)
{
    cout << "DoSomething<" << typeid(T).name() << ">() is called...t
= " << t << endl;
}

void DoSomething(double t)
{
    cout << "oveloaded DoSomething<double> is called...t = " << t <<
endl;
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
template<>
void DoSomething(int t)
{
    cout << "specialized DoSomething<int> is called...t = " << t <<
endl;
}

void DoSomething(int t)
{
    cout << "overloaded DoSomething<int> is called...t = " << t <<
endl;
}


int main(int argc, char **argv)
{
    DoSomething(1.1);
    DoSomething<double>(1.2);
    DoSomething(2);
    DoSomething<int>(3);
    DoSomething((float)4);
}
```

פלט תוכנית הוא:

```
oveloaded DOSOMETHING<double> is called...t = 1.1
DoSomthing<double>() is called...t = 1.2
overloaded DoSomthing<int> is called...t = 2
specialized DoSomthing<int> is called...t = 3
DoSomthing<float>() is called...t = 4
```

כפי שאפשר להבחין:

- פונקציה שנקראת ללא ציון מפורש של סוג הנתונים בסוגריים זוויתיים, תופעל על ידי הפונקציה המתאימה ביותר מבחינת סוגי הפרמטרים. אם קיימת התאמה שווה בין overloaded function לבין template function, תינתן עדיפות לפונקציה הרגילה. אם נבחרה פונקציית תבנית, תיבדק האפשרות להשתמש ב-specialized function המתאימה לה. אם אין התמחות מתאימה, ה-Compiler ייצור בעצמו מופע של פונקציית התבנית (אם הוא לא יצר כבר) עבור סוג הנתונים שבקריאה.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- אם פונקציה נקראת באמצעות סוגריים זוויתיים המציינים את סוג הנתונים, ה-Compiler יבחר בפונקציית תבנית מתאימה, גם אם קיימת פונקציה רגילה בעלת אותו שם (Overloaded Function). אם קיימת התמחות (Specialization) מתאימה של התבנית שנבחרה, ייעשה בה שימוש.

Template Classes 5.3

באופן דומה להגדרת פונקציות תבנית, ניתן להגדיר גם מחלקות תבנית.

דוגמה

```
#include <iostream>

using namespace std;

template<class T>
class Stack
{
private:
    T *m_arr;
    size_t m_maxSize;
    int m_current;

public:
    Stack(size_t maxSize)
    {
        m_arr = new T[m_maxSize = maxSize];
        m_current = 0;
    }

    Stack& Push(T value)
    {
        if(m_current < (int)m_maxSize)
        {
            *(m_arr + m_current++) = value;
        }
        return *this;
    }

    T Pop()
    {
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
        if(m_current > 0)
        {
            return *(m_arr + --m_current);
        }
        else
        {
            throw "Stack is empty";
        }
    }

    void Show()const
    {
        cout << '[' << m_current << " of " << m_maxSize << "] = { ";
        if(m_current > 0)
        {
            cout << *(m_arr + 0);
        }
        for(int i = 1; i < m_current; i++)
        {
            cout << ", " << *(m_arr + i);
        }
        cout << '}' << endl;
    }

    int GetSize() const { return m_current; }

    int GetMaxSize() const { return m_maxSize; }

    ~Stack()
    {
        delete[] m_arr;
    }
};

int main(int argc, char **argv)
{
    Stack<int> s1 = Stack<int>(10);
    s1.Push(3).Push(4).Push(-9).Push(10);
    s1.Show();
    s1.Pop();
    s1.Show();

    Stack<double> s2 = Stack<double>(12);
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    s2.Push(0.2).Push(0.3).Push(-9.5);
    s2.Show();
}
```

פלט התוכנית הוא:

```
[4 of 10] = { 3, 4, -9, 10}
[3 of 10] = { 3, 4, -9}
[3 of 12] = { 0.2, 0.3, -9.5}
```

בתוכנית זו יצרנו Stack גנרי, והשתמשנו בו ליצירת שני Stacks קונקרטיים - אחד מסוג int והשני מסוג double. לאחר מכן "דחפנו" מספר אלמנטים לכל Stack והדפסנו את תוכן כל Stack.

בקטע קוד זה, ה-Compiler יצר עבורנו קוד עבור שתי מחלקות Stack אחת עבור הסוג int והשנייה עבור הסוג double. מצב של שכפול קוד הוא נורמלי, אך יכול לגרום לניפוח התוכנית במיוחד במקרים בהם נעשה שימוש אינטנסיבי במחלקות תבנית.

על מנת לצמצם את כמות הקוד הנוצרת כאשר הקומפיילר יוצר העתקים של מחלקת התבנית, אפשר ליצור מחלקת בסיס שאינה תבנית, ולרכז בתוכה את כל הקוד שאינו תלוי בסוג הנתונים. ממחלקת בסיס זו אפשר אח"כ לרשת מחלקת תבנית, ובאופן כזה מצמצמים את שכפול הקוד של מחלקת הבסיס.

דוגמה

בדוגמה זו נעביר חלק מן הפונקציונליות של המחלקה Stack למחלקה אחרת, בשם StackBase. מחלקה זו תכיל את כל הקוד שאינו תלוי בסוג הנתונים. ממחלקה זו נגזור מחלקת תבנית בשם Stack. להלן קטע הקוד (Ex06):

```
#include <iostream>

using namespace std;

class StackBase
{
protected:
    size_t m_maxSize;
    int m_current;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    StackBase(size_t maxSize) { m_current = 0; m_maxSize = maxSize;
}
    virtual void PrintElementAt(int index)const = 0;
public:
    void Show()const
    {
        cout << '[' << m_current << " of " << m_maxSize << "] = { ";
        if(m_current > 0)
        {
            PrintElementAt(0);
        }
        for(int i = 1; i < m_current; i++)
        {
            cout << ", ";
            PrintElementAt(i);
        }
        cout << '}' << endl;
    }
    int GetSize() const { return m_current; }
    int GetMaxSize() const { return m_maxSize; }
    virtual ~StackBase(){ };
};


template<class T>
class Stack : public StackBase
{
private:
    T *m_arr;

public:
    Stack(size_t maxSize) : StackBase(maxSize)
    {
        m_arr = new T[maxSize];
    }

    Stack& Push(T value)
    {
        if(m_current < (int)m_maxSize)
        {
            *(m_arr + m_current++) = value;
        }
        return *this;
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    }

    T Pop()
    {
        if(m_current > 0)
        {
            return *(m_arr + --m_current);
        }
        else
        {
            throw "Stack is empty";
        }
    }

    void PrintElementAt(int index)const
    {
        cout << *(m_arr + index);
    }

    ~Stack()
    {
        delete[] m_arr;
    }
};

int main(int argc, char **argv)
{
    Stack<int> s1 = Stack<int>(10);
    s1.Push(3).Push(4).Push(-9).Push(10);
    s1.Show();
    s1.Pop();
    s1.Show();

    Stack<double> s2 = Stack<double>(12);
    s2.Push(0.2).Push(0.3).Push(-9.5);
    s2.Show();
}
```

פלט התוכנית הוא:

```
[4 of 10] = { 3, 4, -9, 10}
[3 of 10] = { 3, 4, -9}
[3 of 12] = { 0.2, 0.3, -9.5}
```

פלט זה זהה לפלט התוכנית הקודמת.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

נשים לב לדברים הבאים:

- העברנו למחלקת הבסיס את כל המשתנים והפונקציות שאפשר "לדחוס" לשם ושאינם תלויים בסוג האלמנט של ה-Stack.
- יצרנו פונקציה וירטואלית טהורה בשם PrintElementAt() המדפיסה אלמנט אחד במיקום נתון. על המחלקה היורשת לממש פונקציה זו - וזה אכן מה שנעשה ב-Stack. באופן זה יכולנו להעביר את כל המימוש של הפונקציה Show למחלקת הבסיס.
- שים לב שהקצאת הזיכרון ל-Stack נעשית במחלקה הנגזרת, ולכן חשוב להבטיח שה-Destructor של המחלקה הנגזרת ייקרא (כיוון ששם מתבצע שחרור הזיכרון). משיגים זאת על ידי הגדרת ה-Destructor של מחלקת הבסיס כווירטואלי.

5.4 העברת פרמטרים לתבנית

יחד עם סוגי הנתונים המועברים לתבנית, ניתן להעביר פרמטרים.

דוגמה

ה-Stack שהגדרנו בתוכניות הקודמות קיבל ב-Constructor את גודל ה-Stack המקסימלי. יכולנו להגדיר את גודל ה-Stack כאחד הפרמטרים של התבנית. באופן הבא (Ex07):

```
(same as the previous program)
...
...
template<class T, int MaxSize>
class Stack : public StackBase
{
private:
    T *m_arr;
public:
    Stack() : StackBase(MaxSize)
    {
        m_arr = new T[MaxSize];
    }

    Stack& Push(T value)
    {
        if(m_current < (int)m_maxSize)
        {
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
            *(m_arr + m_current++) = value;
        }
        return *this;
    }

    T Pop()
    {
        if(m_current > 0)
        {
            return *(m_arr + --m_current);
        }
        else
        {
            throw "Stack is empty";
        }
    }

    void PrintElementAt(int index)const
    {
        cout << *(m_arr + index);
    }

    ~Stack()
    {
        delete[] m_arr;
    }
};

int main(int argc, char **argv)
{
    Stack<int, 10> s1 = Stack<int, 10>();
    s1.Push(3).Push(4).Push(-9).Push(10);
    s1.Show();
    s1.Pop();
    s1.Show();
    Stack<double, 12> s2 = Stack<double, 12>();
    s2.Push(0.2).Push(0.3).Push(-9.5);
    s2.Show();
}
```

פלט התוכנית הוא:

```
[4 of 10] = { 3, 4, -9, 10}
[3 of 10] = { 3, 4, -9}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
[3 of 12] = { 0.2, 0.3, -9.5}
```

כפי שרואים הפלט לא השתנה. מה שהשתנה הוא שאת גודל ה-Stack אנו מעבירים כפרמטר לתבנית. במקרה זה הקומפיילר יוצר העתק קוד של התבנית עבור כל גודל שונה.

5.4.1 ברירת מחדל של פרמטרים של תבנית

ניתן לקבוע ערך ברירת מחדל לפרמטרים של תבנית כמו גם לסוגי הנתונים עצמם, באופן דומה לקביעת ערכי ברירת מחדל לפרמטרים של פונקציות.

דוגמה

```
template <class T = int, int size = 1024>
class Buffer
{
    ...
};
```

הערה: הגדרת ערכי ברירת מחדל לפרמטרים ולסוגי נתונים אפשרית רק במחלקות תבנית.

5.5 שימוש - אובייקט פונקציה

ניתן ליצור מחלקת תבנית ב-C++ שתתנהג כמו פונקציה על ידי העמסת האופרטור ().

דוגמה

ניצור פונקציה בשם Arithmetic המחשבת איבר בסדרה חשבונית על פי האינדקס. להלן הקוד (Ex08):

```
...
...
template<class T> class Arithmetic
{
private:
    T m_a1, m_d;
public:
    Arithmetic(T a1, T d) : m_a1(a1), m_d(d) { }
    T operator()(size_t n) { return m_a1 + ((n - 1) * m_d); }
};

...
...
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
int main(int argc, char **argv)
{
    Arithmetic<double> f(2.0, 3.0);
    cout << f(9) << endl;

    Arithmetic<complex> g(complex(1,3), complex(2,4));
    cout << g(3) << endl;
...
...
}
```

פלט התוכנית הוא:

```
26
5 + 11i
```

כפי שאפשר לראות הגדרנו מחלקה תבנית בשם Arithmetic המוגדרת על ידי סוג האיברים בסדרה. ה-Constructor של המחלקה מקבל את האיבר הראשון בסדרה ואת ההפרש. כמו כן, הגדרנו את האופרטור () כך שיקבל אינדקס ויחשב את האיבר במקום האינדקס שהועבר.

בפונקציה Main יצרנו שני אובייקטים של המחלקה והשתמשנו בהם. שים לב שאת האובייקט f יצרנו עם סוג נתונים double ואת g עם סוג הנתונים complex (ההגדרה המלאה של המחלקה complex נמצאת בתוכנית הדוגמה). לאחר מכן הפעלנו את האופרטור () של כל אחד מהאובייקטים כאילו האובייקט הוא פונקציה.

Template 5.6 ממספר סוגים

ניתן להעביר מספר סוגי נתונים בפרמטרי התבנית.

דוגמה

נניח כי ברצוננו ליצור מחלקה המפעילה פונקציה מסוימת על משתנה קלט ולאחסן את הקלט ואת התוצאה במערכים המיועדים לכך. באופן כזה קריאות חוזרות לפונקציה במחלקה יצברו תוצאות קלט ופלט. את סוג הקלט, סוג הפלט, ואת הפונקציה נרצה לקבל כפרמטרים של התבנית. להלן קוד התוכנית (Ex08 - התוכנית הקודמת במלואה):

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
#include <iostream>
#include <cstdlib>
#include "complex.h"

using namespace std;

template<class T> class Arithmetic
{
private:
    T m_a1, m_d;
public:
    Arithmetic(T a1, T d) : m_a1(a1), m_d(d) { }
    T operator()(size_t n) { return m_a1 + ((n - 1) * m_d); }
};

template<class T> class ComplexAbsolute
{
public:
    T operator()(const complex &c) {return c.absolute(); }
};

template<class I, class O, class F>
class Sequence
{
private:
    I *m_input;
    O *m_output;
    F *m_function;
    char *m_functionName;
    int m_current;
    size_t m_maxSize;

public:
    Sequence(const char *functionName, size_t maxSize)
    {
        m_current = 0;
        m_input = new I[maxSize];
        m_output = new O[maxSize];
        m_maxSize = maxSize;
        m_functionName = strdup(functionName);
    }

    void CalcFor(I input)
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    {
        if(m_current < (int) m_maxSize)
        {
            O output = F()(input);
            m_input[m_current] = input;
            m_output[m_current] = output;
            m_current++;
        }
    }

    template<class I2, class O2, class F2>
    friend ostream &operator<<(ostream &o, const Sequence<I2, O2,
F2> &seq);

    virtual ~Sequence()
    {
        delete[] m_input;
        delete[] m_output;
        free(m_functionName);
    }
};

template<class I, class O, class F>
ostream &operator<<(ostream &o, const Sequence<I, O, F> &seq)
{
    if(seq.m_current > 0)
    {
        o << seq.m_functionName << "(" << seq.m_input[0] << ") = "
            << seq.m_output[0];
    }
    for(int i = 1; i < seq.m_current; i++)
    {
        o << " ; " << seq.m_functionName << "(" << seq.m_input[i] <<
") = "
            << seq.m_output[i];
    }
    return o;
}

int main(int argc, char **argv)
{
    Arithmetic<double> f(2.0, 3.0);
    cout << f(9) << endl;
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    Arithmetic<complex> g(complex(1,3), complex(2,4));
    cout << g(3) << endl;

    Sequence<complex, double, ComplexAbsolute<double>> seq("abs",
10);
    seq.CalcFor(complex(3,4));
    seq.CalcFor(complex(7,24));
    seq.CalcFor(complex(8,15));
    cout << seq << endl;
}
```

פלט התוכנית הוא:

```
26
11 + 5i
abs(3 + 4i) = 5 ; abs(7 + 24i) = 25 ; abs(8 + 15i) = 17
```

כפי שאפשר לראות, הגדרנו מחלקה בשם Sequence המאחסנת סדרה של קלטים ופלטים שאותם היא צוברת בכל קריאה לפונקציה CalcFor. את סוג הקלט, הפלט ואת סוג הפונקציה היא מקבלת כפרמטרים של התבנית. ב-Constructor של המחלקה אפשר לתת שם לפונקציה ולהגדיר את המספר המקסימלי של תוצאות שיאוחסנו.

שים לב לאופן בו הגדרנו פונקציה ידידותית (friend) עבור האופרטור >>. השתמשנו במילה template ובהגדרת הסוגים, הן בהגדרת הפונקציה והן בהצהרת "הידידות".

הערה: מחלקות Stack ו-Sequence מנהלות זיכרון דינמי, אך לצורך פשטות הדוגמאות לא הוגדרו בהן Copy Constructor ואופרטור השמה. העתקה בטוחה פירושה יצירת עותקים עצמאיים של הנתונים, כך ששני אובייקטים לא יחזיקו במצביע לאותו בלוק זיכרון המנוהל על ידם. ללא מימוש מתאים, העתקת אובייקטים ממחלקות אלה עלולה לגרום לשחרור כפול של הזיכרון.

5.7 המילה השמורה typename

חסרון של התוכנית הקודמת הוא בכך שהיה עלינו להעביר שלושה סוגי נתונים לתבנית Sequence. ברור שהפונקציות Arithmetic ו-ComplexAbsolute "יודעות" עם אילו סוגי

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

נתונים הן פועלות, לכן די אם נעביר את סוג הפונקציה ל-Sequence ושם נזהה מהו סוג ה-Input של הפונקציה ומהו סוג ה-Output שלה.

להלן קוד התוכנית (Ex09), ולאחריו ההסבר על השימוש שנעשה במילה typename:

```
#include <iostream>
#include <cstdlib>
#include "complex.h"

using namespace std;

template<class T> class Arithmetic
{
private:
    T m_a1, m_d;
public:
    typedef T OutputType;
    typedef size_t InputType;
    Arithmetic(T a1, T d) : m_a1(a1), m_d(d) { }
    T operator()(size_t n) { return m_a1 + ((n - 1) * m_d); }
};

template<class T> class ComplexAbsolute
{
public:
    typedef T OutputType;
    typedef complex InputType;
    T operator()(const complex &c) {return c.absolute(); }
};

template<class F>
class Sequence
{
private:
    typename F::InputType *m_input;
    typename F::OutputType *m_output;
    F *m_function;
    char *m_functionName;
    int m_current;
    size_t m_maxSize;

public:
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    Sequence(const char *functionName, size_t maxSize)
    {
        m_current = 0;
        m_input = new typename F::InputType[maxSize];
        m_output = new typename F::OutputType[maxSize];
        m_maxSize = maxSize;
        m_functionName = strdup(functionName);
    }

    void CalcFor(typename F::InputType input)
    {
        if(m_current < (int) m_maxSize)
        {
            typename F::OutputType output = F()(input);
            m_input[m_current] = input;
            m_output[m_current] = output;
            m_current++;
        }
    }

    template<class F2>
    friend ostream &operator<<(ostream &o, const Sequence<F2> &seq);

    virtual ~Sequence()
    {
        delete[] m_input;
        delete[] m_output;
        free(m_functionName);
    }
};

template<class F>
ostream &operator<<(ostream &o, const Sequence<F> &seq)
{
    if(seq.m_current > 0)
    {
        o << seq.m_functionName << "(" << seq.m_input[0] << ") = " <<
seq.m_output[0];
    }
    for(int i = 1; i < seq.m_current; i++)
    {
        o << " ; " << seq.m_functionName << "(" << seq.m_input[i] <<
") = " << seq.m_output[i];
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    }
    return o;
}

int main(int argc, char **argv)
{
    Arithmetic<double> f(2.0, 3.0);
    cout << f(9) << endl;

    Arithmetic<complex> g(complex(1,3), complex(2,4));
    cout << g(3) << endl;

    Sequence<ComplexAbsolute<double>> seq("abs", 10);
    seq.CalcFor(complex(3,4));
    seq.CalcFor(complex(7,24));
    seq.CalcFor(complex(8,15));
    cout << seq << endl;
}
```

הסבר

- כל פונקציה מגדירה בתוכה באמצעות typedef את ה-InputType ואת ה-OutputType.
- Sequence משתמשת בסוגים הללו על מנת להצהיר על מערך ה-Input ועל מערך ה-Output.
- המילה typename משמשת כדי להגיד ל-Compiler כי מדובר בסוג נתונים ולא במשהו אחר. הסיבה נעוצה בכך שלעתים ישנן מחלקות המצהירות על משתנים סטטיים, ואז רישום כגון F::InputType *m_input; יכול להתפרש ככפל שבין שני מספרים. הוספת המילה typename מנחה את ה-Compiler כי מדובר בסוג נתונים.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שיעור 6:
RTTI & Casting

שים לב: קוד התוכניות של שיעור זה נמצא בספרייה Demos\Chapter 06.

בשיעור זה נכיר את האופן בו C++ מאפשר לזהות בזמן ריצה סוגי נתונים שאינם ידועים בזמן הקימפול. כמו כן נכיר אופרטורים ש-C++ מספק עבור המרה (Casting) של סוגי נתונים.

6.1 *Runtime Type Information - RTTI*

בניגוד לגישה הפולימורפית בה אנו מגדירים מחלקות פולימורפיות כך שפונקציות מופעלות על פי האובייקט הקונקרטי, באמצעות RTTI אנו יכולים "לשבור" התנהגות זו ולטפל בזמן ריצה באופן שונה באובייקטים מסוגים שונים.

6.1.1 המחלקה type_info

לכל סוג נתונים ב-C++ אפשר לקבל מופע מהמחלקה type_info הכולל בתוכו אינפורמציה אודות סוג נתונים. להלן ההגדרה של type_info (כמובן שישנו חלק שממומש באופן שונה בכל סביבה):

```
class type_info
{
public:
    virtual ~type_info();
    bool operator== (const type_info& rhs) const;
    bool operator!= (const type_info& rhs) const;
    bool before (const type_info& rhs) const;
    const char* name() const;
private:
    type_info (const type_info& rhs);
    type_info& operator= (const type_info& rhs);
};
```

כפי שניתן לראות, ה-Copy Constructor ואופרטור ההשמה נמצאים בחלק הפרטי. לא ניתן ליצור או להעתיק אובייקט מסוג type_info והדרך היחידה לקבל אובייקט כזה היא, כאמור, על ידי האופרטור typeid.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

בנוסף, כפי שרואים, אפשר להשוות שני מבנים מסוג type_info, כלומר ניתן להשתמש ב-typeid על מנת להשוות את סוגי הנתונים של שני משתנים בזמן ריצה. בנוסף ניתן לקבל את השם של סוג המשתנה באמצעות הפונקציה ()name.

הפונקציה ()before מאפשרת לקבל את יחס הסדר שבין שני עצמים מסוג type_info. בדרך כלל, יחס הסדר שיוחזר תלוי באופן המימוש של הקומפיילר, ולרוב זהו סדר לקסיקוגרפי. שים לב שיחס הסדר אינו חייב להיות קשור להיררכיית המחלקות.

6.1.2 האופרטור typeid

האופרטור שמאפשר לקבל אינפורמציה אודות סוג הנתונים של אובייקט נתון הוא typeid. אופרטור זה מחזיר Reference לאובייקט מסוג type_info.

דוגמה א

בדוגמה נפעיל את typeid על סוגי נתונים בסיסיים (Ex01):

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main()
{
    double d;
    double *pd;
    cout << "double is: " << typeid(double).name() << endl;
    cout << "d is: " << typeid(d).name() << endl;
    cout << "pd is: " << typeid(pd).name() << endl;
    cout << "*pd is: " << typeid(*pd).name() << endl << endl;
    cout << "typeid(d) " << (typeid(d) == typeid(*pd) ? "=" : "!")
        << "= typeid(*pd)" << endl;
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פלט התוכנית הוא:

```
double is: double
d is: double
pd is: double *
*pd is: double
typeid(d) == typeid(*pd)
```

כפי שרואים, אפשר להעביר לאופרטור typeid הן את סוג הנתונים עצמו והן את אובייקט מסוג הנתונים.

דוגמה ב

בדוגמה הבאה נראה כי קבלת מידע בזמן ריצה אינה עובדת באופן מושלם עבור אובייקטים שאינם פולימורפיים. להזכירך, אובייקט פולימורפי הוא אובייקט שמוגדרת בו לפחות פונקציה וירטואלית אחת. אם כן, במקרה של אובייקטים שאינם פולימורפיים זיהוי הסוג לא יזהה את האובייקט הקונקרטי אלא יזהה את סוג הנתונים כאילו הוא מסוג המצביע. להלן קוד התוכנית (Ex02):

```
#include <iostream>
#include <typeinfo>

using namespace std;

class Base {};
class Derived : public Base {};

int main()
{
    Derived derived;
    Base* pBase = &derived;
    cout << "Base is: " << typeid(Base).name() << endl;
    cout << "Derived is: " << typeid(Derived).name() << endl <<
endl;

    cout << "derived is: " << typeid(derived).name() << endl;
    cout << "*pbase is: " << typeid(*pBase).name() << endl;
    cout << "derived " << (typeid(derived) == typeid(*pBase) ? "=" :
"!")
        << "= *pBase" << endl;
}
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פלט התוכנית הוא:

```
Base is: class Base
Derived is: class Derived

derived is: class Derived
*pbase is: class Base
derived != *pBase
```

כפי שרואים, למרות שהאובייקט הקונקרטי המוצבע על ידי pBase הוא מסוג Derived, הפלט לא משקף את המצב בפועל.

דוגמה ג

בדוגמה זו נראה כי במקרה של אובייקטים פולימורפיים, סוג הנתונים המוחזר הוא סוג הנתונים שמוצבע בפועל, כלומר סוג הנתונים של האובייקט הקונקרטי. להלן קוד התוכנית (Ex03):

```
#include <iostream>
#include <typeinfo>

using namespace std;

class Base {virtual void Member(){}};
class Derived: public Base {};

int main()
{
    Derived derived;
    Base *pBase = &derived;
    cout << "derived is: " << typeid(derived).name() << endl;
    cout << "*pBase is: " << typeid(*pBase).name() << endl;
    cout << "derived " << (typeid(derived) == typeid(*pBase) ? "=" :
"!")
         << "= *pBase" << endl;
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פלט התוכנית הוא:

```
derived is: class Derived
*pBase is: class Derived
derived == *pBase
```

כפי שרואים, הביטוי typeid(*pBase) מחזיר את סוג הנתונים של האובייקט הקונקרטי, כלומר של class Derive.

Casting 6.2 - המרת סוגים ב-C++

משפת C אנו מכירים המרות של נתונים ומצביעים באופן הבא:

```
Class B {...};
Class D {...};
B b;
D *pD = (D*)&b;
```

המרה מסוג זה אינה מסבירה את כוונת המתכנת, מה גם שאינה הגיונית במקרה המתואר, שכן האובייקט b אינו יורש מ-D או בסיס למחלקה D.

במקרים מסוימים המרה מהסוג הישן רק משנה את אופן ההתייחסות למצביע, ובמקרים אחרים היא גם משנה את ערכו (כפי שניווכח בהמשך), בהתאם לסוגי הנתונים המעורבים בהמרה.

להמרת הסוגים בסגנון ההמרות הישן המוכר לנו משפת C יש חסרון נוסף, מתכנת שיקרא את מה שכתב מתכנת אחר, לא תמיד יצליח להבין מה הייתה כוונת המחבר. למשל, באמצעות המרה אפשר לבטל קביעות של אובייקט. למשל:

```
const B b;
...
...
B *pB = (B*)&b;
```

האם המתכנת שכתב שורות אלה התכוון באמת לבטל את הקביעות של האובייקט b או שמא ללא כוונת מכוון עשה זאת.

C++ מגדירה אופרטורי המרה שנועדו לענות בדיוק על הצרכים האלה.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

static_cast 6.2.1

המרה סטטית מתבצעת בזמן הקימפול של התוכנית. המרה זו לא בודקת בזמן ריצה את סוג הנתונים ומניחה שהמתכנת "יודע" מה הוא ממיר ולמה הוא ממיר. ההגדרות של המחלקות השונות חייבות להיות ידועות לפני שמבצעים המרה סטטית כיוון שעל ה-Compiler לדעת כיצד לבצע את ההמרה. תחביר ההמרה הוא:

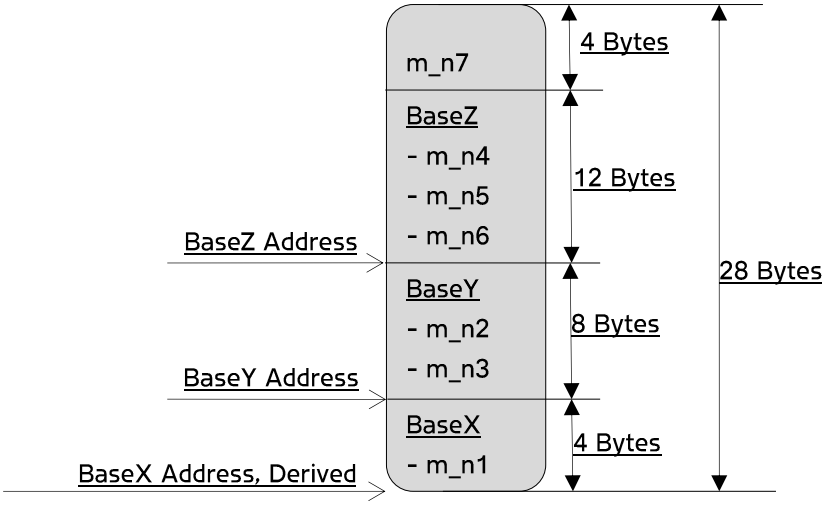
```
static_cast<requested_type>(value)
```

כאשר requested_type הוא הסוג הנדרש, ו-value הוא האובייקט שיש להמיר.

בדומה להמרה דינמית, כפי שנראה, המרה סטטית משנה כתובות במידת הצורך, אם כי בשונה מהמרה דינמית - המרה סטטית אפשרית גם עבור אובייקטים שאינם פולימורפים (עוד על המרה דינמית בהמשך).

דוגמה

בדוגמה הבאה, נגדיר שלוש מחלקות: BaseX, BaseY ו-BaseZ. בנוסף נגדיר מחלקה בשם Derived שתיגזר משלוש המחלקות הללו. פריסת הזיכרון עבור המחלקה Derived מתוארת בתרשים הבא:



© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

להלן קוד התוכנית (Ex04):

```
#include <iostream>
#include <cstdio>

class BaseX
{
private:
    int m_n1;
public:
    void PrintAddresses()
    {
        printf("BaseX(%d bytes): this = %.8X, &m_n1 = %.8X\n",
sizeof(BaseX),
        this, &m_n1);
    }
};

class BaseY
{
private:
    int m_n2;
    int m_n3;
public:
    void PrintAddresses()
    {
        printf("BaseY(%d bytes): this = %.8X, &m_n2 = %.8X,
               &m_n3 = %.8X\n",sizeof(BaseY), this, &m_n2, &m_n3);
    }
};

class BaseZ
{
private:
    int m_n4;
    int m_n5;
    int m_n6;
public:
    void PrintAddresses()
    {
        printf("BaseZ(%d bytes): this = %.8X, &m_n4 = %.8X, &m_n5 =
%.8X,
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
                &m_n6 = %.8X\n", sizeof(BaseZ), this, &m_n4, &m_n5,
&m_n6);
    }
};

class Derived : public BaseX, public BaseY, public BaseZ
{
private:
    int m_n7;
public:
    void PrintAddresses()
    {
        BaseX::PrintAddresses();
        BaseY::PrintAddresses();
        BaseZ::PrintAddresses();
        printf("Derived(%d bytes): this = %.8X, &m_n7 = %.8X\n",
                sizeof(Derived), this, &m_n7);
    }
};

int main(int argc, char **argv)
{
    Derived *pDerived = new Derived();
    pDerived->PrintAddresses();
    printf("\n");
    printf("pDerived = %.8X\n", pDerived);
    printf("static_cast<BaseX *>(pDerived) = %.8X\n",
           static_cast<BaseX *>(pDerived));
    printf("static_cast<BaseY *>(pDerived) = %.8X\n",
           static_cast<BaseY *>(pDerived));
    printf("static_cast<BaseZ *>(pDerived) = %.8X\n",
           static_cast<BaseZ *>(pDerived));
    printf("\n");
    static_cast<BaseX&>(*pDerived).PrintAddresses();
    delete pDerived;
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

פלט התוכנית הוא:

```
BaseX(4 bytes): this = 00164BA8, &m_n1 = 00164BA8
BaseY(8 bytes): this = 00164BAC, &m_n2 = 00164BAC, &m_n3 = 00164BB0
BaseZ(12 bytes): this = 00164BB4, &m_n4 = 00164BB4, &m_n5 =
00164BB8, &m_n6 = 00164BBC
Derived(28 bytes): this = 00164BA8, &m_n7 = 00164BC0

pDerived = 00164BA8
static_cast<BaseX *>(pDerived) = 00164BA8
static_cast<BaseY *>(pDerived) = 00164BAC
static_cast<BaseZ *>(pDerived) = 00164BB4

BaseX(4 bytes): this = 00144BA8, &m_n1 = 00144BA8
```

כפי שאפשר להבחין, המרה סטטית מחזירה לעתים כתובת מצביע שונה על פי ה-offset המתאים. כמו כן רואים שאפשר לבצע המרות גם לייחוסים.

בנוסף, אם היינו מנסים להמיר מצביע מסוג BaseX למצביע מסוג BaseY היינו מקבלים שגיאת קומפילציה, כיוון שה-Compiler מבחין בכך שהדבר אינו אפשרי. אפשר לעקוף מגבלה על ידי המרת המצביע למצביע מסוג void * או מצביע מסוג Derived ורק לאחר מכן המרתו למצביע מסוג *BaseY. מיותר לציין שאין הגיון לעשות זאת.

6.2.2 dynamic_cast

המרה דינמית מאפשרת להמיר מצביע (או ייחוס) מסוג אחד לסוג שני, כאשר מתבצעת בדיקה בזמן ריצה שהאובייקט הקונקרטי אכן נגזר מסוג המצביע (או הייחוס) הנדרש בהמרה. המרה דינמית אפשרית רק עבור אובייקטים פולימורפיים, כלומר אובייקטים שהוגדרה עבורם לפחות פונקציה וירטואלית אחת.

תחביר ההמרה הוא:

```
dynamic_cast<requested_type>(value)
```

כאשר requested_type הוא הסוג המבוקש (מצביע או ייחוס), ו-value הוא האובייקט שיש להמיר.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

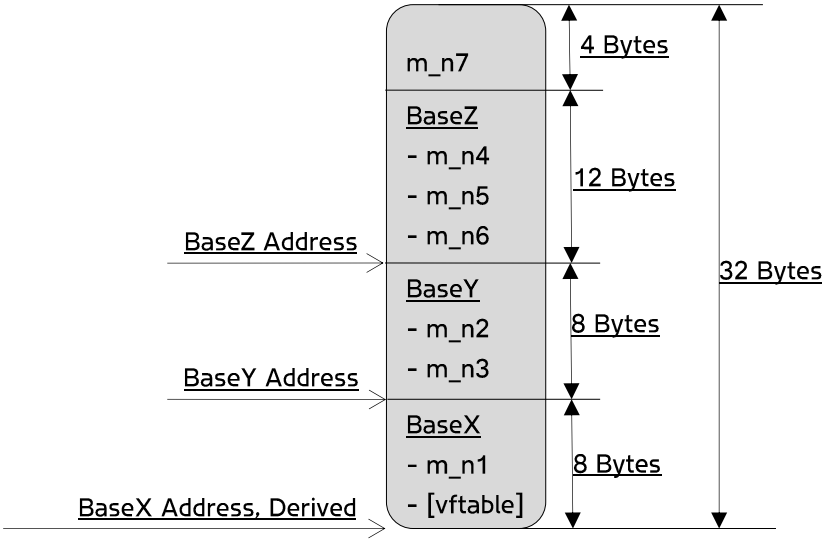
dynamic_cast מחזיר מצביע מהסוג המבוקש, או 0 (NULL) אם לא ניתן להמיר לסוג המבוקש. לעומת זאת, אם המרה לייחוס לא מצליחה ייזרק exception מסוג bad_cast (זה קורה כיוון שאין אפשרות אחרת לציין שההמרה לא הצליחה).

בניגוד ל-static_cast ב-dynamic_cast אפשר לנסות ולהמיר מצביע מסוג אחד לסוג אחר למרות שאין ביניהם קשר היררכי. במקרה זה יוחזר 0 (NULL) או שייזרק exception. במילים אחרות, dynamic_cast מבצע בדיקות מסוימות כבר בזמן הקומפילציה, אך בנוסף מאפשר לבדוק בזמן ריצה אם האובייקט הקונקרטי תומך בהמרה המבוקשת. זו אחת הסיבות לשימושיות שלו.

דוגמה

בדוגמה זו שינינו את התוכנית הקודמת בכך שהוספנו פונקציה וירטואלית למחלקה BaseX. שינוי זה הוסיף 4 בתים לזיכרון של BaseX וכתוצאה מכך גם לגודל של Derived (אלו הם ארבעת הבתים המשמשים מצביע ל-Virtual Function Table), והפך את המחלקות BaseX ו-Derived למחלקות פולימורפיות.

להלן פריסת הזיכרון של היררכיית המחלקות החדשה:



© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

להלן קוד הדוגמה (Ex05):

```
#include <iostream>
#include <cstdio>
#include <typeinfo>

using namespace std;

class BaseX
{
private:
    int m_n1;
public:
    void PrintAddresses()
    {
        printf("BaseX(%d bytes): this = %.8X, &m_n1 = %.8X\n",
sizeof(BaseX),
               this, &m_n1);
    }
    virtual void dummy(){}
};

class BaseY
{
private:
    int m_n2;
    int m_n3;
public:
    void PrintAddresses()
    {
        printf("BaseY(%d bytes): this = %.8X, &m_n2 = %.8X, &m_n3 =
%.8X\n",
            sizeof(BaseY), this, &m_n2, &m_n3);
    }
};

class BaseZ
{
private:
    int m_n4;
    int m_n5;
    int m_n6;
public:
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    void PrintAddresses()
    {
        printf("BaseZ(%d bytes): this = %.8X, &m_n4 = %.8X, &m_n5 =
%.8X,
            &m_n6 = %.8X\n", sizeof(BaseZ), this, &m_n4, &m_n5,
&m_n6);
    }
};

class Derived : public BaseX, public BaseY, public BaseZ
{
private:
    int m_n7;
public:
    void PrintAddresses()
    {
        BaseX::PrintAddresses();
        BaseY::PrintAddresses();
        BaseZ::PrintAddresses();
        printf("Derived(%d bytes): this = %.8X, &m_n7 = %.8X \n",
               sizeof(Derived), this, &m_n7);
    }
};

int main(int argc, char **argv)
{
    Derived *pDerived = new Derived();
    pDerived->PrintAddresses();
    printf("\n");
    printf("pDerived = %.8X\n", pDerived);
    printf("dynamic_cast<BaseX *>(pDerived) = %.8X\n",
           dynamic_cast<BaseX *>(pDerived));
    printf("dynamic_cast<BaseY *>(pDerived) = %.8X\n",
           dynamic_cast<BaseY *>(pDerived));
    printf("dynamic_cast<BaseZ *>(pDerived) = %.8X\n",
           dynamic_cast<BaseZ *>(pDerived));
    delete pDerived;

    printf("\n");
    BaseX x;
    printf("dynamic_cast<BaseY*>(&x) = %.8X\n",
dynamic_cast<BaseY*>(&x));
    try
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    {
        dynamic_cast<BaseY&>(x);
        printf("Reference conversion succeeded!\n");
    }
    catch(bad_cast &bc)
    {
        printf("Reference conversion failed ('%s')!\n", bc.what());
    }

}
```

פלט התוכנית הוא:

```
BaseX(8 bytes): this = 000F4BA8, &m_n1 = 000F4BAC
BaseY(8 bytes): this = 000F4BB0, &m_n2 = 000F4BB0, &m_n3 = 000F4BB4
BaseZ(12 bytes): this = 000F4BB8, &m_n4 = 000F4BB8, &m_n5 =
000F4BBC, &m_n6 = 00
0F4BC0
Derived(32 bytes): this = 000F4BA8, &m_n7 = 000F4BC4

pDerived = 000F4BA8
dynamic_cast<BaseX *>(pDerived) = 000F4BA8
dynamic _cast<BaseY *>(pDerived) = 000F4BB0
dynamic _cast<BaseZ *>(pDerived) = 000F4BB8

dynamic_cast<BaseY*>(&x) = 00000000
Reference conversion failed ('Bad dynamic_cast!')!
```

בדוגמה זו אפשר לראות שהמרה דינמית משנה, במידת הצורך, את כתובת המצביע, בדיוק כמו בהמרה סטטית. כמו כן רואים שהמרת מצביע שלא מצליחה מחזירה 0 (NULL), ואילו המרת ייחוס שלא מצליחה גורמת לזריקת bad_cast exception - בהמרה סטטית לא יכולנו לנסות לבצע המרה בין שני סוגי מצביעים שאין ביניהם קשר היררכי, בהמרה דינמית זה מתאפשר - אם כי התוצאות בהתאם (NULL או bad_cast exception).

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

const_cast 6.2.3

המרה זו מאפשרת לבטל "קביעות" של אובייקט. תחביר האופרטור הוא:

```
const_cast<requested_type>(value)
```

דוגמה

להלן דוגמה הממחישה את אופן השימוש באופרטור const_cast (Ex06):

```
#include <iostream>

using namespace std;

class Object
{
private:
    int m_value;

public:
    Object(int value) { m_value = value; }
    void Display() const { cout << m_value << endl << endl; }
    void Change() { m_value++; }
    void ForceChange() const { const_cast<Object*>(this)->m_value++;
}
};

void OuterChangeFunction(const Object &object)
{
    // the following line causes a compilation error
    // object.Change();
    object.ForceChange();
    object.Display();
}

void OuterForceChangeFunction(const Object &object)
{
    Object &tmp = const_cast<Object &>(object);
    // Now both lines are OK!
    tmp.Change();
    tmp.ForceChange();
    tmp.Display();
}
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
void OuterForceChangeFunctionV2(const Object &object)
{
    Object *pTmp = const_cast<Object *>(&object);
    // Now both lines are OK!
    pTmp->Change();
    pTmp->ForceChange();
    pTmp->Display();
}

int main(int argc, char **argv)
{
    Object object(10);
    OuterChangeFunction(object);
    OuterForceChangeFunction(object);
    OuterForceChangeFunctionV2(object);
}
```

פלט התוכנית הוא:

```
11
13
15
```

כפי שניתן לראות, פונקציות חברות במחלקה המוגדרות כפונקציות const אינן יכולות לשנות ערך של חבר במחלקה, ולכן מתבצעת המרת const_cast למצביע this על מנת שאפשר יהיה לשנות את ערך המשתנה m_value.

באותו אופן, פונקציות חיצוניות למחלקה המקבלות את האובייקט כפרמטר מסוג const, אינן יכולות להפעיל פונקציות שאינן מוגדרות כפונקציות const במחלקה. אם ברצוננו לעשות זאת, עלינו להמיר את האובייקט שהועבר באמצעות const_cast ורק אז אפשר יהיה להפעיל פונקציה שאינה מוגדרת כפונקציית const.

הערה: const_cast מאפשר להסיר את מגבלת ה-const ממצביע או מייחוס, אך אינו מבטל את הקביעות של האובייקט עצמו. אם האובייקט הוגדר מלכתחילה כ-const, ניסיון לשנות את ערכו באמצעות ההמרה יגרום להתנהגות בלתי מוגדרת (Undefined Behavior). הסיבה לכך היא שה-Compiler רשאי להניח שערכו של אובייקט קבוע אינו משתנה, ולבצע אופטימיזציות בהתאם. למשל, הוא עשוי להשתמש בערך המקורי של המשתנה גם לאחר שנעשה ניסיון לשנותו, או למקם אותו באזור זיכרון המוגן מפני

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

כתיבה. לכן, השינוי עשוי להיראות כאילו הצליח, להיכשל או לגרום לתוצאות בלתי צפויות. לעומת זאת, כאשר האובייקט המקורי אינו const, ניתן לשנות אותו בדרך זו באופן תקין.

לרוב שימוש ב-const_cast מעיד על עיצוב תוכנה לא טוב, שכן בעיצוב תוכנה טוב אין צורך לבטל את המאפיין const של אובייקט בזמן ריצה. עם זאת, ייתכן מצב בו יתעורר צורך כזה.

דוגמה

נניח שלשימושים פנימיים רוצים לספור את כמות הפעמים שקראו לפונקציית הדפסה של אובייקט. ברור כי פונקציית ההדפסה אינה משנה את ערכו של האובייקט ולכן נרצה להגדירה כפונקציית const, עם זאת כן נרצה להגדיל את המונה של מספר הקריאות לפונקציית ההדפסה. כדי לעשות זאת, בפונקציית ההדפסה נמיר את המצביע this באמצעות האופרטור const_cast. להלן קוד התוכנית (Ex07):

```
#include <iostream>

using namespace std;

class Object
{
private:
    int m_value;
    int m_displayCount;

public:
    Object(int value) { m_value = value; m_displayCount = 0; }
    void Display() const
    {
        cout << m_value << endl;
        const_cast<Object*>(this)->m_displayCount++;
    }
    int GetDisplayCount() { return m_displayCount; }
};

int main(int argc, char **argv)
{
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    Object object(10);
    object.Display();
    object.Display();
    object.Display();
    cout << "display count = " << object.GetDisplayCount() << endl;
}
```

פלט התוכנית הוא:

```
10
10
10
display count = 3
```

כפי שאפשר לראות, למרות שהפונקציה Display מוגדרת כפונקציית const (כלומר כפונקציה שלא יכולה לשנות ערכי משתנים חברים), היא בכל זאת עושה זאת על ידי המרה מתאימה של המצביע this.

הערה: יש לציין כי אפשר להשיג את אותו אפקט, וללא המרת const_cast על ידי הגדרת המשתנה m_displayCount כמשתנה מסוג mutable, כפי שהוסבר בשיעור 6.

6.2.4 reinterpret_cast

המרה זו היא המקבילה להמרה הישנה המוכרת לנו משפת C. המרה זו מאפשרת לשנות כל סוג של מצביע לכל סוג של מצביע, כמו גם כל סוג של ייחוס לכל סוג של ייחוס. לא מתבצעת בדיקה כלשהי לגבי ההמרה (לא בזמן ריצה ולא בזמן קימפול), וכתובות ההצבעה אינן משתנות. מובן שהמרות כאלה מסוכנות, אך זהו המפלט האחרון להמרות שאין לנו שום דרך לבצע אותן.

תחביר הפקודה הוא:

```
reinterpret_cast<requested_type>(value)
```

כאשר requested_type הוא סוג הנתונים המבוקש, ו-value הוא האובייקט להמרה.

דוגמה

הדוגמה הבאה מציגה שימוש בהמרה מסוג reinterpret_cast (Ex08):

```
#include <iostream>
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
using namespace std;

class A { };
class B { };

int main(int argc, char **argv)
{
    int n = 123456;
    float* pF = reinterpret_cast<float*>(&n); /* No error */

    A a;
    B *pB = reinterpret_cast<B*>(&a); /* No error */
    B &anotherB = reinterpret_cast<B&>(a); /* No error */
}
```

כפי שאפשר לראות אף שגיאה לא נזרקת בהמרות האלה, לא בזמן הקימפול ולא בזמן הריצה. כמובן שאם ננסה לבצע, בזמן ריצה, פעולות הייחודיות ל-B על האובייקט שהוא למעשה אובייקט מסוג A, סביר מאוד שהתוכנית תתנהג באופן לא צפוי.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שיעור 7:
STL - The Standard Library

שים לב: קוד התוכניות של שיעור זה נמצא בספרייה Demos\Chapter 07.

STL הוא אוסף מחלקות תבנית. ספרייה זו ברובה היא פרי רעיונותיו של אדם אחד - אלכסנדר סטפנוב. הוא התחיל לפתח את רעיונות התכנות הגנרי שלו עוד בשנת 1979 ולאחר שנים יישם את רעיונותיו בשפות Ada ו-C++. בהמשך STL נכלל באופן רשמי כחלק מתקן שפת C++.

אפשר לחלק את STL לארבעה מרכיבים עיקריים:

- **Containers** (מכולות) - אלו מחלקות תבנית המממשות מבני נתונים שונים, למשל: vector - מערך דינמי, queue - תור, list – רשימה, map וכולי.
- **Iterators** (סורקים) - אלו הם עצמים המאפשרים ביצוע סריקות על מכולות.
- **Algorithms** - כמות נכבדת של אלגוריתמים המבצעים סריקה, חיפוש, מיון וכולי.
- **Functors** (אובייקטי פונקציה) - אלו הן מחלקות המגדירות את האופרטור ().

בשיעור זה נסקור חלק מהמחלקות והפונקציות הגנריות הללו. לאחר שצורת העבודה תהיה מובנת לך, תוכל בקלות רבה ללמוד את היתר.

7.1 מבוא לאיטרטורים

איטרטור הוא אובייקט שתפקידו לאפשר סריקה של מערך. האובייקט, בהתנהגותו, מזכיר את המצביע המוכר לנו, אם כי הוא אובייקט של ממש ובדרך כלל בכל תבנית מכולה מוגדרים מחלקות איטרטורים מסוגים שונים המאפשרים סריקות על איברי המכולה. את האיטרטורים אפשר לחלק לסוגים הבאים:

- **random access** - איטרטורים כאלה מאפשרים גישה אקראית לכל האיברים המאוחסנים במכולה.
- **bi-directional** - איטרטורים המאפשרים סריקה דו כיוונית.
- **forward** - איטרטורים המאפשרים סריקה קדימה בלבד.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- **input** - איטרטורים הנעים קדימה מאפשרים קריאה בלבד של עצמים מהמכולה.
- **output** - איטרטורים הנעים קדימה מאפשרים כתיבה בלבד של עצמים למכולה.

כפי שנראה השימוש באיטרטורים הוא אינטואיטיבי הרבה יותר מהמצופה. האופרטורים ++ ו -- להם אנו רגילים ממצביעים רגילים, עובדים בדיוק באותו אופן באיטרטורים.

7.2 האוסף vector - מערך דינמי

המחלקה vector מאפשר ליצור מערך בעל גודל משתנה, לבצע עליו סריקות שונות וכולי.

שים לב: קוד התוכנית המלא המדגים את השימושים האפשריים של וקטור נמצא בתוכנית Ex01.

7.2.1 הגדרת הווקטור

הגדרת הווקטור היא:

```
template<class T, class A = allocator<T>>
class vector
{
...
};
```

הווקטור מקבל שני סוגי נתונים:

- T - סוג הנתונים במערך הדינמי
- A - מקצה הזיכרון של פריטים במערך: ניתן להגדיר מהי המחלקה שתקצה את הזיכרון לאיברים במערך. בדרך כלל נסתפק במקצה ברירת המחדל.

7.2.2 אתחול וקטור

לווקטור מספר Constructors המאפשרים אתחול באופנים שונים.

דוגמאות

```
vector<int> v1;
```

בדוגמה זו יצרנו וקטור של מספרים שלמים ריק.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
vector<int> v2(10, 5);
```

בדוגמה זו יצרנו וקטור מספרים שלמים שאותחל עם 10 איברים שערך כל אחד מהם הוא 5.

```
vector<int> v3(v2.begin(), v2.end());
```

בדוגמה זו יצרנו מערך דינמי (v3) שאותחל על ידי מערך אחר (v2). כדי לעשות זאת, העברנו ל-v3 שני איטרטורים, הראשון מצביע על האיבר הראשון שיש להעתיק מ-v2 והשני על האחרון (לא כולל). השתמשנו בפונקציה begin() המחזירה איטרטור המצביע על תחילת המערך, והפונקציה end() מחזירה איטרטור המצביע על איבר אחד אחרי האחרון שבמערך.

```
vector<int> v4(v2.begin() + 2, v2.end() - 3);
```

בדוגמה זו אתחלנו מערך באופן דומה למערך v3, אלא שכאן אנו מעתיקים מ-v2 החל מהאיבר השלישי ועד לאיבר השלישי מהסוף (לא כולל).

7.2.3 גודל הווקטור וניצול הזיכרון

הפונקציה size() מחזירה את גודל הווקטור.

הפונקציה capacity() מחזירה את ניצולת הזיכרון של המערך (בין אם אלה איברים שבשימוש ובין אם לאו).

דוגמה

```
cout << "v4.size()= " << v4.size() << endl;
cout << "v4.capacity()= " << v4.capacity() << endl;
```

7.2.4 מחיקת כל איברי המערך

הפונקציה clear() מאפשרת למחוק את כל איברי המערך.

דוגמה

```
v2.clear();
v3.clear();
```

כאן רוקנו שני המערכים v2-וv3.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

7.2.5 הוספת איברים

- הפונקציה push_back מאפשרת להוסיף איבר לסוף המערך.
- הפונקציה insert מאפשרת להוסיף איבר או איברים במיקום מסוים במערך

דוגמאות

```
srand(time(NULL));
for(int i = 0; i < 10; i++)
{
    v1.push_back(rand() % 100);
    v2.insert(v2.begin(), rand() % 100 + 100);
}
```

כאן הוספנו עשרה ערכים אקראיים למערך v1, ולמערך v2. ל-v1 הוספנו בכל איטרציה איבר אחד לסוף המערך. ל-v2 הוספנו איבר אחד בכל פעם, לתחילת המערך.

```
v3.insert(v3.begin(), v1.rbegin(), v1.rend());
```

בשורה זו הוספנו את איברי המערך v1 בסדר הפוך למערך v3 החל מתחילת v3.

7.2.6 מחיקת איברים במערך

הפונקציה pop_back() מוחקת את האיבר האחרון במערך.

דוגמה

```
v1.pop_back();
```

הפונקציה erase() מאפשרת למחוק איבר אחד על ידי העברת איטרטור המצביע על האיבר, או קבוצה של איברים על ידי העברת איטרטור לתחילת הקבוצה ואיטרטור לסוף הקבוצה (לא כולל). הפונקציה תחזיר איטרטור המצביע על האיבר הראשון שאחרי האחרון שהוסר.

דוגמה

```
v2.erase(v2.begin() + 3, v2.begin() + 5);
```

בדוגמה זו מוחקים את האיברים הרביעי והחמישי של המערך.

7.2.7 גודל מקסימלי של מערך

הפונקציה max_size() מחזירה את מספר האיברים המקסימלי האפשרי במערך.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

7.2.8 איטרטורים לתחילת המערך וסוף המערך

הפונקציות הבאות, בהן כבר נתקלת בדוגמאות הקודמות, מחזירות איטרטורים:

- **begin()** - מחזיר איטרטור לתחילת המערך
- **end()** - מחזיר איטרטור לסוף המערך.
- **rbegin()** - מחזיר איטרטור הפוך לתחילת המערך (כלומר זהו איטרטור שלמעשה מצביע על סוף המערך, וקידומו ב-1 הוא בעצם התקדמות לכיוון תחילת המערך - כלומר חזרה אחורה)
- **rend()** - מחזיר איטרטור הפוך לסוף המערך (למעשה איטרטור זה מצביע על תחילת המערך, והפחתה ב-1 של ערכו מקדמת אותו מקום אחד לכיוון סוף המערך).

נשים לב כי האיטרטורים של המחלקה vector הם איטרטורים לגישה אקראית.

דוגמה

```
static void print(vector<T,A> &v)
{
    /* example of forward iteration */
    vector<T,A>::iterator iter = v.begin();
    while(iter != v.end())
    {
        cout << *iter++ << " ";
    }
    cout << endl;
}

static void reverse_print(vector<T,A> &v)
{
    /* example of backward iteration */
    vector<T,A>::reverse_iterator riter = v.rbegin();
    while(riter != v.rend())
    {
        cout << *riter++ << " ";
    }
    cout << endl;
}
```

בדוגמה זו הגדרנו שתי פונקציות להדפסת איברי מערך, אחת הדפסה רציפה קדימה והשנייה הדפסה בסדר הפוך.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

שים לב: כדי לקבל את תוכן האיבר הנוכחי משתמשים באופרטור *, בדיוק כפי שהיינו עושים אם היה מדובר במצביעים. ההתנהגות המוכרת לנו ממצביעים מקילה עלינו להבין את אופן השימוש באיטרטורים של STL.

7.2.9 אחזור איבר

כדי לקבל איבר אפשר להשתמש בפונקציה ()at או באופרטור []. ניתן לשנות את ערך האיבר עם קבלתו.

```
v1.at(0) = 9;
v1[6] = 11;
```

7.2.10 התייחסויות לאיבר ראשון ואיבר אחרון

הפונקציות ()front ו-()back מחזירות התייחסות לאיבר הראשון והאחרון במערך, בהתאמה. בשונה מהאיטרטורים המוזכרים קודם, כאן מקבלים התייחסות לאיבר עצמו ואין צורך להשתמש באופרטור * על מנת לקבל את תוכנו. בנוסף, אי אפשר להשתמש בהתייחסויות כאיטרטורים - כלומר לקדמם או להסיגם.

דוגמה

```
cout << "v2.front() = " << v2.front() << endl;
cout << "v2.back() = " << v2.back() << endl;
```

7.3 אחסון אובייקטים ממחלקה

בסעיף הקודם הדגמנו את השימוש ב-vector עבור אחסון דינמי של מספרים שלמים. כמובן שאפשר גם לאחסן איברים ממחלקה כלשהי. עם זאת חשוב לציין שעל מנת לנצל את האפשרויות הרבות ש-STL מספק (כפי שנראה בהמשך), יש צורך לעתים להגדיר אופרטורים מסוימים, פונקציות מסוימות, constructors מסוימים וכו'. למשל, על מנת לנצל את אלגוריתמי המיון יהיה עלינו להגדיר יחס גודל במחלקה שלנו באמצעות הגדרת האופרטור > (קטן מ). הדרישות עשויות להשתנות מאלגוריתם לאלגוריתם וממכולה אחת לאחרת.

חשוב להדגיש שבמכולות STL מאוחסן עותק של האובייקט שמועבר לאחת מפונקציות האחסון של כל מכולה. לפיכך, במיוחד במקרים בהם מנוצלים משאבים על ידי האובייקט

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

(כגון מחלקת String שבדוגמה הבאה), יהיה עלינו להגדיר היטב את ה-Copy Constructor של המחלקה ואופרטור ההשמה.

7.3.1 האלגוריתם sort

במהלך הדוגמאות הבאות נציג מספר אלגוריתמים. כל האלגוריתמים דורשים הכללה של הקובץ <algorithm>.

האלגוריתם הראשון שנכיר הוא sort. אלגוריתם זה משתמש באופרטור > (קטן מ) על מנת לבצע את המיון, או בגרסתו השנייה באובייקט Compare המועבר לפונקציה. מיון זה אינו מתחייב להיות יציב (המיון stable_sort מבטיח להיות יציב).

הערה: מיון יציב הוא מיון המבטיח ששני איברים בעלי ערך זהה לפי קריטריון המיון ישמרו על הסדר המקורי ביניהם.

דוגמה

להלן קוד תוכנית לדוגמה (Ex02):

```
#include <iostream>
#include <cstring>
#include <vector>
#include <algorithm>
#include <cstdlib>

using namespace std;

template<typename T, typename A = allocator<T>>
class vector_utils
{
public:
    static void print(vector<T,A> &v)
    {
        /* example of forward iteration */
        vector<T,A>::iterator iter = v.begin();
        while(iter != v.end())
        {
            cout << *iter++ << " ";
        }
        cout << endl;
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    }
};

class String
{
private:
    char *m_str;
public:
    String(const char *str) { m_str = strdup(str); }
    String(const String &str) { m_str = strdup(str.m_str); }
    const String &operator=(const String &str)
    {
        if (this == &str)
            return *this;

        free(m_str);
        m_str = strdup(str.m_str);
        return *this;
    }
    bool operator<(const String &str) const
    {
        return strlen(m_str) < strlen(str.m_str);
    }
    ~String() { free(m_str); }
    friend ostream& operator<<(ostream &o, const String str);
};
ostream& operator<<(ostream &o, const String str)
{
    o << str.m_str;
    return o;
}

int main(int argc, char **argv)
{
    vector<String> v;
    v.push_back("ilan");
    v.push_back("amoyal");
    v.push_back("is");
    v.push_back("the");
    v.push_back("writer");
    v.push_back("of");
    v.push_back("this");
    v.push_back("book");
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    sort(v.begin(), v.end());
    vector_utils<String>::print(v);
}
```

פלט התוכנית הוא:

```
is of the ilan this book amoyal writer
```

בדוגמה זו הגדרנו מחלקה בשם String שיחס הגודל בה בין שני Strings נקבע על פי אורך המחרוזת (ולא בסדר לקסיקוגרפי). לאחר מכן השתמשנו באלגוריתם מיון ש-STL מספק על מנת למיין את איברי המערך שיצרנו (בהמשך נכיר אלגוריתמים נוספים). היה עלינו להגדיר את האופרטור > (קטן מ), על מנת שהמיון יוכל לפעול על המכולה. לפונקציית המיון העברנו את האיטרטור לתחילת התחום שאנו רוצים למיין ואת האיטרטור לסוף התחום. שים לב גם להגדרה של ה-Copy Constructor, אופרטור ההשמה ושחרור הזיכרון ב-Destructor. ללא פונקציות אלה תפקוד התוכנית יהיה לא צפוי או שלחילופין יהיו זליגות זיכרון (Memory Leaks).

7.4 המחלקות PhoneEntry ו-PhoneEntryPredicate

בתוכניות הבאות נעשה שימוש במחלקה PhoneEntry, ולכן נציג את קוד המחלקה לפני תוכניות הדוגמה שבסעיפים הבאים. בנוסף, יצרנו מחלקה בשם PhoneEntryPredicate המגדירה את האופרטור () כך שאופרטור זה מקבל מופע של PhoneEntry ובודק האם הוא שווה ל-value שהועבר ב-Constructor. הבחירה אם ההשוואה תהיה תלויה ברישיות (case sensitive) והאם יש להשוות לערך השם או ערך הטלפון - נעשית בפרמטרים המועברים ב-Constructor.

להלן הצהרת המחלקות:

```
File: PhoneEntry.h

#ifndef PHONE_ENTRY_H
#define PHONE_ENTRY_H

#include <iostream>
#include <cstdlib>

class PhoneEntry
{
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
private:
    char *m_name, *m_phone;
public:
    PhoneEntry();
    PhoneEntry(const char *name, const char *phone);
    PhoneEntry(const PhoneEntry &phoneEntry);
    ~PhoneEntry();

    const PhoneEntry &operator=(const PhoneEntry &phoneEntry);
    bool operator==(const PhoneEntry &phoneEntry)const;
    bool operator<(const PhoneEntry &phoneEntry)const;
    bool operator>(const PhoneEntry &phoneEntry)const;
    bool operator<=(const PhoneEntry &phoneEntry)const;
    bool operator>=(const PhoneEntry &phoneEntry)const;

    friend std::ostream &operator << (std::ostream &o, const
PhoneEntry &p);

    const char *get_name()const;
    const char *get_phone()const;
};

class PhoneEntryPredicate
{
private:
    char *m_value;
    bool m_ignoreCase;
    bool m_byPhone;

public:
    PhoneEntryPredicate();
    PhoneEntryPredicate(const char*value, bool ignoreCase = false,
bool byPhone = false);
    PhoneEntryPredicate(const PhoneEntryPredicate &pred);
    bool operator()(const PhoneEntry &p);
    ~PhoneEntryPredicate();
};

#endif // PHONE_ENTRY_H
```

להלן מימושי המחלקות:

```
File: PhoneEntry.cpp
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
#include "PhoneEntry.h"
#include <cstdlib>


PhoneEntry::PhoneEntry() { m_name = m_phone = NULL;}

PhoneEntry::PhoneEntry(const char *name, const char *phone)
{
    m_name = strdup(name);
    m_phone = strdup(phone);
}

PhoneEntry::PhoneEntry(const PhoneEntry &phoneEntry)
{
    m_name = strdup(phoneEntry.m_name);
    m_phone = strdup(phoneEntry.m_phone);
}

PhoneEntry::~PhoneEntry()
{
    free(m_name);
    free(m_phone);
}

const PhoneEntry &PhoneEntry::operator=(const PhoneEntry
&phoneEntry)
{
    if (this == &phoneEntry)
        return *this;

    free(m_name);
    free(m_phone);
    m_name = strdup(phoneEntry.m_name);
    m_phone = strdup(phoneEntry.m_phone);
    return *this;
}

bool PhoneEntry::operator==(const PhoneEntry &phoneEntry)const
{
    return stricmp(m_name, phoneEntry.m_name) == 0;
}

bool PhoneEntry::operator<(const PhoneEntry &phoneEntry)const
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
{
    return stricmp(m_name, phoneEntry.m_name) < 0;
}

bool PhoneEntry::operator>(const PhoneEntry &phoneEntry)const
{
    return stricmp(m_name, phoneEntry.m_name) > 0;
}

bool PhoneEntry::operator<=(const PhoneEntry &phoneEntry)const
{
    return stricmp(m_name, phoneEntry.m_name) <= 0;
}

bool PhoneEntry::operator>=(const PhoneEntry &phoneEntry)const
{
    return stricmp(m_name, phoneEntry.m_name) >= 0;
}

const char *PhoneEntry::get_name() const
{
    return m_name;
}

const char *PhoneEntry::get_phone() const
{
    return m_phone;
}

std::ostream &operator << (std::ostream &o, const PhoneEntry &p)
{
    o << p.m_name << ": " << p.m_phone;
    return o;
}

PhoneEntryPredicate::PhoneEntryPredicate()
{
    m_value = NULL;
    m_ignoreCase = m_byPhone = false;
}

PhoneEntryPredicate::PhoneEntryPredicate(const char*value, bool
ignoreCase, bool byPhone)
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
{
    m_value = strdup(value);
    m_ignoreCase = ignoreCase;
    m_byPhone = byPhone;
}

PhoneEntryPredicate::PhoneEntryPredicate(const PhoneEntryPredicate
&pred)
{
    m_value = strdup(pred.m_value);
    m_ignoreCase = pred.m_ignoreCase;
    m_byPhone = pred.m_byPhone;
}

bool PhoneEntryPredicate::operator()(const PhoneEntry &p)
{
    const char *value = m_byPhone ? p.get_phone() : p.get_name();
    if(m_ignoreCase) { return stricmp(m_value, value) == 0;}
    else { return strcmp(m_value, value) == 0;}
}

PhoneEntryPredicate::~PhoneEntryPredicate()
{
    free(m_value);
}
```

כפי שאפשר לראות מימשנו את אופרטורי ההשוואה השונים, כמו גם Copy Constructor ו-Default Constructor. בנוסף, כיוון שמתבצעת הקצאה דינמית של מחרוזות במחלקה זו - הגדרנו את ה-Destructor במחלקה.

7.5 האוסף list - רשימה מעגלית דו כיוונית

list היא מחלקת תבנית המממשת רשימה מקושרת מעגלית דו כיוונית. לעומת ה-vector, איברי list אינם רציפים בזיכרון, ולכן בכל פעם שיש צורך להקצות מקום לאיברים נוספים, אין צורך להעתיק את כל הבלוק הקודם כמו ב-vector. מאידך האיטרטור של ה-list הוא דו כיווני ואינו אקראי כמו זה של הווקטור, לכן התנועה באמצעות האיטרטור אפשרית קדימה או אחורה בלבד, ללא דילוגים.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

7.5.1 פונקציות של list

להלן רשימה חלקית של הפונקציות של list:

- **front()** - מחזירה התייחסות לאיבר הראשון ברשימה.
- **back()** - מחזירה התייחסות לאיבר האחרון ברשימה.
- **begin()** - מחזיר איטרטור המצביע על האיבר הראשון ברשימה.
- **end()** - מחזיר איטרטור המצביע על המיקום שאחרי האיבר האחרון ברשימה.
- **rbegin()** - מחזירה איטרטור הפוך לתחילת הרשימה (כלומר איטרטור זה מצביע למעשה על סוף הרשימה וקידומו ב-1 גורם לחזרה איבר אחד אחורה).
- **rend()** - מחזירה איטרטור הפוך לסוף הרשימה (כלומר איטרטור זה מצביע למעשה על תחילת הרשימה והפחתתו ב-1 גורם למעשה לקידומו לאיבר הבא ברשימה).
- **clear()** - מוחקת את כל האיברים מהרשימה
- **erase()** - מוחקת איבר אחד או קבוצה של איברים מהרשימה ומחזירה איטרטור לאיבר הראשון הנמצא אחרי האחרון שהוסר.
- **push_back()** - מוסיפה איבר אחד לסוף הרשימה.
- **push_front()** - מוסיפה איבר אחד לתחילת הרשימה.
- **pop_back()** - מסירה איבר אחד מסוף הרשימה.
- **pop_front()** - מסירה איבר אחד מתחילת הרשימה.
-
- **sort()** - ממיינת את איברי הרשימה.
- **unique()** - מסירה איברים רצופים זהים, ומשאירה במקומם מופע אחד. נהוג להשתמש בפונקציה זו לאחר הקריאה ל-()sort על מנת להסיר מופעים כפולים של איבר.
- **merge()** - ממזג את איברי הרשימה הממוינת המועברת אליה עם הרשימה הממוינת הנוכחית. האיברים מוסרים מהרשימה המועברת. הרשימה הנוכחית נותרת ממוינת לאחר המיזוג.
- **splice()** - מכניס את אברי הרשימה המועברת אל תוך הרשימה הנוכחית. האיברים מוסרים מהרשימה המקורית.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- **reverse()** - הופכת את סדר האיברים ברשימה.
- **swap()** - מחליף בין איברי שתי רשימות.

7.5.2 האלגוריתם find_if

האלגוריתם מחזיר איטרטור לאלמנט הראשון המקיים את פונקציית התנאי (Predicate) המועברת אליו.

אלגוריתם זה משתמש באיטרטורים המועברים אליו כדי להפעיל את ה-Predicate על כל האיברים בטווח. כאשר האלגוריתם נתקל באיבר הראשון עבורו ה-Predicate מתקיים, האלגוריתם מחזיר איטרטור המצביע לאותו איבר. אם האלגוריתם לא מצא איבר העונה ל-Predicate, מוחזר איטרטור המצביע על ()end.

יעילות האלגוריתם היא O(n).

דוגמה

בתוכנית הבאה ניצור שתי רשימות של טלפונים, נמיין אותן, נדפיס אותן, נהפוך בין איבריהן ונמזג אותן לרשימה אחת.

להלן קוד התוכנית (Ex03):

```
#include <iostream>
#include <list>
#include <algorithm>
#include "PhoneEntry.h"

using namespace std;

template<typename T, typename A = allocator<T>>
class list_utils
{
public:
    static void print(const char *listName, list<T,A> &l,
                      const char *separator = "\n")
    {
        /* example of forward iteration */
        cout << listName << ": ";
        list<T,A>::iterator iter = l.begin();
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
        if(iter != l.end())
        {
            cout << *iter++;;
        }
        while(iter != l.end())
        {
            cout << separator << *iter++;
        }
        cout << endl;
    }

    static void reverse_print(  const char *listName, list<T,A> &l,
                                const char *separator = "\n")
    {
        /* example of backward iteration */
        cout << listName << "(r): ";
        list<T,A>::reverse_iterator riter = l.rbegin();
        if(riter != l.rend())
        {
            cout << *riter++;
        }
        while(riter != l.rend())
        {
            cout << separator << *riter++;
        }
        cout << endl;
    }
};

int main(int argc, char **argv)
{
    list<PhoneEntry> l1;
    l1.push_back(PhoneEntry("Paul", "111111"));
    l1.push_back(PhoneEntry("Paula", "222222"));
    l1.push_back(PhoneEntry("Ilan", "4568721"));
    l1.push_back(PhoneEntry("Linda", "3216457"));
    l1.push_back(PhoneEntry("Dana", "666666"));
    l1.push_back(PhoneEntry("Ilan", "4545454"));
    list_utils<PhoneEntry>::print("list1", l1, " ; ");

    /* sort */
    l1.sort();
    list_utils<PhoneEntry>::print("list1", l1, " ; ");
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    /* removes duplicats */
    l1.unique();
    list_utils<PhoneEntry>::print("list1", l1, " ; ");

    /* creates another list */
    list<PhoneEntry> l2;
    l2.push_front(PhoneEntry("Goerge", "3521654"));
    l2.push_front(PhoneEntry("Roni", "6547895"));
    l2.push_front(PhoneEntry("Aya", "5689755"));

    /* swaps elements */
    l1.swap(l2);
    list_utils<PhoneEntry>::print("list1", l1, " ; ");
    list_utils<PhoneEntry>::print("list2", l2, " ; ");

    /* merge lists */
    l1.sort();
    l2.sort();
    l1.merge(l2);
    list_utils<PhoneEntry>::print("list1", l1, " ; ");
    list_utils<PhoneEntry>::print("list2", l2, " ; ");

    /* search by a name */
    list<PhoneEntry>::iterator searchIter = find_if(l1.begin(),
l1.end(),
                                    PhoneEntryPredicate("ilan",
true));
    if(searchIter != l1.end())
    {
        cout << "Found: " << *searchIter << endl;
    }
    else
    {
        cout << "Not found!" << endl;
    }

    /* search by a phone */
    searchIter = find_if(l1.begin(), l1.end(),
                     PhoneEntryPredicate("3216457", true, true));
    if(searchIter != l1.end())
    {
        cout << "Found: " << *searchIter << endl;
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    }
    else
    {
        cout << "Not found!" << endl;
    }

    /* another search */
    searchIter = find_if(l1.begin(), l1.end(),
PhoneEntryPredicate("Orna", true));
    if(searchIter != l1.end())
    {
        cout << "Found: " << *searchIter << endl;
    }
    else
    {
        cout << "Not found!" << endl;
    }
}
```

פלט התוכנית הוא:

```
list1: Paul: 111111 ; Paula: 222222 ; Ilan: 4568721 ; Linda: 3216457
; Dana: 666666; Ilan: 4545454
list1: Dana: 666666 ; Ilan: 4568721 ; Ilan: 4545454 ; Linda: 3216457
; Paul: 111111; Paula: 222222
list1: Dana: 666666 ; Ilan: 4568721 ; Linda: 3216457 ; Paul: 111111
; Paula: 222222
list1: Aya: 5689755 ; Roni: 6547895 ; Goerge: 3521654
list2: Dana: 666666 ; Ilan: 4568721 ; Linda: 3216457 ; Paul: 111111
; Paula: 222222
list1: Aya: 5689755 ; Dana: 666666 ; Goerge: 3521654 ; Ilan: 4568721
; Linda: 3216457; Paul: 111111 ; Paula: 222222 ; Roni: 6547895
list2:
Found: Ilan: 4568721
Found: Linda: 3216457
Not found!
```

הערות

- כפי שאפשר לראות, לפני שמיזגנו את הרשימות היינו חייבים למיין אותן.
- עבור החיפוש, השתמשנו באלגוריתם find_if המבצע בדיקה סדרתית.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- בנינו את האיטרטור כך שנוכל להגדיר ב-Constructor מה הערך אותו רוצים לחפש, האם זה שם או טלפון, והאם ההשוואה צריכה להתעלם מגודל האות. השתמשו במאפיינים אלה ביצירת ה-Predicates בכל אחת מהקריאות.

7.6 האוסף deque - תור כפול (Double ended queue)

תור כפול דומה מאוד מבחינת הממשק ל-vector, אלא שב-deque אפשר גם להוסיף אלמנטים ולהסיר אותם בתחילת האוסף (push_front ו-pop_front) ולא רק מהסוף (push_back ו-pop_back). באופן פנימי, צורות המימוש של vector ו-deque שונות באופן מהותי:

- vector ממומש באמצעות מערך איברים רציף בזיכרון. כאשר הוספת איבר חורגת מהקיבולת שכבר הוקצתה למערך (capacity), מתבצעת הקצאה חדשה, העברת האיברים למערך החדש והוספת האיבר החדש. היתרון הוא שמובטחת לנו רציפות האיברים בזיכרון במקרה שאנו רוצים לבצע חשבון מצביעים על אזור הזיכרון בו מאוחסנים איברי ה-vector (שים לב שהאיטרטור של vector מאפשר פעולות הדומות לפעולות על מצביע, ובחלק מהמימושים הוא אף מצביע ממש).
- deque ממומש באמצעות מערך מצביעים המצביעים על בלוקים. כל בלוק זיכרון כזה הוא מערך המכיל מספר אלמנטים. מבחינתנו - המימוש נסתר ולנו נראה כאילו מדובר במערך אחד. המימוש הזה מאפשר שיפור ביצועים ניכר ביחס למערך כאשר מוסיפים ומסירים איברים מסוף המערך או תחילתו מספר רב של פעמים. במקרה זה יידרשו הרבה פחות הקצאות זיכרון ופעולות העתקה מאשר ב-vector, כיוון שכל שיש לעשות הוא ליצור בלוק חדש ולעדכן את טבלת המצביעים. החיסרון הוא שאין בידינו ערובה לרציפות התאים בזיכרון במקרה שאנו רוצים לבצע חשבון מצביעים על הזיכרון בו מאוחסן ה-deque (שים לב שגם האיטרטור של deque מאפשר פעולות הדומות לפעולות על מצביע, אך בשונה מ-vector, האיברים אינם בהכרח רציפים בזיכרון, ולכן לא ניתן להתייחס אליהם כאל מערך רציף באמצעות חשבון מצביעים).

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

במקרה של הוספת איברים באמצע האוסף, הן deque והן vector עשויים להידרש להזזת איברים רבים. לעומתם, list מאפשרת הוספה יעילה במיקום נתון כאשר כבר קיים איטרטור למיקום ההוספה.

7.6.1 פונקציות של deque

להלן רשימה חלקית של הפונקציות ב-deque:

- **()begin** - מחזירה איטרטור לתחילת ה-deque.
- **()end** - מחזירה איטרטור לסוף ה-deque.
- **()rbegin** - מחזירה איטרטור הפוך לתחילת האוסף. כלומר איטרטור המצביע על סוף האוסף וקידומו הוא למעשה חזרה לאיבר הקודם.
- **()rend** - מחזירה איטרטור הפוך לסוף האוסף. כלומר איטרטור המצביע על תחילת האוסף והפחתתו היא למעשה התקדמות לאיבר הבא.
- **()size** - מחזירה את גודלו העדכני של ה-deque.
- **()max_size** - מחזירה את הגודל המקסימלי האפשרי של ה-deque.
- **()resize** - משנה את גודל ה-deque ומוסיפה איברים במידת הצורך.
- **()empty** - מחזירה true אם ה-deque ריק. אחרת מחזירה false.
- **[]operator** - מחזירה ייחוס לאלמנט ב-deque לפי אינדקס.
- **()at** - מחזירה ייחוס לאלמנט ב-deque לפי אינדקס.
- **()front** - מחזיר ייחוס לאלמנט הראשון ב-deque.
- **()back** - מחזירה ייחוס לאלמנט האחרון ב-deque.
- **()assign** - מחליפה את תוכן ה-deque בערכים שנמסרו לה.
- **()push_back** - מוסיפה אלמנט לסוף ה-deque.
- **()push_front** - מוסיפה אלמנט לתחילת ה-deque.
- **()pop_back** - מסירה את האלמנט מסוף ה-deque.
- **()pop_front** - מסירה את האלמנט מתחילת ה-deque.
- **()insert** - מוסיפה אלמנט/ים ל-deque במיקום שמועבר.
- **()erase** - מוחקת אלמנטים מה-deque.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

- **swap()** - מחליפה בין תוכנם של שני deque.
- **clear()** - מוחקת את כל איברי ה-deque.

7.6.2 האלגוריתם stable_sort

אלגוריתם זה ממיין באופן יציב את ה-container המועבר אליו. יציבות המיון מתבטאת בכך ששני איברים בעלי ערך זהה שומרים על מיקומם היחסי כמו לפני המיון. מיון זה משתמש באופרטור < (קטן מ) או בפונקציית ה-predicate המועברת אליו כפרמטר.

דוגמה

```
#include <iostream>
#include <deque>
#include <algorithm>

using namespace std;

template<typename T, typename A = allocator<T>>
class deque_utils
{
public:
    static void print(const char *name, deque<T,A> &c,
                      const char *separator = "\n")
    {
        /* example of forward iteration */
        cout << name << ": ";
        deque<T,A>::iterator iter = c.begin();
        if(iter != c.end())
        {
            cout << *iter++;;
        }
        while(iter != c.end())
        {
            cout << separator << *iter++;
        }
        cout << endl;
    }

    static void reverse_print(const char *name, deque<T,A> &c,
                              const char *separator = "\n")
    {
```


© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
        /* example of backward iteration */
        cout << name << "(r): ";
        deque<T,A>::reverse_iterator riter = c.rbegin();
        if(riter != c.rend())
        {
            cout << *riter++;
        }
        while(riter != c.rend())
        {
            cout << separator << *riter++;
        }
        cout << endl;
    }
};

bool less_ints(int a, int b);

int main(int argc, char **argv)
{
    deque<char> d1, d2;
    for(int i = 0; i < 5; ++i)
    {
        d1.push_back('A' + i);
    }
    for(int i = 0; i < 5; ++i)
    {
        d2.push_front('A' + i);
    }
    deque_utils<char>::print("d1", d1, ", ");
    deque_utils<char>::print("d2", d2, ", ");

    /* copy d2 to the end of d1 */
    d1.insert(d1.end(), d2.begin(), d2.end());
    deque_utils<char>::print("d1", d1, ", ");
    deque_utils<char>::print("d2", d2, ", ");

    /* revese sort */
    stable_sort(d1.rbegin(), d1.rend(), less_ints);
    deque_utils<char>::print("d1", d1, ", ");
}

bool less_ints(int a, int b)
{
```

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il

```
    return a < b;
}
```

פלט התוכנית הוא:

```
d1: A, B, C, D, E
d2: E, D, C, B, A
d1: A, B, C, D, E, E, D, C, B, A
d2: E, D, C, B, A
d1: E, E, D, D, C, C, B, B, A, A
```

הערות

- השתמשנו בפונקציה push_front על מנת להכניס איברים לתחילת התור. היעילות של פעולה זו, כפי שהסברנו, היא מצוינת.
- בהמשך העתקנו את איברי d2 ל-d1.
- בסוף השתמשנו ב-stable_sort על מנת למיין את התור d1. כאמור, מיון זה שומר על הסדר היחסי של איברים בעלי ערך זהה. כמובן שבמקרה של integers מיותר לשמור על יציבות, אבל במקרה של אובייקטים מורכבים יותר ייתכן שיציבות המיון תהיה קריטית לתהליך תקין.
- האופרטור > (קטן מ) מוגדר עבור מספרים שלמים, כך שיכולנו להפעיל את המיון ללא צורך בהעברת הפונקציה less_ints. בכל אופן, לצורך ההמחשה בלבד אנו רואים כי גם פונקציה יכולה לשמש כ-Predicate ולא רק אובייקט פונקציה. כלומר אם אפשר להפעיל את האופרטור () על ה-Predicate אזי הוא תקין.
- Predicate של אלגוריתם מיון מקבל שני איברים וצריך להחזיר true אם הראשון קטן מהשני.

© 2010 - כל הזכויות שמורות לאילן אמויאל
אין להעתיק ללא רשות בכתב מאת המחבר
kita.co.il