סיכום לשפת $C++$

Hello class

```
// This line defines standard I/O library
#include <iostream>   // notice no .h here
int main() {
        std::cout  <<  "Hello class!\n";
        return 0;
}
```

בשונה מ-$stdio.h$ של C, ב-CPP נוסיף את $\#include\ \langle iostream\rangle$ כדי לאפשר פעולות קלט/פלט.

בקוד הנ"ל הדפסנו את "$Hello\ class!\backslash n$" לסטרים $std::cout$ (מקביל ל-$stdout$ ב-C)

הסימן >> מסמן שמדובר ב-$stream$, וכיוון החצים מסמנים שזורם מידע כלפי חוץ

(כלומר מידע מהתוכנית שלנו זורם לסטרים חיצוני)

הקידומת $std::$ מציינת שאנחנו נשתמש ב-$namespace$ של הספרייה הסטנדרטית,

במקרה הזה $cout$ (שמציין את שם ה-$standart\ output\ stream$)

קימפול והרצה

g++ -Wall hello.cpp -o hello – פקודה זו מקמפלת את הקובץ $hello.cpp$ לפי הסטנדרט הדיפולטיבי (על מחשבי האקוריום זה סטנדרט 14)

g++ -Wall -std=c++17 hello.cpp -o hello – בצורה זו נקמפל בעזרת סטנדרט ספציפי (17).

<u>הערה</u>: ב-2 המקרים קראנו ישירות לקומפיילר $g++$ כדי לקמפל קובץ CPP אך יכלנו לרשום גם gcc ובגלל שמדובר בקובץ cpp, הוא ידע להפעיל לבד את $g++$.

String and Boolean

ב-CPP להבדיל מ-C, הטיפוסים $bool$ ו-$string$ הם חלק מהשפה.

($bool$ לא מצריך $\#include$ כלשהו אבל $string$ מצריך $\#include\ \langle string \rangle$)

נראה דוגמה לשימוש ב-$bool$:

```
#include <iostream>
int main() {
        int a = 5;
        bool isZero = (a == 0);
        // all are same condition
        if ( isZero == false && !isZero && isZero != true && !!!isZero && a ) {
                        std::cout << "a is not zero\n";
        }
}
```

<u>הערה</u>: נשים לב שגם a הוא תנאי תקין כמו שהיה ב-C (שונה מ-0 ולכן $true$)

נראה דוגמה לשימוש ב-$String$:

```
#include <iostream>
#include <string> // part of the standard library
int main() {
        std::string str1 = "Hello";
        std::string str2("world");
        std::cout << str1 + str2 << std::endl;
        return 0;
}
```

<u>הערות</u>:

- 2 השורות הראשונות ב-$main$ מייצרות מחרוזות.
- השורה השלישית מדפיסה את החיבור שלהן (בעזרת האופרטור +)
- כדי להוסיף "$\backslash n$" אפשר להשתמש ב-$std::endl$
- פלט התוכנית הוא $Hello\ world$
- מחרוזת היא רצף של תווים המוגדר בין מרכאות והוא כולל '$\backslash 0$' בסופו (בדיוק כמו ב-C)
- שינוי מחרוזת יגרור התנהגות בלתי מוגדרת. מחרוזות יכולות להיות מאוחסנות במקטע זיכרון המוגדר לקריאה בלבד, או שהן יכולות להיות משולבות עם מחרוזות נוספות.
- נוכל להגדיר מחרוזות עם מצביעים כמו ב-C, אך מומלץ להשתמש במחלקה $string$ ולהימנע מטיפול מצביעים ככל שניתן. (המחלקה $string$ עושה הכל בשבילנו)

Streams

ראינו שימוש ב-$cout$ :: std, כעת נראה סטרימים נוספים:

```
#include <iostream>
#include <string>
int main() {
        std::string str;
        int a;
        double b;
        std::cin >> str >> a >> b;
        if (std::cin.fail()) {
                std::cerr << "input problem\n";
                return 1;
        }
        std::cout << "I got:" << str << ' ' << a << ' ' << b << std::endl;
}
```

<u>הערות</u>:

- כדי לקרוא קלט מהמשתמש למשתנים str, a, b, השתמשנו בסימון ההפוך >> , כלומר קראנו מידע מהסטרים לתוך המשתנים הנ"ל.
  str יקבל את הביטוי הראשון (עד לרווח או שורה חדשה), a את השני ו-b את השלישי.
- הוספנו בדיקה האם ניסיון הקריאה כשל.

<u>דוגמה להרצה</u>:

עבור הקלט 5.0 5 $string$ יתקיים: $str = string, a = 5, b = 5$

עבור הקלט $string55.0$ (בלי רווחים) יתקיים:

$str = string55.0$ ואז הוא יחכה ל-2 קלטים נוספים עבור a, b

עבור הקלט $string\ string\ string$ (3 מחרוזות) תודפס שגיאה.

<u>הערה</u>: הקלט שנמצא ב-$std::cin$ יופרד למשתנים לפי רווחים, לכן אם נרצה לתת כקלט מחרוזת שמכילה רווחים, נצטרך להשתמש ב-$getline$ ולא בשיטה הנ"ל.

משתנה מסוג רפרנס

בנוסף למשתנים פרימיטיביים ומצביעים שראינו ב-C, ב-$C++$ קיים משתנה מסוג נוסף.

רפרנס הוא קישור / שם נוסף למשתנה קיים. נראה דוגמה:

```
int i = 10;
int& ref = i; // ref is an int reference
              // initialized at definition, only once!
ref += 5; // changes both ref and i's value
```

יצרנו משתנה בשם ref מסוג רפרנס (בעזרת ה-&), ובכך הגדרנו ל-i שם נוסף.

השורה האחרונה תוסיף 5 גם ל-ref וגם ל-i.

השימוש העיקרי ברפרנס הינו העברת ארגומנטים לפונקציות וערכי החזרה שלהן.

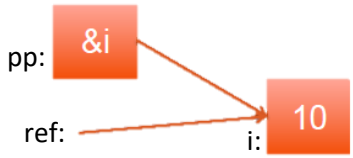
חייב לאתחל משתנה מסוג רפרנס כבר בהכרזה, כלומר $int\&\ ref;$ יגרור שגיאה.

לא ניתן להצביע למשתנה מסוג רפרנס, לא ניתן לייצר מערכים של רפרנסים.

ניתן לחשוב על משתנה מסוג זה כמצביע **קבוע** למשתנה.

מרגע הגדרתו, משתנה מסוג רפרנס לא יכול להצביע על משהו אחר.

```
int i = 10;
int& ref = i;
int *pp = &ref;
```



Pointer vs Reference

```
void swap(int *a, int *b) {
int temp = *a;
        *a = *b;
        *b = temp;
}
...
int main() {
        int x = 3, y = 7;
        swap(&x, &y);
        // x == 7, y == 3
}
```

```
void swap(int& a, int& b) {
int temp = a;
        a = b;
        b = temp;
}
...
int main() {
        int x = 3, y = 7;
        swap(x, y);
        // x == 7, y == 3
}
```

את הגרסה השמאלית אנחנו מכירים מ-C, ביצוע $Swap$ בעזרת מצביעים.

ב-CPP נוכל לשלוח רפרנסים למשתנים, ולהשתמש בהם כרגיל ללא מצביעים. (הדוגמה הימנית)

נשים לב שבדוגמה השמאלית היינו צריכים לשלוח כתובת, ובימנית שלחנו את המשתנים עצמם.

הערות לגבי רפרנסים:

- חייבים להתייחס תמיד לאובייקטים קיימים (כאלה שהוקצה להם מקום בזיכרון)
- לאחר הגדרה, לא ניתן לגרום לרפרנס להצביע לאובייקט אחר.
- ניתן להתייחס לרפרנסים כאילו הם האובייקטים עצמם.
- לרוב ממומש על ידי הקומפיילר כמצביעים קבועים.

```
int i = 3;
int j = 5;
int& ref = i;
std::cout << i << std::endl << j << "\n" << ref << std::endl;
ref = j;
std::cout << i << std::endl << j << "\n" << ref << std::endl;
j = 7;
std::cout << i << std::endl << j << "\n" << ref << std::endl;
```

ההדפסה הראשונה תדפיס $3\backslash n5\backslash n3\backslash n$

השורה $ref = j;$ תבצע השמה של 5 ב-i כי זה בדיוק כמו לעשות $i = j;$

לכן ההדפסה השנייה תדפיס $5\backslash n5\backslash n5\backslash n$

וההדפסה השלישית תדפיס $5\backslash n7\backslash n5\backslash n$

נסכם:

```
int *func(int *var0, int& var1, int var2);
```

Using Pointer — By reference — By Value

למה להשתמש ברפרנס?

- יעילות – נמנע מהעתקת ארגומנטים.
- מאפשר שינוי משתנים מחוץ לתחום הפונקציה.

(כל דבר שניתן לעשות עם רפרנסים, ניתן לעשות גם עם מצביעים,
אך עם מצביעים יש יותר מקום לטעויות)

- לקומפיילר קל יותר לבצע אופטימיזציה לרפרנסים.
- נוח יותר בהרבה מקרים.
- נפוץ בעיקר לשליחת ארגומנטים וכערכי החזרה.

הערה: בדיוק כמו עם מצביעים, לא נשלח רפרנסים למשתנים לוקאליים !

מבנים/מחלקות ורפרנסים

```
void add(Point& a, Point b) {
        // a is reference, b is a copy
        a._x += b._x;
        a._y += b._y;
}
int main() {
        Point p1(2,3), p2(4,5);
        add(p1, p2); // note: we don't send pointers!
        // p1 is now (6,8)
        ...
}
```

בדוגמה הנ"ל שלחנו רפרנס ל-p_1 ועותק ל-p_2.

הקוד עובד בצורה נכונה כי לא ביצענו שינוי ב-p_2 אלא רק ב-p_1.

למרות זאת, נעדיף לשלוח את p_2 כרפרנס קבוע ($const$). גם כדי למנוע העתקה מיותרת,

וגם כדי שהקומפיילר יוודא עוברנו שלא ביצענו שינוי ב-p_2 בטעות. השינוי הנדרש ב-add הינו:

```
void add(Point& a, const Point& b) {
        // a is reference,  b is a const ref
        a._x+= b._x;
        a._y+= b._y;
}
```

רפרנס כערך החזרה

```
Point& add(Point& a, const Point& b) {
        // a is reference, b is a const ref
        a._x+= b._x;
        a._y+= b._y;
        return a;
}
int main() {
        Point p1(2,3), p2(4,5), p3(0,1);
        add(add(p1, p2), p3); // now p1 is (6,9)
        cout << add(p1, p2).getX(); // note the syntax
        ...
}
```

כלומר ערך ההחזרה של השיטה add הינו $Point\&$.

פיצ'רים חדשים של $C++11$

$nullptr$

ב-C השתמשנו במאקרו $NULL$ שהוא בעצם קבוע 0 עם קאסט ל-$void*$

$$\#define\ NULL\ \ ((void*)0)$$

ב-$C++11$ קיים מצביע $NULL$ אמיתי, והוא נקרא $nullptr$.

לולאת $Range$

```
#include <iostream>
int main() {
        int arr[] = {1,2,10,4};
        for(int i : arr) {
                std::cout << i << " ";
        }
        std::cout << std::endl;
        return 0;
}
OUTPUT: 1 2 10 4
```

בדיוק כמו בפייתון, ניתן לרוץ בלולאות על איברי המערך.

כלומר i בדוגמה הנ"ל יהיה בסיבוב ה-j הערך שיושב במערך במקום ה-$j-1$.

אם נצטרך גם קאונטר, נתחזק קאונטר חיצוני או שנשתמש ב-for רגיל.

לולאת $Range$ עובדת עבור מערכים וכל מבנה נתונים אחר שמממש את הפונקציות $begin()$ ו-$end()$, ומחזיר איטרטור (מוסבר בהמשך). לא עובד עם מצביעים.

מחלקות

ב-C ראינו את הפתרון הבא לבעיית ה-$Draw\ shape$:

```
typedef struct Shape {
        enum { RECTANGLE, CIRCLE, TRIANGLE } _type;
        double _x, _y;
        double _height, _width;
} Shape;

void Draw( Shape const *shape) {
        switch( shape-> _type ) {
        case RECTANGLE:
        ...
        case CIRCLE:
        ...
```

<u>יתרונות</u>: פשוט, ישיר

<u>חסרונות</u>: הוספת צורות חדשות דורשת שינויים בהרבה מקומות בקוד.

בנוסף יש שימוש מאסיבי של $if/switch$ שמכביד על הקוד.

כשלמדנו מצביעים לפונקציות, הוספנו לכל $Shape$ שדה עם מצביע לפונקציה שמציירת אותו, וככה ידענו איזה פונקציה להפעיל למבנה מסוים.

<u>יתרונות</u>: ניתן להרחבה, אינקפסולציה של שיטות הציור, יעיל

<u>חסרונות</u>: יכול לגרום לשגיאות של חוסר התאמה בין הפונקציה לבין הקלט.

```
shape1-> draw = draw_shape2;
(*shape1-> draw)(shape3);
```

חסרונות אפשריים למבנים באופן כללי:

- גישה פומבית לכל המידע הפנימי של המבנה.
- מי שמשתמש במבנה צריך לאתחל ידנית את כל המשתנים שלו.
- אין חבילה אחת המכילה גם את המבנה וגם את השיטות הקשורות אליו.

נתחיל ללמוד כעת על מחלקות, ובהמשך הסיכום נראה פתרון נוסף לבעיה הנ"ל.

נראה דוגמה להכרזה על מחלקה (קובץ $counter.h$)

```
#ifndef _COUNTER_H_
#define _COUNTER_H_
class Counter { // capital 'C'
public:
        Counter(); // Constructor
        void increment(); // A method
        int value(); // A method
private:
        int _count;
};
#endif // _COUNTER_H_
```

מימוש המחלקה (קובץ $counter.cpp$):

```
#include "Counter.h"
Counter::Counter() {
        _count = 0;
}
void Counter::increment() {
        _count++;
}
int Counter::value() {
        return _count;
}
```

שימוש במחלקה (קובץ $app.cpp$):

```
#include <iostream>
#include "Counter.h"
int main() {
        Counter cnt; // constructor!
        std::cout << "Initial value = " << cnt.value() << std::endl;
        cnt.increment();
        std::cout << "New value = " << cnt.value() << std::endl;
}
```

<u>הערות</u>:

- המשתנה cnt מאוחסן על ה-$stack$.
- הפונקציה $increment()$ (למשל) היא שורה אחת, לכן נוכל לממש אותה כבר ב-$header$, ופונקציות **של מחלקה** שממומשות בקובץ $header$ עצמו יסומנו כפונקציית $inline$ (כפי שלמדנו ב-C). בפועל ניתן לממש גם פונקציות ארוכות יותר ב-$header$, אך כלל אצבע יהיה שאם מדובר בפונקציה עם מספר שורות נממש אותן ב-cpp. המימוש ב-$header$ יראה כך: $void\ increment() \{ \_count++; \}$ (ללא $inline$ כי זה מתווסף לבד אם זה ב-$header$)

Public/Private - מילים שמורות המציינות לאיזה משתנים ופונקציות ניתן לגשת מבחוץ.

כל מה שיוגדר מתחת ל-*public* יהיה פומבי ונגיש לכל מי שמשתמש במחלקה.

כל מה שתחת *private* זמין רק לשיטות מתוך המחלקה עצמה.

ההבדל בין מחלקה למבנה:

ב-$C++$ מחלקות ומבנים פועלים באותה צורה, אך כברירת מחדל, כל עוד לא רשמנו *private* ידנית, משתנים של מבנה יהיו *public*, ומשתנים של מחלקה יהיו *private*!

לאן נעלם ה-*typedef*?

ב-C היינו צריכים לעשות *typedef struct MyStruct MyStruct* כדי לתת שם למבנה שלנו, אך ב-$C++$ פעולה זו מתבצעת באופן אוטומטי. כלומר:

```
struct MyStruct {
        int x;
};
```

מגדיר גם את שם המבנה, ונוכל פשוט לייצר טיפוסים מסוג *MyStruct*.

עדיין אפשר להשתמש ב-*typedef* במקרים אחרים, הוא קיים גם ב-$C++$.

במקרה של התנגשות בשמות של משתנים, נוכל להשתמש ב-"*class*" או "*struct*":

```
class A{};
int main() {
        int A = 5;
        A a; // ERROR
        class A a; // OK
}
```

פונקציות *const*:

```
class Point {
public:
        Point(int x, int y);
        ~Point();
        int getX() const;
        int getY() const;
private:
        int _x, _y;
};
```

בדוגמה זו הגדרנו את הפונקציות *getX()* ו-*getY()* כ-*const*, ובכך יידענו את הקומפיילר שהפונקציות האלה לא אמורות לשנות שום דבר בתוך המחלקה. (תהיה שגיאת קומפילציה אם כן) נשאף להגדיר כל פונקציה כזאת (שלא משנה כלום) כ-*const*.

Overloading

ב-CPP (בשונה מ-C) ניתן לממש מספר פונקציות עם אותו שם, בתנאי שמספר הארגומנטים שלהן או הסדר שלהם שונה.

פונקציות שרק ערך ההחזרה שלהן שונה לא מבצעות $overloading$ ותהיה שגיאת קומפילציה !

```
#include <iostream>
void foo() {
        std::cout << "foo()\n";
}
void foo(int n) {
        std::cout << "foo(" << n << ")\n";
}
```

הקוד הנ"ל יתקמפל.

<u>default parameters</u>:

שיטות ב-CPP יכולות להגדיר ארגומנטים דיפולטיביים (ראינו ב-$python$),

כך שנוכל לבחור בקריאה לפונקציה אם לשנות אותם או לא. נראה דוגמה:

```
#include <iostream>
void foo(int a, int b = 5, int c = 3) {
        std::cout << a << " " << b << " " << c << std::endl;
}
int main() {
        foo(3); // output: 3 5 3
        foo(3,1); // output: 3 1 3
        foo(3,1,2); // output: 3 1 2
}
```

<u>הערה</u>: כל הפרמטרים הדיפולטיביים חייבים להיות מוגדרים כפרמטרים הימניים ביותר.

כלומר $void\ foo(int\ a = 3, int\ b, int\ c = 3)$ לא תקין !

<u>שיטה ללא ארגומנטים</u>:

ב-C למדנו ששיטה עם סוגריים ריקים מגדירה מספר לא ידוע של משתנים.

אם היינו רוצים להגדיר פונקציה שלא מקבלת משתנים בכלל, היינו רושמים $int\ f(void)$;

ב-$C++$ זה כבר לא המצב. סוגריים ריקים יהיו בדיוק כמו לרשום $int\ f(void)$;.

```
int f(void); // int f(void) in both C and C++
int f(); // int f(void) in C++, int f( unknown ) in C
```

overload resolution:

נניח שיש לנו משתנה a מטיפוס A שניתן להמיר אותו ל-$short$, ונניח שיש לנו את הפונקציות:

$void\ foo(int);\quad void\ foo(double);$

איזו פונקציה תיבחר בקריאה $foo(a);$?

תהליך בחירת הפונקציה נקרא $overload\ resolution$ והוא פועל כך:

1. נמצא את כל הפונקציות עם שם הפונקציה.
2. מתוך הפונקציות הנ"ל, נמצא את הפונקציות עם מספר הארגומנטים המתאים.
3. מתוך הפונקציות הנ"ל, נמצא את הפונקציות שה-$casting$ שהן יבצעו יהיה המתאים ביותר.
4. אם נשארנו עם פונקציה אחת, נריץ אותה. אחרת (0 או יותר מ-1) נזרוק שגיאת קומפילציה.

שלב 3 בודק עבור איזה פונקציות מתבצע כמה שפחות $implicit\ casting$,

בנוסף נרצה שה-$cast$ יהיה "קטן" ככל שניתן.

למשל $cast$ מ-$short$ ל-int יהיה טוב יותר מ-$short$ ל-$long$. נראה דוגמאות:

```
double add( int, double);
double add( double, int x = 4);
double add( double, double);
add( 1, 1.5 ); // Calls first function
add( 1.2, 1.5 ); // Calls third function
add( 1.2 ); // Calls second function
add( 1 ); // Calls second function
add( 1, 4 ); // Compilation Error!
```

האפשרות האחרונה תגרום לשגיאת קומפילציה, כי לאחר תהליך הסינון נישאר עם 2 פונקציות שמתאימות לפרמטרים 1,4 והקומפיילר לא ידע במה לבחור. דוגמה נוספת:

```
void func(int a, int b, int c, double d) {
        std::cout << 1 << std::endl;
}
void func(int a, double b, double c, double d) {
        std::cout << 2 << std::endl;
}
int main() {
        func(1,2,3,4); // 1 will be printed
}
```

friend

מחלקה יכולה להגדיר מחלקה אחרת כ"חברה" שלה, ובכך לתת לה גישה למשתנים הפרטיים שלה.

```
class A {
        int a; // private by default
        friend class B;
}
class B {
        int getA(A a) {
                return a.a;
        }
}
```

הקוד הנ"ל מתקמפל ותקין (גישה למשתנה a של המחלקה a למרות שהוא כברירת מחדל $private$)
וזה בגלל שהגדרנו את B כ-$friend$. נשתדל להימנע מזה כל עוד אפשר כי זה שובר את
האינקופסולציה, אך יהיו מקרים בהם אין ברירה. (נראה בהמשך)
ניתן להגדיר גם $friend\ int\ getA(\ A\ a\ );$ במקום $friend\ class\ B;$
וכך לתת רק לשיטה $getA$ גישה למשתנים הפרטיים של המחלקה.

this

כשאנחנו מייצרים מופע של מחלקה, כל מתודה פועלת על המופע הספציפי, אך כדי לאפשר זאת,
הקומפיילר מבצע שינויים בזמן קומפילציה. כשאנחנו מממשים פונקציה כזו:

```
void S :: increment() {
        _a++;
}
```

הקומפיילר משנה אותה אחר-כך ל:

```
void S :: increment(S *this) {
        this-> _a++;
}
```

כאשר $this$ הוא מצביע למופע הרצוי.
בדומה ל-$java$, נוכל להשתמש ב-$this$ גם באופן מפורש כדי להתייחס לאובייקט הנוכחי.

Static

ב-C , השתמשנו במילה השמורה $static$ כדי לחסום גישה לשיטה / משתנה מקובץ אחר.

ב-$C++$ (בדומה ל-$Java$) המילה השמורה $static$ מאפשרת פונקציונליות אחרת:

משתנים סטטיים ומתודות סטטיות במחלקה יהיו מקושרים למחלקה ולא למופע ספציפי.

```
//h file
class List {
public:
        static int getMaxSize(); //isn't related to an instance
        int getSize();
        static int max_size;// only init static in − class if it's a const integral type
        int _size = 0; //can init non − static (C++11).
};
//cpp file
int List::max_size = 10;
```

כלומר בקובץ h נצהיר על משתנים סטטיים אך נוכל לאתחל אותם רק בקובץ cpp

(למעט מקרה בודד שבו המשתנה הסטטי הוא מסוג $const\ int$)

הגישה למשתנים היא בעזרת "שם המחלקה :: שם המשתנה".

כעת נראה שימוש של פונקציות סטטיות.

```
int List::getMaxSize() { // no 'static' in static member def
        return this−> _size; // compilation error!
        return max_size; // ok
}
int List::getSize() {
        return this−> _size * _size; //ok
}
int main() {
        List l;
        l.getSize();
        List::getMaxSize();
        l.getMaxSize();
}
```

בקובץ CPP לא נוסיף $static$ לפני חתימת הפונקציה. (רק בקובץ $header$)

כדי לקרוא לפונקציה סטטית, לא חייב לייצר מופע של המחלקה

(ניתן לקרוא לפונקציה דרך מופע קיים אך נעשה זאת לרוב עם שם המחלקה)

בנאים

- ניתן לייצר מספר בנאים (שמקבלים מספר שונה של משתנים / סדר שונה של טיפוסים)
- בנאי הוא פונקציה מיוחדת של המחלקה שמאתחלת את האובייקט ברגע שהוא נוצר.
- לבנאי יש את אותו השם כמו למחלקה, ואין לו ערך החזרה (וכך הקומפיילר מזהה אותו).
- בנאי דיפולטיבי הוא בנאי שלא מקבל שום פרמטר.

אם לא הגדרנו בנאים בכלל במחלקה, "נקבל" מהקומפיילר בנאי דיפולטיבי במתנה.

<u>הערה חשובה</u>:

כשאנחנו מייצרים מופע חדש ונכנסים לבנאי שלו, דבר ראשון אנחנו מייצרים מופעים חדשים של השדות שלו ורק אז אנחנו ממשיכים הלאה למימוש עצמו של הבנאי. נראה דוגמה:

```
class Point {
public:
        Point() {
                std::cout << "Point - default ctor\n";
        }
};

class Segment {
        Point _p1, _p2;
public:
        Segment() {
                std::cout << "Segment default ctor\n";
        }
};
```

עבור $int\ main()\ \{\ Segment\ s;\ \}$ יודפס:

```
Point - default ctor
Point - default ctor
Segment - default ctor
```

כלומר קודם נכנסנו לבנאי של $Point$ עבור $\_p1$ ו-$\_p2$ ורק אז לבנאי של $Segment$.

פעולה זו אפשרית רק בגלל שקיים הבנאי הדיפולטיבי. (בנאי ללא ארגומנטים)

אם נשנה את הבנאי של $Point$ לקבל משתנה מסוג int, נקבל שגיאת קומפילציה,

כי הקומפיילר לא ידע לאיזה בנאי לפנות כדי להגדיר את $\_p1$ ו-$\_p2$.

נוכל לטפל בבעיה הזו בעזרת רשימת איתחול המוסברת בעמוד הבא.

member initializer list:

בנאים מאפשרים לאתחל משתנים בעזרת סינטקס מיוחד:

```
Segment(int i, int j) : _p1(i), _p2(j) {
        ...
}
```

בצורה זו אנחנו פונים לבנאי הנכון, ומונעים את שגיאת הקומפילציה שהסברנו בעמוד קודם.

(ניתן לקרוא לבנאים הנכונים גם בתוך הבנאי)

במקרים בהם המשתנים הם *const* או *reference*, נהיה **חייבים** לאתחל אותם ברשימת האתחול.

ניסיון לאתחל אותם בתוך הבנאי יגרום לשגיאת קומפילציה.

הערות:

- מאפשר אתחול שדות האובייקט.
- הדרך היחידה לאתחל קבועים ורפרנסים.
- בהמשך נלמד: מאפשר אתחול מחלקות אב.
- מהיר יותר ובטוח יותר להשתמש ב-*initialization list* מאשר אתחול בתוך הבנאי.
- החל מ-$C++11$ ניתן לאתחל משתנים (לא סטטיים) ישירות בהצהרה שלהם:

```
class C {
        int x = 7; //class member initializer
        ...
};
```

Constructors delegation

החל מ-$C++11$ בנאי יכול לקרוא לבנאי אחר. במקרים בהם כל בנאי צריך לבצע מספר פעולות בסיסיות (כמו השמה למשתנים) נוכל להגדיר בנאי אחד שעושה את זה בפועל, וכל שאר הבנאים יקראו לו ואחר-כך יבצעו את הפעולות שייחודיות להן:

```
struct Complex {
public:
        Complex(double real, double img) : _real(real), _img(img) {...}
        Complex() : Complex(0,0) {...}
        Complex(double real) : Complex(real, 0) {...}
        Complex(double data[2]): Complex(data[0], data[1]) {...}
private:
        double _real;
        double _img;
};
```

הערות:

- אם בנאי קורא לבנאי אחר (מאותה מחלקה),
  הוא לא יכול לבצע אתחול משתנים דרך ה-*initialize members* שלו.
- ניתן לשרשר מספר בנאים (בנאי אחד קורא לבנאי 2 שקורא לבנאי 3),
  אך זו אחריות שלנו לוודא שאין מעגל.

Operators Overloading

אופרטורים כמו $+,-,*$ הם בעצם מתודות, וניתן לדרוס אותן. למה זה טוב?

נסתכל לדוגמה על מחלקה של מספרים מרוכבים שאנחנו נממש.

המחלקה מכילה 2 משתנים: $real$ ו-$image$. נניח שיש לנו 2 אובייקטים b,c המייצגים מספרים ממשיים, ואובייקט נוסף a שיכיל את תוצאת החיבור שלהם. אם נגדיר במחלקה שלנו את האופרטורים $+,=$, זה יראה כך: $a = b + c$ במקום $a.set(add(b,c))$

וכמובן שהאפשרות הראשונה אינטואיטיבית הרבה יותר.

<u>חוקים</u>:

- נממש אופרטורים רק עם ההתנהגות הצפויה שלהם. למשל לא נממש את $+$ לבצע כפל.
- נחקה את ההתנהגות של אופרטורים ממחלקות אחרות בספרייה הסטנדרטית, למשל:

  $<<,>>$ - משמשים לביצוע פעולות ביטיות על פרימיטיבים,

  אך בספרייה הסטנדרטית משמשים להתנהלות עם סטרימים.

  [ ] – באותו אופן כמו מערכים, המחלקה $vector$ מימשה את האופרטור לאותה מטרה.

למדנו ב-C וגם כאן בהקשר של סטרימים, שניתן לשרשר אופרטורים:

```
int a = 0, b = 2, c = 5;
a = b = c;
cout << b << a << endl;
```

איך זה מתבצע? קודם כל מתבצעת השמה של c ל-b,

השיטה שאחראית על ההשמה מחזירה כערך החזרה את b,

ולאחר מכן ערך ההחזרה מושם ל-a. נרצה לשחזר את ההתנהגות הזו,

ולכן כשנדרוס את האופרטורים נחזיר כערך החזרה את המשתנה השמאלי מבין ה-2.

<u>איך נראית הפונקציה</u>?

$X\&\ operator = (const\ X\&\ rval)$

X הוא שם המחלקה

$X\&$ הוא טיפוס ערך ההחזרה ($reference$ לאובייקט מסוג X)

$operator =$ זה שם הפונקציה למימוש האופרטור "="

והמשתנה $rval$ הוא הפרמטר שנמצא מימין לאופרטור.

שימוש ב-$friend$ כדי לדרוס אופרטורים:

בגרסה הקודמת, השיטה $operator =$ קיבלה רק משתנה אחד, וזה בגלל שהאובייקט השני הוא בעצם ה-$this$ שנשלח על ידי הקומפיילר (כפי שהסברנו בעמודים קודמים)

נראה מימוש נוסף שבו השיטה מקבלת 2 משתנים, ומוגדרת כ-$friend$:

```
//complex.h
class Complex {
private:
        double_re;
        double_im;
public:
        friend Complex& operator+= (Complex& left, const Complex& right);
};
//complex.cpp
Complex& operator+= (Complex& left, const Complex& right) {
        left._re+= right.re;
        left._me+= right.re;
        return left;
}
```

הפונקציה $operator+=$ היא פונקציה גלובלית ולא $member$ של המחלקה (כי היא $friend$),

ולכן הקומפיילר לא מוסיף את $this$ בקריאה שלה.

איזה דרך עדיפה?

כפי שאמרנו, ננסה להמעיט בשימוש של $friend$, ולכן הדרך הראשונה היא עדיפה.

יש מקרים בהם הדרך השנייה הכרחית. למשל במקרה שנרצה לעשות $int + complex$

כלומר שהמשתנה השמאלי הוא לא מאותו סוג של המחלקה שלנו, ואז הקומפיילר לא ישלח מצביע של $this$ לפונקציה. במקרה זה נגדיר את הפונקציה כך:

```
friend Complex operator + (const int left, const Complex& right);
```

דוגמה נוספת לכך הוא אופרטור ההדפסה. את אופרטור ההדפסה נהיה חייבים לממש עם $friend$,

והסיבה היא אותה סיבה, המשתנה השמאלי הוא מסוג $stream$ ולא מסוג המחלקה שלנו.

unary operators:

אופרטורים אונאריים (הפועלים על משתנה בודד) מתחלקים ל-2 חלקים:

$Prefix$ – כאשר האופרטור מגיע לפני האופרנד: $!a$, $-a$, $+a$, $++a$ וכן הלאה.

$Postfix$ – כאשר האופרטור מגיע אחרי האופרנד: $a++, a--$

נראה דוגמה למימוש האופרטורים $a++$ ו-$++a$.

```
Number& operator++(); //prefix++
const Number operator++(int); //postfix++
```

מה שונה מהכרזה של אופרטורים של 2 אופרנדים?

עבור אופרטורים $prefix$, לא נשלח משתנה נוסף.

עבור אופרטורים $postfix$, נוסיף משתנה int סתמי כדי להבדיל בין השיטות (לא נשתמש בו).

```
Number& Number::operator++() {
        _num++; // actual increment takes place here
        return *this;
}
```

במימוש ה-$prefix$ נוסיף 1 לערך ונחזיר רפרנס.

```
const Number Number::operator++(int) { // argument is ignored
        Number tmp = *this; // copy current value
        operator++(); // pre-increment
        return tmp; // return old value
}
```

במימוש ה-$postfix$, נשמור את הערך הנוכחי בצד,

נגדיל את הערך הנוכחי ב-1, ונחזיר את ה-$temp$ ששמרנו בצד.

נשים לב שאנחנו מחזירים משתנה $const$, ולא $const\ reference$ כי tmp הוגדר על ה-$stack$,

לכן נצטרך להחזיר אותו $by\ value$ ולא $by\ reference$.

הערה: בהינתן המימוש שהגדרנו לעיל, השורה השלישית בקוד הבא לא תתקמפל:

```
int i = 10;
int j = 20;
i += 9;
```

הסיבה (מעבר לכך שהמשתנה המוחזר על ידי $i++$ הוא $const$)

היא ש-$i++$ מחזיר משתנה כללי כלשהו, אף משתנה לא תופס אותו והוא "באוויר".

כלומר ההשמה של 9 תתבצע על משתנה לא מוגדר, ולכן זה לא יתקמפל.

Destructors

מפרק ($destructor$) הוא פונקציה מיוחדת המופעלת כאשר זמן החיים של אובייקט נגמר.

המטרה העיקרית של המפרק היא לשחרר את המשאבים שהוקצו על-ידי הבנאי.

למפרק ניתן את שם המחלקה ולפניו נוסיף טילדה. (~)

```
MyClass::MyClass() {
        _mem = (char *) malloc(1000);
}
MyClass::~MyClass() {
        free(_mem);
}
```

Const

ב-CPP לרוב נעביר אובייקטים לפי רפרנס, ולכן החשיבות של $const$ משמעותית יותר מב-C.

אנחנו שולחים את האובייקט עצמו ולא עותק שלו ולכן נרצה לוודא שלא ישנו אותו.

נזכור תמיד לסמן בפרמטרים של שיטה האם הם $const$ או לא.

$C++$ מאפשרת $overloading$ על פונקציות שהוגדרו עם $const$ וכאלה שלא, למשל:

$$void\ A::foo()\ const\{\dots\} \quad vs \quad void\ A::foo()\{\dots\}$$

אבל **לא** כאלה שההבדל היחיד ביניהן הוא שאחד הפרמטרים הוא $const$ והשני לא:

$$foo(int) \quad vs \quad foo(const\ int)$$

נראה דוגמה:

```
class A {
public:
        A(){}
        void foo() const;
        void foo();
};
void A::foo() const { cout << "const foo\n"; }
void A::foo() { cout << "foo\n"; }
```

עבור ה-$main$ הבא:

```
int main() {
        A a;
        const A ca;
        a.foo ();
        ca.foo();
}
```

יודפס:

```
foo
const foo
```

mutable

mutable זו מילה שמורה לשדות של מחלקה שהם **לא** *static* ולא *const*,

המציינת שהשדה יכול להשתנות על ידי פונקציית *const*. (אפילו אם **האובייקט** מוגדר כ-*const*)

נראה דוגמה:

```
class X {
public:
        X() : m_flag(true) {}
        bool getFlag() const {
                m_accessCount++;
                return m_flag;
        }
private:
        bool m_flag;
        mutable int m_accessCount;
};
int main() {
        const X x;
        x.getFlag();
}
```

כלומר הגדרנו את המשתנה *m_accessCount* כ-*mutable*,

ולכן אנחנו יכולים לשנות אותו בתוך השיטה *getFlag*, למרות שהיא מוגדרת כ-*const*.

namespace

namespace הוא מכניזם לאיגוד לוגי של הכרזות ומימושים תחת { }

ניתן לגשת לתוכן ה-*namespace* מבחוץ על ידי *scope resolution operator* (::)

לחילופין, נוכל להשתמש במילה השמורה *using* כדי "לייבא" את ה-*namespace* לקובץ שלנו,

ובכך נוכל לגשת למשתנים ישירות. יצירת *namespace* חדש:

```
namespace smallNamespace {
        int count = 0;
        void abc();
}
```

גישה למשתנה פנימי:

```
smallNamespace::count += 1;
```

אם נוסיף בתחילת הקובץ שלנו את הפקודה:

```
using namespace smallNamespace;
```

נוכל לגשת לתוכן שלו ישירות:

```
count += 1;
abc();
```

הערה: נתקלנו ב-$namespace$ של הספרייה הסטנדרטית std הרבה פעמים,

למשל כדי להשתמש במחרוזת השתמשנו ב-$std :: string$.

יכלנו להוסיף $using\ namespace\ std;$ כדי להימנע משימוש מיותר ב-$std ::$,

אך פעולה זו יכולה לגרום לבעיות בהמשך. נניח שהוספנו את השורות הבאות לתוכנית שלנו:

```
using namespace foo;
using namespace bar;
```

והשתמשנו בשיטה $Blah()$ מ-foo ובשיטה $Quux()$ מ-bar.

לאחר כמה זמן המחלקה foo התעדכנה והוסיפה אליה את השיטה $Quux()$.

כלומר 2 המחלקות מכילות את אותה שיטה ושתיהן יובאו לקוד שלנו, ולכן נצטרך לבצע תיקון בקוד.

אם מראש נשתמש ב-$foo :: Blah()$ ו-$bar :: Quux()$ לא ניתקל בבעיה הזו.

Anonymous namespace:

נוכל להגדיר $namespace$ ללא שם , והוא יהיה זמין רק לקובץ הנוכחי בו הוא הוגדר:

```
namespace {
        int a;
}
```

למה זה טוב? למדנו ששימוש במשתנים גלובליים יכול לגרום להתנגשות עם משתנים אחרים עם אותו שם. עבור הדוגמה הנ"ל, a יהיה נגיש לכל השיטות בקובץ (כמו משתנה גלובלי) אך הוא לא גלובלי ! אם בקובץ אחר יש $namespace$ אנונימי נוסף עם משתנה באותו שם, לא תהיה התנגשות כי ה-$namespace$ הזה נגיש לקובץ הנוכחי בלבד. הגישה למשתנה תהיה ישירה ללא ::, למשל:

```
a = 1;
```

קינון *namespaces*:

נוכל להכריז על $namespace$ בתוך $namespace$ אחר, והגישה תהיה בעזרת " :: " נוסף:

```
int a = 1;
namespace A {
        int a = 2;
        namespace B {
                int a =::a;
        }
}
int main() {
        cout <<::a << endl;
        cout << A::a << endl;
        cout << A::B::a << endl;
}
```

במקרה זה יודפס $1, 2, 1$.

הקצאת זיכרון דינמי

ב-C השתמשנו ב-$malloc/calloc/realloc/free$, אך ב-$C++$ נעבור עם new ו-$delete$.

new – מקצה זיכרון ב-$Heap$, מאתחל אותו על ידי קריאה לבנאי ולבסוף מחזיר מצביע לאובייקט.

$delete$ – קורא ל-$destructor$ ולאחר מכן משחרר את המקום בזיכרון.

לעומת זאת $malloc$ לא יקרא לבנאי (והשדות לא יאותחלו),

ו-$free$ לא יקרא ל-$destructor$ (והשדות לא ישוחררו !)

ניתן להשתמש ב-new עם כל טיפוס:

```
int *i = new int;
char **p = new (char *);
```

<u>הערות</u>:

- new הוא אופרטור גלובלי.
- שימוש ב-new יפעיל את האופרטור new שיקצה מקום בזיכרון ולאחר מכן יפעיל את הבנאי.
- ניתן לדרוס ($overload$) או להחליף את השימוש ב-new.
- כברירת מחדל, שגיאת הקצאה תזרוק $Exception$ במקרה שההקצאה לא צלחה:

```
MyClass *p1 = new MyClass;
```

אם אנחנו לא רוצים שתיזרק חריגה:

```
MyClass *p2 = new (std :: nothrow) MyClass;
```

ובמקרה זה $p2$ יהיה מצביע ל-$nullptr$ אם ההקצאה לא צלחה.

הקבוע $std :: nothrow$ הוא ארגומנט שנשלח לאופרטור new כדי לציין

שאנחנו לא מעוניינים שתיזרק שגיאה, אלא שיוחזר $nullptr$.

- סינטקס:

```
new T; // calls operator new (sizeof(T))
```

קריאה לבנאי הדיפולטיבי (שלא מקבל ארגומנטים)

```
new T(2); // calls operator new (sizeof(T))
```

קריאה לבנאי שמקבל ארגומנט int בודד.

```
new T[5]; // calls operator new[] (sizeof(T) * 5)
```

יצירת מערך של 5 אובייקטים מסוג T על ידי קריאה לבנאי הדיפולטיבי ! (שחייב להיות קיים)

```
new T[n]; // calls operator new[] (sizeof(T) * n)
```

יצירת מערך של n אובייקטים מסוג T גם כן על ידי קריאה לבנאי הדיפולטיבי.

```
new (2, f) T; // calls operator new (sizeof(T) * 2, f)
```

הסינטקס האחרון נקרא $placement\ new$. אפשר לקרוא עליו כאן:

https://isocpp.org/wiki/faq/dtors#placement-new

- אם נרצה לאתחל מערך של אובייקטים עם בנאי שהוא לא הדיפולטיבי,
נקצה מערך של מצביעים לאובייקטים וכך לא תתבצע קריאה לבנאי.
לאחר מכן נאתחל אותם ידנית עם הבנאי הרצוי:

```
int n = 4;
MyClass **arr = new MyClass *[n];
for (int i = 0; i < n; i++) {
        arr[i] = new MyClass (i);
}
```

- כדי למחוק מערכים נשתמש בסינטקס הבא:

```
int *a = new int[10];
int *b = new int[10];
delete[] a;
```

בצורה זו תתבצע קריאה ל-$destructor$ עבור כל אובייקט במערך.
(במקרה זה מדובר ב-int אך עבור מערך של $MyClass$ למשל זה יהיה רלוונטי יותר)
אזהרה: הפעולה $delete\ b$; תגרור התנהגות בלתי רצויה, שתתקמפל אך תגרום ככל הנראה לדליפות זיכרון ובעיות בתוכנה !

- לכל new צריך להיות $delete$! לכל $new[\,]$ צריך להיות $delete[\,]$

auto

החל מ-$C++11$ השימוש של המילה $auto$ השתדרג, וכעת נוכל להשתמש בו כדי לגרום לקומפיילר להתאים את הטיפוס של אובייקט לבד, לפי מידע שהוא כבר יודע. למשל:

```
auto d = 5.0; // 5.0 is a double literal, so d will be type double
auto i = 1 + 2; // 1 + 2 evaluates to an integer, so i will be type int
```

ובנוסף זה יעבוד גם עם ערכי החזרה של פונקציות:

```
auto add(int x, int y) { return x + y; } // x + y is sum of int so auto will be int
int main() {
  auto sum = add(5, 6); // add() returns an int, so sum will be type int
  return 0;
}
```

<u>הערה</u>: לא נוכל להשתמש במילה $auto$ בהכרזה שלא כוללת הגדרה. כלומר $auto\ x$; לא תקין.
לקומפיילר אין דרך לדעת איזה טיפוס אנחנו מכוונים שיהיה המשתנה x.
<u>הערה 2</u>: לא ניתן להגדיר ארגומנטים של פונקציה עם $auto$. (מאותה סיבה)
<u>הערה 3</u>: כדי להגדיר רפרנס עם $auto$, נוסיף & באותו אופן: $auto\ \&vecRef = vec$;

אופרטור =

לכל מחלקה ב-$C++$, הקומפיילר מוסיף מימוש דיפולטיבי לאופרטור =. עבור סדר הפעולות הבא:

$$Complex\ a(1,0);$$
$$Complex\ b(2,0);$$
$$b = a;$$

האופרטור = הדיפולטיבי יעתיק את כל השדות של a בהעתקת $shallow\ copy$ ל-b,

כלומר הוא יעתיק כל שדה של a לשדה המתאים של b בעזרת האופרטור = .

במקרים מסוימים נרצה לשנות את המימוש הדיפולטיבי, נראה דוגמה.

נניח שבנינו בעצמנו מחלקת $String$, וכל אובייקט של המחלקה מחזיק מערך של תווים שהוקצה בעזרת new (כלומר ב-$heap$). עבור סדר הפעולות הבא:

```
MyString str1("Foo");
MyString str2("Bar");
str1 = str2;
```

יתבצע:

```
str1 constructor
str2 constructor
str1.operator = (str2)
str2 destructor
str1 destructor
```

המצביע למערך של המחרוזת "Foo" יידרס על ידי המצביע של "Bar",

ובכך נגרום לדליפת זיכרון (כי נאבד את המצביע למחרוזת "Foo").

בנוסף, 2 פעולות ה-$destructor$ האחרונות ינסו לשחרר את אותו מקום בזיכרון (המחרוזת "Bar").

במקרה זה נרצה לדרוס את הפעולה הדיפולטיבית של =:

```
MyString& MyString::operator = (const MyString &rhs) {
        if(this == &rhs) { return *this; }
        delete [] _string;
        _length = rhs._length;
        _string = new char[_length];
        strncpy(_string, rhs._string, _length);
        return *this;
}
```

<u>מה עשינו?</u>

1. התנאי if הראשון מוודא שהאובייקט שקיבלנו הוא לא $this$, כלומר לא אותו אובייקט, ואז נשחרר לעצמנו בטעות את המחרוזת. למשל:

```
MyString str("Foo");
str = str;
```

2. אחרת, נשחרר את המחרוזת שלנו מהזיכרון ונעתיק את המחרוזת מהאובייקט $\&rhs$.

כעת נראה בעיה נוספת שנגרמת עם בנאי העתקה דיפולטיבי.

בנאי העתקה

בנאי העתקה הוא בנאי המקבל אובייקט מאותו סוג, ומעתיק את הנתונים ממנו לאובייקט החדש.

בנאי העתקה דיפולטיבי (המוגדר על ידי הקומפיילר אם לא הגדרנו כזה בעצמנו),

יעתיק את השדות של האובייקט הנתון בעזרת האופרטור = ,

וכמו שלעיתים נצטרך לדרוס את מימוש האופרטור =, נצטרך לדרוס גם את בנאי ההעתקה:

```
void doNothing( MyString S ) { }
void anotherExample() {
        MyString str("foo");
        doNothing(str);
}
```

בקטע הקוד הנ"ל הגדרנו פונקציה ריקה שלא עושה כלום, ושלחנו אליה את str.

בגלל שלא שלחנו רפרנס, ישלח עותק של המופע על ידי קריאה לבנאי העתקה.

הבעיה בקטע קוד הנ"ל היא שתתבצע העתקה רדודה למצביע של המחרוזת "foo",

וכשהשיטה $doNothing$ תקרוס, ה-$destructor$ של S יופעל,

וישחרר את המחרוזת "foo" – מה שלא רצינו שיקרה.

לכן נצטרך לדרוס את הגדרת בנאי ההעתקה הדיפולטיבית,

ולהעתיק את המחרוזת בעזרת $strncpy$, כלומר לבצע $deep\ copy$.

(במקרה זה יכלנו לשלוח רפרנס, אך יהיו מקרים שנרצה לשלוח עותק).

```
MyString::MyString(const MyString &rhs) {
        _length = rhs._length;
        _string = new char[_length];
        strncpy(_string, rhs._string, _length);
}
```

אם מחלקה צריכה $deep\ copy$ (למשל במקרים של הקצאת זיכרון דינמי):

- נגדיר בעצמנו את $operator =$ ואת בנאי ההעתקה
- נזכור להעתיק את כל המשתנים בצורה חכמה
- נזכור להוסיף בדיקות למקרים של $self - copy$
- נזכור ש-$operator =$ צריך להחזיר רפרנס לאובייקט
- נגדיר $destructor$

<u>חוק השלוש</u>: אם בנינו מחלקה והגענו למסקנה שצריך לממש את אחד מהבאים בעצמנו:

$$destructor, copy\ constructor, operator =$$

ככל הנראה אנחנו צריכים לממש את שלושתם !

STL – Standard Template Library

STL הוא קיצור של Standard Template Library. ספריית ה־STL מכילה אוסף של כלים,

ובהם מימוש למספר רב של מבני נתונים נפוצים, אלגוריתמים, וכלים נוספים.

המחלקה כולה ממומשת באופן גנרי, כלומר מבני הנתונים יכולים להכיל כל סוג של אובייקט,

אך רק סוג אחד באותו מבנה נתונים. נבחר את הסוג הרצוי באתחול, למשל:

$std::vector\langle int\rangle\ intVec;\ //\ vector\ of\ ints$

$std::vector\langle std::string\rangle\ intVec;\ //\ vector\ of\ strings$

$std::vector\langle MyClass\rangle\ intVec;\ //\ vector\ of\ MyClass\ objects$

$std::vector\langle std::vector\langle int\rangle\rangle\ vecOfIntVec;\ //\ vector\ of\ vector\ of\ ints$

<u>טבלת זמני ריצה של מבני הנתונים ב-STL:</u>

$Container$	$Insertion$	$Access$	$Erase$	$Find$
$vector\ /\ string$	$Back: O(1)\ or\ O(n)$	$O(1)$	$Back: O(1)$	$Sorted: O(log\ n)$
	$Other: O(n)$		$Other: O(n)$	$Other: O(n)$
$deque$	$Back/Front: O(1)$	$O(1)$	$Back/Front: O(1)$	$Sorted: O(log\ n)$
	$Other: O(n)$		$Other: O(n)$	$Other: O(n)$
$list\ /\ forward\_list$	$Back/Front: O(1)$	$Back/Front: O(1)$	$Back/Front: O(1)$	$O(n)$
	$With\ iterator: O(1)$	$With\ iterator: O(1)$	$With\ iterator: O(1)$	
	$Index: O(n)$	$Index: O(n)$	$Index: O(n)$	
$set\ /\ map$	$O(log\ n)$	$-$	$O(log\ n)$	$O(log\ n)$
$unordered\_set\ /\ unordered\_map$	$O(1)\ or\ O(n)$	$O(1)\ or\ O(n)$	$O(1)\ or\ O(n)$	$O(1)\ or\ O(n)$
$priority\_queue$	$O(log\ n)$	$O(1)$	$O(log\ n)$	$-$

Sequential Containers – מבני נתונים המאפשרים אחסון וגישה עבור רצף של אובייקטים.

$std::list\langle T\rangle$ – רשימה מקושרת של אובייקטים עם סדר.

- הוספה והסרה בכל מיקום באופן יעיל (התחלה, אמצע וסוף).
- גישה למיקום ספציפי ברשימה הוא ב-$O(n)$

$std::vector\langle T\rangle$ – מערך דינמי. רצף של אובייקטים בגודל משתנה.

- הוספה והסרה **לסוף** הרצף בצורה יעילה. ($O(1)$ בממוצע)
- גישה בזמן קבוע לכל מיקום.

$std::deque\langle T\rangle$ – תור דו כיווני.

- הוספה והסרה לתחילת התור ו/או לסופו ב-$O(1)$.
- גישה בזמן קבוע לכל מיקום.

$std :: vector\langle T\rangle$

שימוש בסיסי בווקטור:

אתחול וקטור ריק	$std::vector\langle int\rangle\ v;$
אתחול וקטור עם n אפסים	$std::vector\langle int\rangle\ v(n);$
אתחול וקטור עם n עותקים של k	$std::vector\langle int\rangle\ v(n,k);$
הוספת k לסוף הווקטור	$v.push\_back(k);$
הסרת האיבר האחרון	$v.pop\_back();$
אתחול/ניקוי המערך	$v.clear();$
גישה לאיבר ה-i בווקטור (עם בדיקת גבולות)	$int\ k\ =\ v.at(i);$
שינוי האיבר ה-i בווקטור (עם בדיקת גבולות)	$v.at(i)\ =\ k;$
גישה לאיבר ה-i בווקטור (ללא בדיקת גבולות)	$int\ k\ =\ v[i];$
שינוי האיבר ה-i בווקטור (ללא בדיקת גבולות)	$v[i]\ =\ k;$
מחזיר אמת אם הווקטור ריק	$v.isEmpty()$
מחזיר את גודל הווקטור	$v.size()$

<u>הערה</u>: למה יש גישה גם עם ( ) וגם עם [ ]? למה לא לבדוק תמיד גבולות?

הגישה ב-$C++$ היא שאם הקוד בנוי באופן תקין, בדיקת גבולות מאטה את הקוד.

<u>בנאים</u>:

```
#include <vector>
int main() {
        // c++11 initializer list syntax:
        std::vector <std::string> words1 {"the",  "frogurt","is","also","cursed"};
        // words2 == words1 (copy constructor)
        std::vector <std::string> words2(words1);
        // words3 is {"Mo","Mo","Mo","Mo","Mo"}
        std::vector <std::string> words3(5,"Mo");
}
```

<u>איך הווקטור עובד</u>?

לווקטור יש 2 משתנים נוספים $size$ ו-$capacity$ שמטרתם לחסוך בקריאות ל-$realloc$.

כלומר לחסוך במספר איתחולי הזיכרון של המערך הדינמי והעתקת הערכים למיקום החדש.

בכל הוספת איבר חדש מתבצעת הבדיקה הבאה: אם $capacity > size$, נוסיף את האיבר לסוף הווקטור. אחרת, נקצה מקום בגודל $2 \cdot size$ בזיכרון ונעתיק את הערכים הקיימים לשם. כך לא נצטרך לבצע העתקה של כל הערכים למיקום חדש בזיכרון בכל הוספה. עבור הוספה של 5 איברים:

$size\ =\ 0\ capacity\ =\ 0$ // Init
$size\ =\ 1\ capacity\ =\ 1$
$size\ =\ 2\ capacity\ =\ 2$
$size\ =\ 3\ capacity\ =\ 4$
$size\ =\ 4\ capacity\ =\ 4$
$size\ =\ 5\ capacity\ =\ 8$

האם ניתן להימנע לגמרי מביצוע $realloc$ים מיותרים?

כן, אם אנחנו יודעים את גודל המערך מראש (אפילו בזמן ריצה) נוכל להשתמש בפונקציה $reserve$:

```
std::vector<int> test;
test.reserve(5);
```

בצורה זו אנחנו מודיעים לווקטור להקצות מראש 5 תאים (או יותר) בזיכרון.

אם הערך שנשלח ל-$reserve$ גדול מה-$capacity$ הנוכחי, יתבצע $realloc$ עם הגודל הרצוי.

בכל שאר המקרים, לא יתבצע שינוי כלל.

ואם נרצה להימנע משימוש בזיכרון מיותר?

החל מ-$C++11$ נוכל להשתמש בפונקציה $shrink\_to\_fit$:

```
std::vector<int> test;
for(unsigned int i = 0; i < 5; i++) {
        test.push_back(i);
}
test.shrink_to_fit(); // return extra memory
```

הפונקציה מבקשת מהווקטור להקטין את ה-$capacity$ לגודל של $size$ (ובכך להחזיר מקומות לא שימושיים בזיכרון) אך לא בטוח שהפעולה תתבצע (מטעמי אופטימיזציה)

יתרונות לשימוש בווקטור:

- מערך דינמי עם שימוש נמוך יחסית של זיכרון.
- מאפשר גישה מהירה לאיברים.
- מנהל הקצאות זיכרון עבורנו.
- מבנה נתונים גנרי ($Template\ container$) המאפשר להכיל כל סוג של אובייקט.

חסרונות לשימוש בווקטור:

- הוספה והסרה של איברים שלא לסוף הווקטור זו פעולה לא יעילה.

הערה: אם גודל המערך ידוע מראש, ניתן להשתמש ב-$std::array$ (החל מ-$C++11$)

```
std::array<int,3> a = {1,2,3};
```

חישוב סכום הווקטור:

נוכל לחשב את סכום האיברים בווקטור על ידי שימוש באלגוריתם המובנה $std::accumulate$:

```
int sum = std::accumulate(v.begin(),v.end(),0);
```

$v.begin()$ ו-$v.end()$ מגדירים את טווח הריצה על הווקטור.

הארגומנט האחרון (0) מגדיר גם את הערך ההתחלתי של הסכימה (במקרה זה ספירה מ-0) וגם את הטיפוס בתוך הווקטור. למשל במקרה זה 0 הוא int ולכן הסכימה תתבצע על איברי הווקטור כ-$ints$, אפילו אם יש שם $floats$.

$std :: deque\langle T\rangle$

תור דו-כיווני, יכול לעשות כל מה שווקטור יכול, ובנוסף גם $push\_front$ ו-pop_front

עבור ווקטור, אם ננסה להוסיף איברים בתחילת המערך אנחנו נצטרך להזיז את כל האיברים שלו תא אחד ימינה. לכן הוספה לתחילת ווקטור היא $O(n)$ שזה מאוד לא יעיל. עבור תור דו-כיווני,

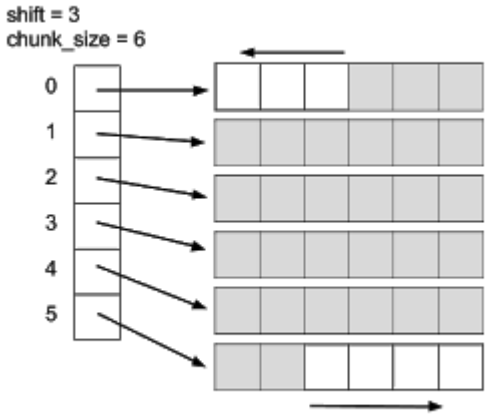
הוספה לתחילת התור יעילה כמעט כמו הוספה לסוף התור. אז למה להשתמש בווקטור אם ככה?

כי בפעולות נפוצות אחרות, כמו למשל גישה לערכים, ווקטור יעבוד מהר יותר.

כמובן שבחירת מבנה הנתונים הנכון הוא תלוי צורך. נבחר בווקטור כמבנה נתונים ברוב המקרים, ונבחר בתור במקרים בהם רוב פעולות ההוספה וההסרה מתבצעות בתחילת / בסוף הרצף.

<u>איך תור דו-כיווני ממומש</u>?

בערך משהו כזה:



כלומר מערך של מערכים ($chunks$) בגודל קבוע. (ובכך נקבל $O(1)$)

בכל קריאה ל-"$push\_back()$", הערך החדש יתווסף ל-$chunk$ האחרון, או שיוקצה $chunk$ חדש שיהפוך להיות האחרון. באותו אופן גם עבור $push\_front()$ הערך יתווסף ל-$chunk$ הראשון, או שיוקצה $chunk$ חדש שיהפוך להיות הראשון.

Container Adapters

הספרייה STL מכילה בנוסף מבנים שהם $container\ adapters$.

כלומר חלק ממבני הנתונים מבוססים על מבני נתונים בסיסיים אחרים.

למשל $queue$ הוא תור חד-כיווני הממומש ב-STL בעזרת $deque$,

ותפקידו להגביל את הפעולות שלו ל-$puch\_back$ ו-pop_front.

בנוסף $stack$ ממומש בעזרת $vector$ המוגבל לפעולות $push\_back$ ו-pop_back בלבד.

ל-$Adaptors$ אין איטרטורים.

Stack

מחסנית הממומשת כתבנית ב-STL , ומקבלת $\langle class\ T, class\ Container = deque\ \langle T\rangle\rangle$.

הפרמטר הראשון מציין את הטיפוס שהמחסנית תכיל, הפרמטר השני יהיה מבנה הנתונים הרצוי, שמממש את הפונקציות $back(),\ push\_back(),\ pop\_back()$. ($deque$ ייבחר כברירת מחדל)

מחסנית מספקת את הפונקציונליות הבאה בלבד:

$empty(),\ size(),\ top(),\ pop(),\ push()$, כאשר בניגוד ל-$Java$, pop היא $void$, כלומר ראש המחסנית לא מוחזר. כדי לגשת לראש המחסנית נשתמש ב-$top()$. אם pop הייתה מחזירה את ראש המחסנית, היא הייתה עושה זאת $by\ value$ ולא $by\ reference$ (כי אז היה מוחזר מצביע למיקום לא נכון), אך החזרה $by\ value$ היא לא יעילה, כי היא תגרור לפחות קריאה מיותרת אחת לבנאי העתקה. לכן לטובת יעילות, pop לא מחזירה כלום, ונוכל לגשת לראש המחסנית עם top.

דוגמה לשימוש במחסנית:

```
int main() {
        stack <int> S;
        S.push(8);
        S.push(7);
        S.push(4);
        assert(S.size() == 3);
        assert(S.top() == 4);
        S.pop();
        assert(S.top() == 7);
        S.pop();
        assert(S.top() == 8);
        S.pop();
        assert(S.empty());
}
```

Associative Containers – מבני נתונים המאפשרים גישה מהירה לאיברים לפי מפתח (key).

במבני נתונים רציפים נוכל לגשת למיקומים מסוימים במבנה רק בעזרת אינדקס מסוג int, אך במבנים אסוציאטיביים נוכל לגשת למיקומים במבנה על ידי טיפוסים נוספים כמו למשל מחרוזות.

<u>המבנים האסוציאטיביים ב-STL הם</u>:

```
std :: map⟨T1, T2⟩
std :: set⟨T⟩
std :: multimap⟨T1, T2⟩
std :: multiset⟨T⟩
std :: unordered_map⟨T1, T2⟩
std :: unordered_set⟨T⟩
```

ה-4 הראשונים הם מבני נתונים **עם סדר** הדורשים שהמפתח יהיו טיפוס שמימש את האופרטור $<$, והם ממומשים בפועל על ידי עץ אדום-שחור. (חיפוש איבר הוא $O(\log n)$)

ה-2 האחרונים הם מבני נתונים **ללא סדר** המבוססים על פונקציית האש. חיפוש איבר יתבצע ב$O(1)$.

במקרה זה המפתחות צריכים להיות אובייקטים שמימשו פונקציית האש.

סיבוכיות המקום של $unordered\ map$ גבוהה מ-map, ולכן הבחירה בהם תלויה בצורך ובמגבלות.

<u>דוגמה לשימוש ב-Map</u>:

```
std::map ⟨const std::string, int⟩ months;
months["january"] = 31;
months["february"] = 28;
months["march"] = 31;
std::string month;
std::cin >> month;
if (months.count(month))
        std::cout << "month -> " << months[month] << std::endl;
else
        std::cout << "not found" << std::endl;
```

האופרטור [ ] משמש לגישה לערכים. הערך המוחזר הוא רפרנס ל-$value$ שמקושר למפתח הנתון.

נעדכן את ה-$value$ בעזרת האופרטור =, ואם המפתח לא קיים, האופרטור יוסיף את המפתח הנתון עם הערך 0. השיטה $count$ סופרת את מספר המופעים של מחרוזת נתונה, במקרה של Map מדובר ב-0 או 1, במקרה של $multimap$ (המאפשר כפילויות של מפתחות) זה יהיה אחרת.

<u>דוגמה לשימוש ב-Set</u>:

set הוא map המקיים $key = value$. נראה דוגמה להוספה הסרה וספירה:

```
std::set ⟨int⟩ s;
s.insert(1); // insert element into the set
s.insert(2); // insert element into the set
std::cout << "Set size = " << s.size() << std::endl;
s.erase(1); // erase element from the set
s.count(2); // count # of '2' in the set
```

איטרטורים

דוגמה חשובה נוספת למבנה נתונים גנרי. מבני נתונים אסוציאטיביים אינם רציפים בזיכרון, איך נוכל לבצע עליהם איטרציה? נשתמש באיטרטורים.

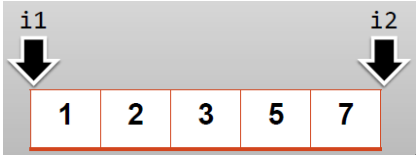
- איטרטורים מאפשרים איטרציה על כל מבנה נתונים, רציף או לא.
- מספקים ממשק סטנדרטי למעבר בלולאה על איברי המבנה.
- השיטות העיקריות שאופרטור צריך לממש הן $operator*$ ו-$operator++$
- יש סוגים שונים של איטרטורים הנבדלים לפי תמיכה בקריאה, כתיבה וגישה.

<u>האיטרטורים $begin()$ ו-$end()$:</u>

$begin()$ – מחזיר איטרטור המצביע לאיבר הראשון ברצף/מבנה.

$end()$ – מחזיר איטרטור המצביע לסוף המבנה (ישר אחרי האיבר האחרון)

```
std::set<int> mySet; // add some integers to mySet
std::set<int>::iterator i1 = mySet.begin();
std::set<int>::iterator i2 = mySet.end();
```



איך ניגש לערך באיטרטור?

איטרטור הוא מצביע לאיבר הראשון לכן נשתמש ב-$*$ ($dereference$) כרגיל:

```
std::cout << *i1 << std::endl;
```

איך נתקדם הלאה עם האיטרטור? נשתמש באופרטור $++$:

```
++i1; // advance i1
```

שימוש באיטרטור בלולאת $while$:

```
while (i1 != i2) {
        std::cout << *i1 << ' ';
        ++i1;
}
```

שימוש באיטרטור בלולאת for:

```
for (std::set<int>::iterator it = mySet.begin(); it != mySet.end(); it++) {
        std::cout << *it << ' ';
}
```

ניתן להחליף את $std::set<int>::iterator$ עם $auto$ ושימוש בלולאת for על הערכים:

```
std::vector<int> v;
int sum = 0;
for (auto n : v) sum += n;
```

איטרטורים ל-Map עובדים בצורה קצת שונה, כי שם יש לנו 2 ערכים.

```
std::map <std::string, int> dictionary;
std::map <std::string, int>::iterator i;
...
i = dictionary.begin();
```

$*i$ הוא $pair$ של $(KeyType, ValueType)$.

האיטרטור של $std::map\langle std::string, int\rangle$ יצביע ל-$std::pair\langle std::string, int\rangle$

גישה לערכים ב-$pair$ היא באמצעות $first$ ו-$second$:

```
std::map <std::string, int> dict;
...
std::map < std::string, int >::iterator i;
for (i = dict.begin(); i != dict.end(); ++i) {
        std::cout << i->first << " " << i->second << std::endl;
}
```

<u>איך נממש איטרטור במחלקה שבנינו?</u>

כדי לממש איטרטור, נגדיר מחלקה מקוננת שתייצג איטרטור, ותחזיק מצביע **פרטי** לאיבר הראשון בקונטיינר, כך שנוכל למנוע שינוי של האיברים מתוך האיטרטור.

בנוסף, נממש במחלקה את השיטות הבאות:

1. $copy\ constructor$
2. $operator = \ (copy)$
3. $operator == \ (compare)\ (and\ !=)$
4. $operator *\ and\ operator->\ (access\ value)$
5. $operator ++\ (increment)$

ואולי (לפי הצורך) גם:

1. $operator[]\ (random\ access)$
2. $operator--\ (decrement)$
3. $operator+= / -=\ (random\ jump)$

במחלקה הראשית, נממש את השיטה $begin()$ שתחזיר מצביע לאיבר הראשון בקונטיינר, ואת השיטה $end()$ שתחזיר מצביע למיקום אחרי האיבר האחרון (כפי שתיארנו בעמוד קודם).

בעמוד הבא נראה מימוש מלא לאיטרטור.

נראה דוגמה למימוש איטרטור (בהנחה שיש לנו מחלקת $Node$):

```
class Stk {
...
        class iterator {
        public:
                iterator(Node *N = nullptr) : _pointer(N) { }
                T& operator * () const {
                        return _pointer-> _value;
                }
                T * operator -> () const {
                        return &_pointer-> _value;
                }
                iterator& operator + +() {
                        _pointer = _pointer-> _next;
                        return *this;
                }
                iterator operator + +(int) {
                        iterator tmp = *this;
                        _pointer = _pointer-> _next;
                        return tmp;
                }
                bool operator == (iterator const& rhs)const {
                        return _pointer == rhs._pointer;
                }
                bool operator! = (iterator const& rhs) const {
                        return _pointer ! = rhs._pointer;
                }
        private:
                Node *_pointer;
        }; // End of iterator inner class

        iterator begin() {
                return iterator(_first);
        }
        iterator end() {
                return iterator(nullptr);
        }
};
```

הערה: השיטה $end()$ מחזירה מצביע ל-$nullptr$ כי ה-$next$ של ה-$Node$ האחרון הוא $nullptr$ ולכן זה עומד בתנאים (מצביע למיקום אחרי האיבר האחרון)

סוגי איטרטורים

כפי שאמרנו, כל איטרטור צריך לממש לפחות את האופרטורים $*$ (דרפרנס) ו- $++$,

ולכן כל איטרטור מאפשר מעבר על האיברים בעזרת $++$. שאר הפעולות תלויות באופן המימוש:

- $\underline{Output}$ – איטרטור המאפשר כתיבה בלבד לתוך האובייקט הנוכחי באיטרציה.

  (למשל האיטרטור של $ostream$)

- $\underline{Input}$ – איטרטור המאפשר קריאה בלבד מתוך האובייקט הנוכחי באיטרציה.

  (למשל האיטרטור של $istream$)

הערה: האיטרטורים הנ"ל $(Output, Input)$ מאפשרים מעבר בודד על מבנה הנתונים, מכיוון שלא מובטח לנו שהאיברים יישארו כמו שהם לאחר המעבר. (יכול להיות שהם ימחקו, ישתנו וכו')

- $\underline{Forward}$ – איטרטור שתומך גם בכתיבה וגם בקריאה.

  (למשל האיטרטור של $unordered\ map, unordered\ set$)

- $\underline{Bi-directional}$ –איטרטור זה מאפשר גם $--$ כלומר אפשר לחזור אחורה.

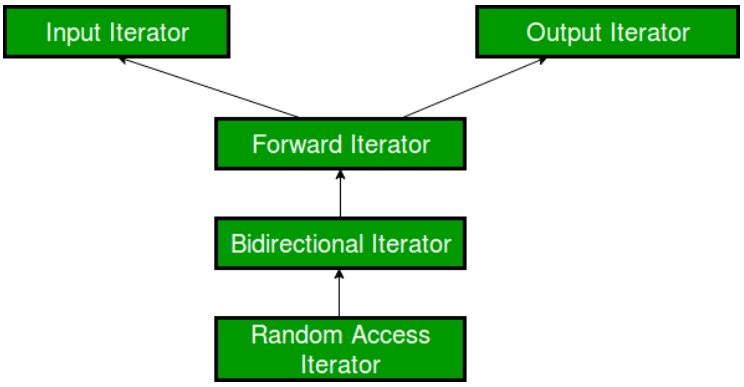
  (למשל האיטרטור של $list, map, set$)

- $\underline{Random}$ – איטרטור זה מאפשר $random\ access$, כלומר מממש את כל האופרטורים שמצביע רגיל מאפשר. מאפשר גם גישה לאיבר מסוים לפי אינדקס !

  (למשל האיטרטור של $vector, array$)

	$Output$	$Input$	$Forward$	$Bi-directional$	$Random$
$Read$		$x = {}^*i$	$x = {}^*i$	$x = {}^*i$	$x = {}^*i$
$Write$	${}^*i = x$		${}^*i = x$	${}^*i = x$	${}^*i = x$
$Iteration$	$++$	$++$	$++$	$++, --$	$++, --, +, -, +=, -=$
$Comparison$		$==,\ !=$	$==,\ !=$	$==,\ !=$	$==, !=, <, >, <=, >=$

הרשימה הבאה מתארת את היחס בין האופרטורים:



(כלומר $Random$ הוא גם כל מה שמעליו וכן הלאה)

Const Iterators

החל מ-$C++11$ ניתן להגדיר גם איטרטורים שמצביעים לאיברים שהם $const$. כלומר, לא ניתן לשנות את האובייקטים במהלך הריצה עליהם. בדומה לשיטות $begin()$ ו-$end()$:

```
iterator begin();
iterator end();
```

עבור איטרטור $const$ נגדיר את השיטות $cbegin()$ ו-$cend()$ כך:

```
class NameOfContainer { ...
typedef ... const_iterator; // iterator type of NameOfContainer
const_iterator cbegin() const; // first element
const_iterator cend() const; // element after last
```

(נשים לב שהן שיטות $const$)

ושימוש בהם יתבצע באופן דומה:

```
NameOfContainer <...> c;
NameOfContainer <...>::const_iterator it;
for( it = c.cbegin(); it! = c.cend(); ++it)
// do something that does not changes *it
```

<u>הערה</u>: עבור אובייקט קונסטי $const\ vector\langle int\rangle$, אם נרוץ עליו באיטרטור, נקבל $const\_iterator$ גם אם נקרא ל-$begin$ ו-end ולא $cbegin$ ו-$cend$, אך גם במקרה זה מומלץ לקרוא להם ישירות לטובת קריאות.

<u>דוגמה לאינקפסולציה של איטרטורים</u>:

נסתכל על השיטות $insert$ ו-$erase$ המוגדרות במבני נתונים רציפים:

```
c.insert(i, x); // insert x before i
c.insert(i, first, last); // insert elements from the range [first, last) before i

c.erase(i); // erase the element that i points to
c.erase(first, last); // erase the elements in the range [first, last)
```

2 השיטות מקבלות איטרטורים, וכל אחת מהן יכולה לקבל אפילו 2 איטרטורים המגדירים טווח.

כלומר $first$ מצביע לאיבר מסוים ו-$last$ מצביע לאיבר כלשהו בהמשך המבנה,

והשיטות יוסיפו או ימחקו את כל הערכים מ-$first$ עד ל-$last$ לא כולל.

הפעולה הזו מתאפשרת בגלל גנריות ואינקפסולציה. לשיטות אין מושג איך מומש האיטרטור,

או איך ממומש המבנה הנתון, ועדיין הפונקציונליות הזאת אפשרית.

Iterators' Validity

בעבודה עם איטרטורים, נזכור תמיד שהולידיות שלהן יכולה להשתנות. למשל עבור קטע הקוד הבא:

```
Container c; Container::iterator i;
for(i = c.begin(); i != c.end(); ++i) {
        if(f(*i)) // f is some test
                c.erase(i);
}
```

תוך כדי ריצה על האיטרטור אנחנו מוחקים איברים במבנה הנתונים.

נראה למה פעולה זו בעייתית, ונפריד ל-2 מקרים.

אם המבנה הנתון הוא וקטור, לאחר מחיקת איבר מסוים האיברים יזוזו תא אחד שמאלה:

בדוגמה זו מחקנו את 2 והאיברים זזו אחד שמאלה כפי שניתן לראות בתמונה הימנית.

לאחר מכן נבצע $++$ ונצביע לאיבר 4, כלומר דילגנו על האיבר 3. זה המקרה הטוב.

במקרה הרע, אנחנו מוחקים מהווקטור איברים ולכן ייתכן שתוך כדי ריצה תתבצע הקצאת זיכרון מחדש, שתתורגם לאחר מכן לגישה לא חוקית לזיכרון. פתרון אפשרי הוא הפעלת האיטרציה כל מחיקה מחדש, או לחילופין לשמור בצד את האינדקסים הרצויים למחיקה ולמחוק בסוף האיטרציה.

אם המבנה הנתון הוא $list, set$ או map, האופרטור $++$ ניגש ל-$next$ של $node$ מסוים במבנה.

אך מכיוון שבמהלך הריצה מחקנו את ה-$node$, גישה ל-$next$ שלו (שדה של $node$) תהיה גישה לא חוקית לזיכרון. כדי לפתור את הבעיה הזו, נוכל להחזיר 2 מצביעים . 1 יצביע ל-$node$ שנרצה למחוק, והשני יצביע ל-$next$ שלו. לאחר מכן נוכל למחוק את $node$ ולהמשיך את האיטרציה מהמצביע השני:

```
Container c;
Container::iterator i = c.begin();
while(i != c.end()) {
        Container::iterator j = i;
        ++i;
        if(f(*j)) { // f is some test
                c.erase(j);
        ...
}
```

אלגוריתמים

בנוסף למבני נתונים ואיטרטורים, ספריית ה-STL מכילה גם אלגוריתמים.

אלגוריתמים הם פונקציות גלובליות שפועלות על איטרטורים, והספרייה ממשיכה לגדול באופן קבוע.
ניתן לראות פירוט מלא כאן: https://en.cppreference.com/w/cpp/algorithm

כל אלגוריתם ידרוש מאיתנו (בתיעוד האלגוריתם, מדובר בחוזה!) את המינימום שהוא צריך.
כלומר, אלגוריתם שזקוק רק לאיטרטור קריאה לא ידרוש איטרטור שממשמש גם כתיבה.
למשל, האלגוריתם $search$, שמחפש איבר מסוים באיטרטור נתון, משתמש ב-== ו-= ! בנוסף לאופרטורים הבסיסיים של איטרטור ($*$ ו-++) ולכן הוא דורש $Forward\ Iterator$.
האלגוריתם $random\_shuffle$ שמבצע ערבוב במבנה נתונים נתון, צריך גישה לאיברים לפי אינדקס ולכן ידרוש $Random\ Iterators$.

כדי להשתמש באלגוריתמים של STL, נוסיף את $\#include\ \langle algorithm \rangle$
חלק מהם מבצעים עיבוד מספרי ולכן כדי להשתמש בהם נוסיף את $\#include\ \langle numeric \rangle$

סוגי אלגוריתמים

$\underline{Non-modifying}$ – אלגוריתמים שלא משנים את הסדר/הערכים של האיברים באיטרטור.
דורשים $input\ /\ forward\ iterators$. פועלים על כל מבני הנתונים הסטנדרטיים של STL.
כוללים בין היתר את:

- $count, count\_if$ – סופרים את האיברים עם ערך מסוים או שעומדים תחת תנאי מסוים.
- $find, find\_if$ – מחפשים אחר האיבר הראשון עם ערך מסוים או תנאי מסוים.

```
#include <algorithm>
std::vector <int> v {1,5,3,2,4};
int n1 = 3;
auto result = std::find(v.begin(), v.end(), n1);
if (result != v.end()) {
        cout << "v contains: " << n1 << endl;
} else {
        cout << "v does not contain: " << n1 << endl;
}
```

- $search$ - מחפש אחר המיקום הראשון של תת-טווח.
- for_each - מבצע פעולה על כל איבר באיטרטור (ולכן יכול **בעקיפין** לשנות את האיברים, תלוי בפעולה הנתונה, למרות שהוא מוגדר כ-$Non-modifying$):

$Function\ for\_each(InputIterator\ first, InputIterator\ last, Function\ f)$

נשים לב שהארגומנט האחרון הוא פונקציה f, שהשימוש בה הינו $f()$.
לכן נוכל לשלוח כארגומנט גם אובייקטים שמממשים את האופרטור $()$ וזה יעבוד.
אובייקטים אלה מדמים פונקציות והם נקראים $Functors$ (נרחיב בהמשך).

הערה: נעדיף תמיד להשתמש בפונקציות המובנות של הקונטיינרים ב-STL מאשר באלגוריתמים. למשל, כדי למצוא איבר ב-set במחלקה שלנו, נעדיף להשתמש בפונקציה $find$ של set מאשר באלגוריתם $find$, מכיוון שהפונקציה שהוגדרה במחלקה של set פועלת תחת הנחות מסוימות שהאלגוריתם הכללי לא יכול לעשות, ולכן היא לרוב תהיה יעילה יותר.

Sorting and related algorithms

אלגוריתמי מיון (רשימה חלקית):

- $sort()$ – ממיין איברים בטווח מסוים.
  ממומש בעזרת $quick-sort$ ופועל בממוצע ב-$O(n \cdot \log n)$. ($O(n^2)$ במקרה הגרוע)
- $stable\_sort()$ – ממיין ושומר על הסדר המקורי של איברים שווים.
  ממומש בעזרת $merge-sort$ ופועל ב-$O(n \cdot \log n)$ אך משתמש בזיכרון נוסף.
- $partial\_sort()$ – ממיין טווח בצורה חלקית. לאחר המיון, $\frac{n}{2}$ האיברים הקטנים ביותר יהיו בחצי הראשון של הטווח, ממוינים מקטן לגדול, כאשר החצי השני יהיו ללא סדר מסוים.
  ממומש בעזרת $heap-sort$ ופועל ב-$O(n \cdot \log n)$.
  לרוב יהיה איטי יותר מ-$sort()$ בפקטור של בין 2 ל-5.

```
#include <algorithm>
std::vector <int> v {1,5,3,2,4};
std::sort(v.begin(),v.end()); // v is now sorted
```

אלגוריתמי חיפוש:

- $binary\_search()$ – חיפוש בינארי.
- $lower\_bound()$ – מחזיר איטרטור המצביע לאיבר הראשון שהוא לא קטן מערך כלשהו.

איחוד:

- $merge()$ – מאחד רשימות ממוינות.

ערימה:

- $make\_heap()$ – מסדר מחדש איברים של מבנה נתונים נתון כך שייצג ערימת מקסימום.
- $sort\_heap()$ – הופך ערימת מקסימום לטווח ממוין.

Conversions & explicit

מוטיבציה: בקטע קוד הבא

```
int foo(int x) {...}
int main() {
        double a = 3.5;
        foo(a);
        return 0;
}
```

אנחנו שולחים משתנה $double$ לשיטה שמקבלת int, וזה תקין.

אומנם אנחנו מאבדים מידע כלשהו בדרך, אך זה מתקמפל ורץ.

לעיתים נרצה לחקות את ההתנהגות הזו גם עבור אובייקטים שאנחנו כותבים.

Implicit conversions – "המרה משתמעת" היא המרה מסוג אחד לשני שאינה דורשת הדפסת טקסט מפורשת. למשל בדוגמה הנ"ל, לא שינינו את a ל-int באופן מפורש. דוגמה נוספת:

```
double res = 15 + 5.785;
```

"15" מזוהה על ידי הקומפיילר כמשתנה מסוג int ו-5.785 מזוהה כ-$double$.

לאחר מכן כדי לבצע את החישוב, הקומפיילר מבצע:

```
double temp = (double)15;
res = temp + 5.785;
```

כלומר ביצענו המרה לא מפורשת, שלאחר מכן הקומפיילר תירגם להמרה מפורשת.

יצירת המשתנה $temp$ תתבצע בעזרת "בנאי המרה" (הסבר בעמוד הבא). למשל:

```
std::string firstString("Convert");
std::string secondString = firstString + "a string"
```

בדוגמה זו הקומפיילר יקרא לבנאי המרה של $std::string$, וייצר מופע עם "$a\ string$":

```
std::string temp("a string");
```

ואז יבצע את חיבור המחרוזות.

בנאי המרה: עד עכשיו השתמשנו בבנאי כדי לייצר מופעים של מחלקה מסוימת.

כעת נראה שימוש נוסף, המאפשר לבצע את מה שתיארנו לעיל:

```
#include <iostream>
class A {
        A(int) {
                std :: cout << "In ctor" << std :: endl;
        }
        A(const A& a) {
                std :: cout << "In copy ctor" << std :: endl;
        }
        A& operator = (const A& a) {
                std :: cout << "In operator = " << std :: endl;
                return *this;
        }
};
```

הגדרנו 2 בנאים (אחד מקבל int והשני בנאי העתקה), ודרסנו את השיטה $operator =$.

נסתכל על קטע הקוד הבא ועל ההדפסות שלו:

```
void f (A a) {
        cout << "In f" << endl;
}
int main() {
        A a1 = 37; // calls the ctor that gets int
        cout << "-------------------" << endl;
        A a2(37); // same
        cout << "-------------------" << endl;
        a1 = 67; // builds a new A and then calls operator =
        cout << "-------------------" << endl;
        f(77); // builds a new A and then send it to f
        cout << "-------------------" << endl;
        return 0;
}
```

```
In ctor
-------------------
In ctor
-------------------
In ctor
In operator=
-------------------
In ctor
In f
-------------------
Press any key to continue . . .
```

2 הפקודות הראשונות הן קריאה לבנאי של A שמקבל int.

(בפעם הראשונה קריאה לא מפורשת, כלומר המרה לא מפורשת, ובפעם השנייה קריאה מפורשת)

הפקודה השלישית היא קריאה לפונקציה $operator =$ שמקבל אובייקט A, אך אנחנו שלחנו מספר.

לכן הקומפיילר יבדוק אם יש בנאי במחלקה A שמקבל int. אם כן, הוא ייצר אובייקט A חדש

עם ה-int הנתון, ואז יכנס לשיטה $operator =$ עם האובייקט הזה. אחרת תהיה שגיאה.

הפקודה הרביעית מבצעת את אותו הדבר עבור השיטה f.

2 הפקודות הנ"ל מבצעות $Implicit\ conversion$, כי לא ציינו במפורש את ההמרה שהתבצעה שם.

explicit

המילה השמורה *explicit* מציינת לקומפיילר שהבנאי שבנינו הוא בנאי רגיל, ולא "בנאי המרה".

למשל במקרה הבא:

```
void foo(const std::vector& v)
...
foo(300);
```

שלחנו את המספר 300 ל-*foo* , ומכיוון שלמחלקה *std :: vector* יש בנאי שמקבל *int*,

ייווצר לנו וקטור באורך 300. אך מה אם זו הייתה טעות? ואין סיבה לממש בנאי המרה כזה?

במקרה כזה נשתמש במילה השמורה *explicit*.

למשל אם נוסיף *explicit* לבנאי הראשון מהדוגמה בעמוד הקודם, כך:

```
explicit A(int) {
        std :: cout << "In ctor" << std :: endl;
}
```

חלק מהאפשרויות שהתקמפלו קודם, כבר לא יתקמפלו:

```
void g() {
        A a1 = 37; // won't compile
        std :: cout << "----------------------" << std :: endl;
        A a2(37); // compile OK
        std :: cout << "----------------------" << std :: endl;
        a1 = 67; // won't compile
        std :: cout << "----------------------" << std :: endl;
        f(77); // won't compile
        std :: cout << "----------------------" << std :: endl;
        f((A)(77)); // compile OK
        std :: cout << "----------------------" << std :: endl;
}
```

קריאות / המרות מפורשות הן תקינות, אך כל קריאה לא מפורשת לבנאי לא תתבצע.

$\underline{user - defined\ conversions}$

עד עכשיו ראינו דרכים להמיר טיפוסים / אובייקטים אחרים לאובייקט מהסוג שלנו,

עכשיו נראה המרה לכיוון ההפוך, מהאובייקט שלנו לטיפוס / אובייקט אחר.

```
class Rational {
public:
        Rational(int x);
        operator int() const;
        const Rational operator + (const Rational& other);
private:
        int _numerator, _denominator;
};

#include "Rational.h"
#include <iostream>
Rational::Rational(int x) {
        std::cout << "Rational from int" << std::endl;
        _numerator = x;
        _denominator = 1;
}
Rational::operator int() const {
        std::cout << "op int (int from Rational)" << std::endl;
        return _numerator/_denominator; // There is data loss
}
```

יצרנו מחלקה בשם $Rational$ עם בנאי שמקבל int ומייצר אובייקט עם מונה x ומכנה 1.

בנוסף דרסנו את האופרטור $int(\,)$, שמשמש לביצוע המרה מפורשת, המחזיר את תוצאת החילוק.

<u>הערה</u>: באותו אופן יכלנו לדרוס את האופרטור $D(\,)$ אם יש לנו בתוכנית מחלקה בשם D.

כעת נוכל לבצע המרה מפורשת של אובייקט מסוג $Rational$ ל-int באופן הבא:

```
#include "Rational.h"
int main() {
Rational x(8); int y = 9;
int r = (int) x + y;
return 0;
}
```

כלומר השתמשנו באופרטור + כדי לחבר 2 int-ים , לאחר שביצענו המרה מפורשת של x ל-int.

מה היה קורה אם לא היינו מבצעים המרה מפורשת? ורושמים רק: $int\ r = x + y;$?

נקבל שגיאת קומפילציה. בעמודים קודמים הסברנו (תחת $overloading$) שבמקרה בו יש 2 פונקציות שמתאימות לקריאה מסוימת, הקומפיילר לא ידע במה לבחור ותהיה שגיאת קומפילציה.

בדוגמה זו גם האופרטור + שממומש ב-int וגם האופרטור + שממומש ב-$Rational$ מתאימים.

Nested Classes

CPP מאפשרת לנו להוסיף מחלקות פנימיות (מקוננות) במחלקות אחרות.

למשל במקרה בו נרצה לממש מחלקת $stack$ שממומשת בעזרת רשימה מקושרת,

נוכל להוסיף מחלקה מקוננת $Node$ בתוך המחלקה $stack$:

```
class IntStack {
private:
        class Node {
        public:
                Node(int& item): pItem(&item), pNext(NULL) {}
                int *pItem;
                Node *pNext;
        };
        Node *pHead;
        ...
};
```

מכיוון שהגדרנו את $Node$ תחת $private$, לא ניתן לייצר מופעים של המחלקה מבחוץ.

כדי להשתמש במחלקה $Node$ מתוך המחלקה $IntStack$ נשתמש בה כרגיל:

```
void IntStack::push(int& item) {
        Node *pNode = new Node(item);
        // no special syntax
        pNode-> pNext = pHead;
        pHead = pNode;
}
```

אם נגדיר את $Node$ במחלקה $IntStack$ תחת $public$, נוכל לגשת אליה מבחוץ כך:

```
class IntStack2 {
        IntStack::Node iWantToo;
};
```

<u>מחלקה מקוננת $Namespaces\ VS$</u>:

- אין הבדל בזמן הריצה.
- נחשוב על $namespace$ כשם משפחתי (חבילה) לטובת זיהוי פונקציות, טיפוסים ומשתנים.
- ניתן להכריז על $namespace$ עם שם של $namespace$ מקובץ אחר (std למשל)
- נשתמש במחלקות פנימיות עבור מידע פרטי אמיתי כמו $Node$ או $Iterator$.
- נשתמש ב-$namespace$ עבור דברים שקשורים לוגית כמו למשל ספריות מתמטיקה.

התנהגות מוזרה של מחלקות מקוננות פרטיות:

```
class enclose {
        struct nested { // private member
                void g() {}
        };
public:
        static nested f() { return nested(); }
};
int main() {
        enclose e = enclose();
        enclose::nested n1 = e.f(); // error: 'nested' is private
        enclose::f().g(); // OK: does not name 'nested'
        auto n2 = enclose::f(); // OK: does not name 'nested'
        n2.g();
}
```

נשים לב ש-$struct$ הוא $member$ של המחלקה ולכן ברירת המחדל היא שהוא $private$.

במקרה זה, כל ניסיון גישה ל-$nested$ מחוץ למחלקה **שכולל את שמו של $nested$** ייכשל, וניסיונות אחרים שלא כוללים את שם המחלקה, יצליחו. לכן המקרה הראשון שבו ציינו באופן מפורש את שם המחלקה ($enclose::nested$) לא מתקמפל, אך כל שאר המקרים כן.

כעת נסתכל על קטע הקוד הבא:

```
struct enclose {
        struct nested {
                static int x;
                void f(int i);
        };
};
int enclose::nested::x = 1; // definition
void enclose::nested::f(int i){...} // definition
```

במקרים בהם נרצה לייצר מחלקה מקוננת בקובץ $header$, שהמימוש שלה ארוך (יותר מכמה שורות), נכריז על הפונקציה ב-$header$ ונממש אותה בקובץ cpp, כמו בדוגמה הנ"ל.

כדי לממש שיטה שהוגדרה בקובץ $header$ עד עכשיו, היינו צריכים למשל $nested :: f$,

אך מכיוון ש-$nested$ נמצאת בתוך $enclose$, נצטרך לרשום $enclose::nested::f$.

באותו אופן, במקרה של משתנה סטטי, נזכור שאפשר להגדיר אותו רק בקובץ cpp,

אז ההכרזה תתבצע בקובץ $header$, והאתחול ב-cpp בעזרת $enclose::nested::x$.

באותו אופן אפשר להגדיר $class$ שלם מחוץ לקובץ ה-$header$, נראה דוגמה:

```
class enclose {
        class nested1; // forward declaration
        class nested2; // forward declaration
        class nested1 {...}; // definition of nested1
};
// definition of nested class nested2
class enclose::nested2 { };
```

<u>הערות</u>:

- מחלקה שבנויה טוב תכיל קובץ $header$ שנכנס למסך אחד או שניים, ומהפונקציות שנמצאות בו נבין את מהות המחלקה.
- נוכל להסתכל על מחלקות מקוננות ב-C באותו אופן כמו מחלקות מקוננות **סטטיות** ב-$Java$. בדוגמה הנ"ל למשל, יצירת אובייקט מסוג $enclose$, **לא** תייצר מופעים של $nested1$ ו-$nested2$. המטרה של מחלקות מקוננות ב-CPP היא הסתרת אופן המימוש (עבור מחלקה מקוננת $private$), וסידור קוד בצורה נוחה למתכנת.
- במידה ואכן נרצה שהמחלקה הכללית תחזיק מופע של המחלקה הפנימית (או להפך), נייצר מופע כזה. (נוסיף שדה של המחלקה הרצויה)
- למחלקות מקוננות **אין** גישה למשתני $private$ של המחלקות החיצוניות, וכנ"ל הפוך.
- חלק מהקומפיילרים יתייחסו למחלקה מקוננת כ-$friend$ של המחלקה החיצונית, אך חלק לא, ולכן לא ניתן לסמוך על זה.
- כברירת מחדל, משתנים של $struct$ הם $public$ ומשתנים של $class$ הם $private$.

<u>שילוב של משתנים גלובליים, משתנים סטטיים ומחלקות מקוננות</u>:

```
int x, y; // globals
class enclose { // enclosing class
        int x; // note: private members
        static int s;
public:
        struct nested { // nested class
                void f(int i) {
                        x = i; // Error: can't write to non
                                        - static enclose::x without instance
                        s = i; // OK: can assign to the static enclose::s
                        ::x = i; // OK: can assign to global x
                        y = i; // OK: can assign to global y
        }
        void g(enclose * p, int i) {
                p->x = i; // OK(?): assign to enclose::x
        }
};
```

inline

כדי לקרוא לפונקציה, הקומפיילר מאחסן את הכתובת הנוכחית בזיכרון, מקצה זיכרון נוסף למשתנים לוקאליים במחסנית, קופץ למיקום אחר בזיכרון ומבצע פעולות נוספות. רצף הפעולות הזה יקר, ובמקרים מסוימים נעדיף להימנע ממנו. ב-C למדנו על פונקציות $inline$ וראינו שמדובר בתחלופה עדיפה על מאקרו ממגוון סיבות (ניתן לראות בסיכום של C).

המילה השמורה $inline$ מציינת בפני הקומפיילר שאולי כדאי להעתיק את התוכן של שיטה מסוימת לכל מקום שקראנו לה, ובכך לחסוך ברצף הפעולות הנ"ל בכל קריאה. הקומפיילר מתייחס לכך כהמלצה בלבד, ובסופו של דבר הוא יבחר את הפעולה הנכונה לפי שיקולי אופטימיזציה.

נשתמש ב-$inline$ (לרוב) עבור פונקציות קצרות באורך שורה או שתיים. איך נעשה זאת?

כל שיטה שנממש בקובץ $header$, הקומפיילר יתייחס אליה כפונקציית $inline$.

לחילופין, נוכל להוסיף את המילה $inline$ בתחילת הפונקציה:

```
inline int square(int x) {
        return x * x;
}
```

אם נרצה שקבצי cpp אחרים יוכלו להשתמש בשיטה , נהיה חייבים להגדיר אותה בקובץ $header$.

<u>האם פונקציות $inline$ תורמות לביצועים</u>?

- שיטות $inline$ עלולות להגדיל את הקובץ
  (למשל אם כל קריאה לפונקציה תוחלף עם תוכן שיטה של מספר שורות)
- שיטות $inline$ יכולות להקטין את הקובץ
  (למשל אם תוכן השיטה קצר יותר מהקריאה שלה)
- שיטות $inline$ יכולות להאט את התוכנית
  (אם הקובץ גדל, למערכת ייקח יותר זמן לקרוא את הקוד מהדיסק)
- שיטות $inline$ יכולות להפוך את התוכנית למהירה יותר
  (כי רצף הפעולות שהסברנו לעיל לא מתקיים)

<u>הערה</u>: קומפיילרים חדשים יכולים לבצע $inline$ גם לשיטות רקורסיביות עד עומק מסוים.

ירושה

הרעיונות המרכזיים של ירושה ב-CPP דומים למה שלמדנו ב-$Java$.

ב-CPP אין $Interface$ אך כן ניתן לבצע ירושות מרובות.

עבור מחלקה בשם $Person$, נגדיר מחלקה בשם $Programmer$ שיורשת מ-$Person$ באופן הבא:

```
#include "Person.h"
class Programmer : public Person {
        ...
};
```

<u>Public, Private, Protected</u>

שדות שאמורים להיות נגישים למחלקות יורשות בלבד, יוגדרו עם המגדיר $protected$. נראה דוגמה:

```
class Person {
public:
        ...
protected:
        ...
};
```

כל מה שנמצא תחת $Protected$ יהיה נגיש למחלקה $Programmer$ ולכל מחלקה יורשת אחרת.

מחלקות אחרות לא יכולות לגשת למה שלא הוגדר תחת $public$.

<u>הערה</u>: נניח שלמחלקה $Programmer$ יש רק את הבנאי הבא:

```
Programmer(std::string name = "", int id = 0, std::string company = "");
```

בגלל שכל הארגומנטים מקבלים ערכים דיפולטיביים, ניתן לקרוא לבנאי בלי ערכים כלל,

ולכן **יש** למחלקה בנאי דיפולטיבי! מימוש הבנאי בקובץ CPP יראה כך:

```
#include "Programmer.h"
Programmer::Programmer(string name, int id, string company)
                : Person(name, id), _company(company) {
        // EMPTY :) Considered elegant
}
```

נשים לב שבקובץ ה-CPP לא רושמים את הערכים הדיפולטיביים שוב.

בדוגמה הנ"ל המשתנים $name$ ו-id מוגדרים במחלקה $Person$ (תחת $Protected$)

והמשתנה $company$ מוגדר במחלקה $Programmer$.

בנוסף, גישה לפונקציות שהוגדרו ב-$Person$ תחת המגדירים $protected/public$

תתבצע באותו אופן כמו גישה לפונקציות שהוגדרו בקובץ $Programmer$.

דריסת פונקציות:

ניתן לדרוס פונקציות שהוגדרו במחלקת האב. למשל אם הפונקציה:

```
void outputDetails(ostream& os) const;
```

הוגדרה במחלקה $Person$, ואנחנו מעוניינים לדרוס אותה כדי להדפיס משהו נוסף,

נוכל להגדיר במחלקה $Programmer$ את אותה הפונקציה:

```
void outputDetails(ostream& os = std::cout) const;
```

במקרה זה הפקודה הבאה:

```
Programmer yoram("Yoram", 1226611, "N.G.C ltd.");
yoram.outputDetails(std::cout);
```

תריץ את הפונקציה שהוגדרה במחלקה $Programmer$ ולא את הפונקציה שהוגדרה ב-$Person$.

ירושה מעגלית

לא ניתן לבצע ירושה מעגלית, מכיוון שכל מחלקה יכולה לרשת רק ממחלקה שהוגדרה מעליה.

המקרה הבא יגרום לשגיאה:

```
class B : public A { ... }; // ERROR invalid use of incomplete type
class A: public B { ... }
```

כלומר המחלקה B לא יכולה לרשת ממחלקה שהוגדרה רק מתחתיה.

גם המקרה הבא יגרום לשגיאה:

```
class B; // forward declaration
class A: public B { ... } // ERROR invalid use of incomplete type
```

בשורה הראשונה הצהרנו על המחלקה B אך לא מימשנו אותה, ולכן לא ניתן לרשת ממנה.

(אחרת נוכל בזמן המימוש להגדיר ש-B ירש מ-A וזה יגרום לירושה מעגלית)

סוגי ירושה

מחלקה יכולה לרשת מחלקה אחרת בעזרת 3 מגדירים:

$class\ Programmer : public\ Person$

במקרה זה, המחלקה $Programmer$ תירש את כל המשתנים והשיטות של $Person$ שהוגדרו תחת $public$ כ-$Public$, ואת כל המשתנים והשיטות שהוגדרו תחת $protected$ כ-$Protected$.

$class\ Programmer : protected\ Person$

במקרה זה, המחלקה $Programmer$ תירש את כל המשתנים והשיטות של $Person$ שהוגדרו תחת $public$ או $protected$, כ-$Protected$.

$class\ Programmer : private\ Person$

במקרה זה, המחלקה $Programmer$ תירש את כל המשתנים והשיטות של $Person$ שהוגדרו תחת $public$ או $protected$, כ-$Private$.

הערה 1: ברירת המחדל של ירושה עבור מבנה ($struct$) הינה $public$,

וברירת המחדל של ירושה עבור מחלקה הינה $private$.

הערה 2: בשלושת האפשרויות הנ"ל, לא ניתן לגשת למשתנים שהוגדרו תחת $private$ ב-$Person$.

גישה למשתני $protected$ של מחלקת האב:

עבור המחלקה A והמחלקה B שיורשת ממנה:

```
class A {
protected:
        int x;
};
class B : public A {
public:
        void foo() {
                cout << x << endl; // OK
        }
        void foo(const B& b) {
                cout << b.x << endl; // OK
        }
        void foo(const A& a) {
                cout << a.x << endl; // Compilation ERROR
        }
};
```

השיטה האחרונה תגרום לשגיאת קומפילציה כי אנחנו מנסים לגשת למשתנה x של אובייקט A כלשהו, ולא למשתנה x שלנו. המשתנה הזה מוגדר כ-$protected$ במחלקה A ולכן לא ניתן לגשת אליו מבחוץ. אמנם אנחנו במחלקה יורשת, אך ניסינו לגשת למשתנה $protected$ שלא ירשנו, אלא של אובייקט אחר.

private inheritance:

עבור המחלקה A הבאה:

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

המחלקה B שיורשת מ-A:

```
class B: private A {
public:
        int x;
        void f() { cout << x << ",” << y << endl; }
};
```

וה-$main$ הבא:

```
int main () {
        A a;
        a.y = 5; // OK
        B b;
        b.x = 2; // OK
        b.y = 5; // ERROR
        b.f();
        A b_as_a = b; // ERROR
}
```

- הסיבה ש-$b.y = 5$ גורם לשגיאה היא שאנחנו ירשנו את A עם המגדיר $private$, ולכן המשתנה y של האובייקט b הוא $private$. (באותו אופן זה היה קורה עם $protected$)
- בשורה האחרונה אנחנו מנסים לבצע $down-casting$ מאובייקט מסוג B לאובייקט חדש מסוג A. השגיאה נגרמת בגלל שהאובייקט b_as_a לא יכול להעתיק את המשתנים שהאובייקט B ירש מ-A, כי הם $private$ כולם (כי B ירש את A עם $private$)

<u>משתנים עם אותו שם</u>:

אם למחלקה A יש משתנה בשם x וגם למחלקה B שירשה אותה יש משתנה בשם x,

הירושה **לא** דורסת את המשתנה של A אלא שומרת את 2 המשתנים.

כדי לגשת למשתנה x של B מתוך המחלקה B, פשוט נשתמש ב-x כרגיל.

כדי לגשת למשתנה x של A מתוך המחלקה B, נשתמש ב-$A :: x$.

מחוץ למחלקה, עבור מופע $B\ b$; נוכל לגשת למשתנה x של A בצורה $b.A :: x$

(בתנאי ש-B ירש את A עם $public$ וגם המשתנה x מוגדר ב-A כ-$public$)

המשתנה x של B מסתיר את המשתנה x של A אך לא דורס אותו.

Order of Construction and Destruction

בקריאה לבנאי של המחלקה B שיורשת מ-A:

- יופעל קודם הבנאי של A
- לאחר מכן יופעלו הבנאים של כל המשתנים של B
- לאחר מכן יופעל הבנאי של B

המפרק ($Destruction$) יפעל בסדר הפוך.

עבור מחלקות A, B, C כך ש-B יורשת מ-A ו-C יורשת מ-B,

קטע הקוד הבא:

```
int main() {
        C obj(1,2,3);
}
```

יגרום לקריאות הבאות לפי הסדר:

```
A ctor
B ctor
C ctor
C dtor
B dtor
A dtor
```

<u>תזכורת לרשימת אתחול</u>:

```
#include <iostream>
class A {
        int a, b;
        public:
        A() : b(a), a(3) {
                std::cout << "a = " << a << ", b = " << b << std::endl;
        }
};
```

מה הערכים של a ו-b? אמנם שמנו את b לפני a ברשימת האתחול,

אך הקומפיילר יאתחל את המשתנים לפי סדר הגדרתם ב-$class$.

בדוגמה הנ"ל הגדרנו $int\ a, b;$ כלומר a הוגדר לפני b ולכן רשימת האתחול תאתחל את a קודם,

ולאחר מכן את b. לכן $a = 3, b = 3$.

במקרה הבא:

```
class Derived : public Base {
    int x; int y;
    Derived ( int x ) :
        Base ( 123 ),// initialize base class
        x ( x ),// x (member) is initialized with x (parameter)
        y ( 0 ) // y initialized to 0
    {} // empty compound statement
    Derived ( double a ) :
        y ( a + 1 ),
        x ( y ) // x will be init before y, its value here is not determined
    {}
```

בבנאי השני אמנם y נמצא לפני x, אך מהסיבה שרשמנו לעיל , x יאותחל קודם עם y.
בנקודה זו y עדיין לא קיבל ערך ולכן x יאותחל עם ערך לא ידוע. בנוסף, בבנאי השני אין קריאה מפורשת לבנאי של $Base$ וזה יתקמפל רק אם ל-$Base$ יש בנאי דיפולטיבי.

הסדר שבו אנחנו כותבים את המשתנים ברשימת האתחול לא רלוונטי,
הקומפיילר יאתחל את המשתנים לפי הסדר בו הם מוגדרים במחלקה.
התהליך כולו מתבצע כך:

1. קריאה לבנאי של מחלקת הבסיס (המחלקה "הגבוהה" ביותר בעץ הירושה) ואתחול המשתנים של מחלקה זו.
2. לאחר מכן נרד "כלפי מטה" בעץ הירושה למחלקות היורשות ונאתחל אותן.
3. אם במהלך הדרך יש מחלקות שיורשות ממספר מחלקות (נראה בהמשך), יתבצע אתחול דומה בסדר DFS (משמאל לימין)
4. לאחר מכן נאתחל משתנים (לא סטטיים) של המחלקה המקורית לפי הסדר בו הם רשומים במחלקה
5. לבסוף ניכנס לגוף הבנאי, מה שמופיע בתוך הסוגריים המסולסלים

קריאה למתודות שירשנו

אמרנו בדוגמה של $Person$ ו-$Programmer$ שבמקרה של 2 שיטות עם אותה חתימה,

קריאה לשיטה דרך מופע של $Programmer$ תפעיל את השיטה שבמחלקה $Programmer$.

מה אם נרצה להפעיל את השיטה שנמצאת ב-$Person$? נוכל לעשות:

```
int main() {
        Person p1("Jane", 1);
        Programmer p2("Joe", 2, "APPL");
        p1.outputDetails(cout);
        Person *p;
        p = &p2;
        p->outputDetails(cout);
        return 0;
}
```

בדוגמה הנ"ל הצבענו על האובייקט $p2$ שהוא מסוג $programmer$, בעזרת מצביע מסוג $Person$.

לאחר מכן קראנו לשיטה $outputDetails$ שמוגדרת גם ב-$Programmer$ וגם ב-$Person$,

והשיטה שהופעלה היא השיטה שנמצאת ב-$Person$. כלומר טיפוס המשתנה הוא זה שקובע איזה שיטה תרוץ, ולא טיפוס האובייקט עצמו. הדוגמה הנ"ל הייתה הקדמה לפולימורפיזם - הנושא הבא.

הערה: באותו אופן כמו משתנים, נוכל לקרוא לשיטה $outputDetails$ של $Person$

מתוך המחלקה $Programmer$ באופן הבא: $Person::outputDetails(os);$

הספרייה הסטנדרטית

הספרייה הסטנדרטית של CPP היא אוסף של פונקציות, קבועים, מחלקות, אובייקטים ותבניות.
היא מרחיבה את השפה ומאפשרת פונקציונליות בסיסית לביצוע מספר משימות כמו למשל מחלקות המאפשרות אינטראקציה עם מערכת ההפעלה, מבני נתונים ואלגוריתמים נפוצים.
ב-C השתמשנו בספריות כמו $stdlib, stdio$. ב-CPP נעדיף להשתמש במחלקות מהספרייה הסטנדרטית של CPP, ונשתמש במחלקות מ-C בלית ברירה. הספרייה הסטנדרטית מכילה ספריות שונות כמו $new, string$ ועוד, וגם את הספרייה $Standart\ Template\ Library$ (STL) המכילה תבניות (נלמד בהמשך) כמו $map, vector$ ועוד. מחלקות שנמצאות בספרייה הסטנדרטית של CPP יהיו תחת ה-$namespace$ הידוע std.

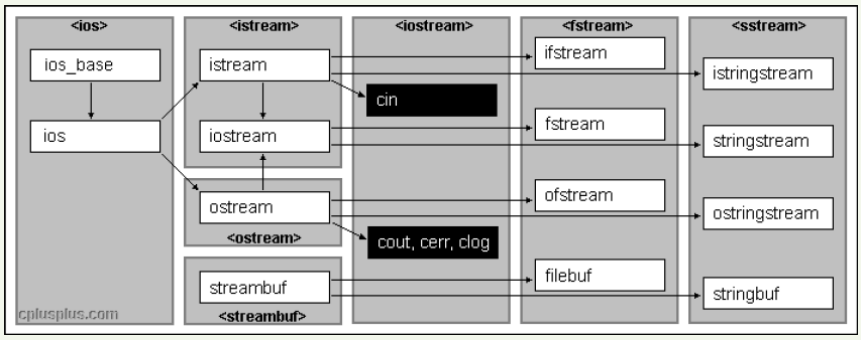
<u>Streams</u>

$Stream$ הוא הפשטה של צינור המעביר מידע מ-$input$ ל-$output$.
בדרך כלל מצד אחד של הצינור יהיה רכיב פיזי, למשל מקלדת, ובצד השני התוכנית שלנו.
בעזרת $stream$ נוכל להעביר מידע מהמקלדת / קובץ וכו' לתוכנית שלנו,
או מהתוכנית שלנו למסך / קובץ וכו'.
$Input\ Stream$ – הזרמה מה-$stream$ לתוכנית שלנו.
$Output\ stream$ – הזרמה מהתוכנית שלנו ל-$stream$.

<u>ספריית $IOstream$</u>

ב-CPP נוכל לגשת לסטרימים הנפוצים בעזרת הספרייה $IOstream$. הספרייה מכילה מחלקות,
אובייקטים כמו $cin, cout$, פונקציות ואופרטורים כמו למשל >> ו-<<,
מניפולטורים כמו $endl, flush, dec, hex, boolalpha$ ועוד. (ניתן להוסיף מניפולטורים משלנו)

עץ הירושה של הספרייה $IOstream$:



כל מלבן אפור הוא אוסף של מחלקות, כל מלבן לבן הוא מחלקה וכל מלבן שחור הוא מופע.

המחלקה $istream$ אחראית על $Input\ Stream$, המחלקה $ostream$ על $Output\ Stream$,

והמחלקה $iostream$ יורשת משתיהן ולכן אנחנו עושים $\#include\ \langle iostream \rangle$

cin הוא מופע של המחלקה $istream$ שנמצא ב-$iostream$,

ו- $cout, cerr, clog$ הם מופעים של $ostream$ שנמצאים גם הם ב-$iostream$.

העמודה השנייה מימין ($fstream$) אחראית על קריאה/כתיבה לקבצים.

$streambuf$ זו מחלקה שמממשת $buffer$ (כמו שלמדנו ב-OOP) ,

וכל מחלקה בעץ הירושה הנ"ל מחזיקה מופע של המחלקה הזו בשביל יעילות.

האופרטור $<<$ מוגדר במחלקה $ostream$ **לכל** טיפוס בסיסי. ($function\ overloading$)

והוא מחזיר רפרנס ל-$output\ stream$ מה שמאפשר שרשור אופרטורים.

באותו אופן האופרטור $>>$ מוגדר במחלקה $istream$ לכל טיפוס בסיסי ומחזיר רפרנס.

עכשיו נוכל להבין את החתימות שלהם בצורה טובה יותר:

$friend\ ostream\&\ operator\ <<\ (ostream\&\ os, const\ Complex\&\ c);$

$friend\ istream\&\ operator\ >>\ (istream\&\ is, Complex\&\ c);$

$clog$-ו $cerr$

אמרנו שלכל מחלקה יש מופע של $streambuf$ לטובת יעילות, כדי לגשת לזיכרון כמה שפחות.

$cerr$ הוא יוצא מן הכלל, הוא אמנם מחזיק מופע כזה, אך בכל הדפסה ל-$cerr$ מתבצעת קריאה לפונקציה $flush$ שתפקידה לרוקן את ה-$buffer$ (כלומר לכתוב למסך באותו רגע).

אם נרצה לדבג את הקוד שלנו, נעדיף להשתמש ב-$cerr$ כדי להדפיס למסך שגיאות ברגע שהן קרו.

ב-$clog$ אנחנו לא נשתמש בקורס אבל נסביר את השימוש שלו בכמה מילים.

$clog$ הוא $stream$ (קובץ) שלרוב נדפיס אליו את מהלך התוכנית שלנו. מה הכוונה?

בכל כניסה לשיטה, בכל יציאה משיטה, בכל קריאה לפונקציה וכו' אנחנו נוסיף הדפסה ל-$clog$ המפרטת את הפעולה, ובהרצת התוכנית קובץ ה-log יכיל מידע מפורט על מהלך התוכנית.

שימושי מאוד לדיבוג במקרים בהם לא נכתוב קוד עם סביבה נוחה כמו $CLion$.

פונקציות של $IOstream$

למחלקות $ostream$ ו-$istream$ יש שיטות נוספות. השיטות מחולקות ל-2 קטגוריות:

$Unformatted\ I/O$: שיטות שקוראות / כותבות ל-$stream$ לפי גודל בתים ולא לפי פורמט. למשל:

```
ostream& write(char const *str, int length);
istream& read(char *str, int length);
```

$Formatted\ I/O$: שיטות שקוראות / כותבות ל-$stream$ לפי פורמט. למשל:

```
ostream& put(char ch);
istream& get(char& ch);
istream& getline(char *buffer, int size, int delimiter = '\n');
```

2 השיטות הראשונות קוראות / כותבות אות בודדת.

השיטה האחרונה קוראת שורה שלמה לפי ה-$delimiter$ הנתון, כשברירת המחדל היא ירידת שורה.

קבצים ב-CPP

כדי לכתוב לקבצים או לקרוא מהם, נוכל להשתמש במחלקות $ifstream, ofstream$ ו-$fstream$:

```
std::ofstream outFile{"out.txt"};
outFile << "Hello world" << endl;
int i;
std::ifstream inFile{"in.txt"};
while(inFile >> i) {
        ...
}
...
```

ב-CPP, קריאה לבנאי של המחלקות הנ"ל תפתח את הקובץ בשבילנו.

למשל השורה הראשונה בדוגמה היא קריאה לבנאי של $ofstream$ עם כתובת לקובץ $out.txt$,

שפותחת עבורנו את הקובץ. בנוסף, במקרה של שגיאה, אין צורך לדאוג לסגור את הקובץ ידנית,

כי קריאה ל-$distructor$ של המופע תבצע סגירה של הקובץ.

הערה: נשים לב שהלולאה רצה כל עוד $inFile >> i$, אך האופרטור $>>$ מחזיר רפרנס ל-$stream$.

אז איך הלולאה יודעת להמשיך לרוץ? המחלקה $ifstream$ מממשת אופרטור המרה $bool()$,

שמאפשר לבצע המרה של $stream$ לערך בוליאני. ערך ההחזרה של האופרטור יהיה $good()$.

$good()$ זו שיטה של המחלקה שבודקת 3 שיטות אחרות:

$bad()$ – תחזיר $true$ אם קרתה שגיאה חמורה כמו למשל שמישהו אחר סגר את הקובץ.

$fail()$ – תחזיר $true$ אם קרתה שגיאה (לא חמורה) כמו למשל ניסיון לקרוא int ללא הצלחה.

$eof()$ – תחזיר $true$ אם הגענו לסוף הקובץ.

ומתקיים: $good() = bad()\ or\ fail()\ or\ eof()$

במידה ונרצה לאתחל את הפרמטרים הנ"ל (לשנות את $bad, fail, eof$ ל-$false$),

נשתמש בשיטה $clear()$.

הפונקציה $clear()$:

```
#include <fstream>
using namespace std;
int main () {
        int num, sum = 0;
        fstream myfile;
        myfile.open("num.txt", ios::in| ios::app);
        while(!myfile.eof()) {
                myfile >> num;
                sum+= num;
        }
        myfile.clear(); // otherwise we get an error
        myfile << sum << endl;
        return 0;
}
```

בדוגמה הנ"ל רצנו על שורות הקובץ $num.txt$ עד לסוף הקובץ.

נניח שאנחנו מעוניינים לכתוב לקובץ לאחר מכן. מה הבעיה?

כשנצא מהלולאה, השיטה $eof()$ תחזיר $true$, ולכן $good()$ תחזיר $false$.

במקרה זה (ש-$good() = false$), לא ניתן לבצע שינויים ב-$stream$ וכל ניסיון כזה יגרום לשגיאה.

לכן נהיה חייבים לקרוא ל-$clear()$ בסוף הלולאה (כפי שעשינו), כדי שנוכל לכתוב לתוך הקובץ.

שימוש בסמן: ב-$Word$ (למשל) יש לנו את הסמן שמסמן איפה יכתבו האותיות הבאות שמגיעות מהמקלדת. נוכל לסמן כל מיקום רצוי במסמך, למשל באמצע שורה, ולכתוב לשם.

את אותה הפעולה נוכל לעשות בעזרת שיטות של $istream$ ו-$ostream$:

$\boldsymbol{Input}$ $(\boldsymbol{get})$: $tellg()$, $\boldsymbol{Output}$ $(\boldsymbol{put})$: $tellp()$

ובאותו אופן לשנות את המיקום שלו בעזרת:

$\boldsymbol{input}$: $seekg(n, location)$, $\boldsymbol{output}$: $seekp(n, location)$

נראה דוגמה:

```
#include <fstream>
using namespace std; //Don't do this at home!
int main () {
        streampos begin, end;
        ifstream myfile{"example.txt"};
        begin = myfile.tellg();
        myfile.seekg(0, ios::end);
        end = myfile.tellg();
        myfile.close();
        cout << "size is: " << (end - begin) << " bytes.\n";
        return 0;
}
```

מתודות וירטואליות ופולימורפיזם

נחזור לבעיית הצורות שלנו מעמוד 9, ונראה איך הכלים הנוספים שלמדנו מאפשרים לנו לייצר פתרון אלגנטי יותר שיהיה עמיד בפני שינויים כמו למשל הוספת צורות חדשות.

<u>פתרון אפשרי ראשון</u>:

נוכל להגדיר מחלקה בשם $Shape$ עם $Enum$ שיכיל את הצורות הרצויות,

ושיטת $Draw$ שתדפיס את הצורה לפי הסוג שלה (ה-$Enum$).

פתרון זה עדיין יצריך שינוי של המחלקה הראשית בכל פעם שנרצה להוסיף מחלקה חדשה.

כלומר המחלקה $Shape$ לא עמידה בפני שינויים.

<u>פתרון אפשרי שני</u>:

נוכל להגדיר לכל צורה מחלקה משלה. למשל מחלקות $Square, Circle$ וכו'.

הוספת צורות חדשות לא דורשת שינוי של המחלקות הקיימות,

אך מצד שני נצטרך לשנות למשל את השיטה שמציירת את כל הצורות,

כדי להוסיף בה תמיכה לצורה החדשה.

<u>פתרון אפשרי שלישי</u>:

נגדיר מחלקת אב $Shape$ ומחלקות יורשות $Square$ ו-$Circle$.

לכל מחלקה תהיה שיטה בשם $Draw$ ושיטה בשם $Area$ (גם ל-$Shape$ עצמה)

```
class Shape {
public:
        void Draw() {cout << 'h';}
        double Area();
};
```

```
class Square: public Shape {
public:
        void Draw() {cout << 'q';}
        double Area();
};
```

```
class Circle: public Shape {
public:
        void Draw() {cout << 'c';}
        double Area();
};
```

בצורה זו נוכל לכתוב קוד גנרי מהצורה:

```
Shape *MyShapes[2];
MyShapes[0] = new Circle();
MyShapes[1] = new Square();
```

מה יודפס אחרי השורה הבאה?

```
MyShapes[0]-> Draw();
```

$'h'$ (כלומר השיטה שהופעלה היא $Shape :: Draw()$)

הסיבה לכך היא שהקומפיילר קורא לשיטה לפי טיפוס האובייקט הנוכחי שלנו. למשל עבור:

```
Circle circle;
circle.Draw();
```

הקומפיילר יקרא ל-$Circle :: Draw()$, אך עבור:

```
Shape *p = new circle();
p-> Draw();
```

הקומפיילר יקרא ל-$Shape :: Draw()$ כי p מטיפוס $Shape$.

<u>הערה 1</u>: אם למחלקה $Circle$ יש שיטה בשם foo , אך במחלקה $Shape$ אין שיטה בשם foo, השורה הבאה תגרום לשגיאת קומפילציה: $p-> foo();$ כי אין שיטה בשם $Shape :: foo()$.

<u>הערה 2</u>: אם למחלקה $Shape$ יש שיטה בשם $moo()$ ול-$Circle$ אין מימוש של השיטה $moo()$, אז השורה הבאה $circle.moo()$ תריץ את $Shape :: moo()$. למה?

תהליך בחירת הפונקציה הנכונה נקרא $static\ resolution$ והוא פועל כך:

- הקומפיילר מתחיל את הבדיקה במחלקה של טיפוס **המשתנה**. בגלל ש-$circle$ הוא אובייקט של $Circle$, הבדיקה תתחיל במחלקה $Circle$. במקרה הראשון, בגלל ש-p מטיפוס $Shape$, הבדיקה תתחיל מהמחלקה $Shape$ **למרות** שהוא מצביע על אובייקט $Circle$.
- לאחר מכן הקומפיילר בודק אם השיטה קיימת במחלקה הזו, ואם כן הוא מפעיל אותה. אחרת הוא עולה "למעלה" בעץ הירושה עד המימוש הראשון של $moo()$ ואז מפעיל אותה.
- אם אין מימוש כזה בכלל , תהיה שגיאת קומפילציה.

<u>נראה דוגמה</u>:

```
Circle *circle = new Circle(1,1,2);
Shape *MyShapes [1];
MyShapes[0] = circle;
circle-> Draw(); // Calls Circle::Draw()
MyShapes[0]-> Draw(); // Calls Shape::Draw()
```

הפתרון השלישי לא מתאים לבעיה שלנו. אנחנו רוצים לאגד את כל האובייקטים תחת רשימה של $Shape$ ולהפעיל את השיטות שנמצאות במחלקות היורשות ולא את השיטות של $Shape$.

בעמודים הבאים נראה איך ניתן לפתור את הבעיה.

Dynamic Resolution in C++:

המילה השמורה *virtual* לפני חתימת שיטה מצהירה שניתן לבצע *overridden* לשיטה הזו:

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

בדוגמה זו, בגלל שהגדרנו את *bar* במחלקה *Base* עם המילה השמורה *virtual*,

רצף השורות הבא:

```
Base *s = new Derived;
s-> bar();
```

יפעיל את *bar()* שנמצאת במחלקה *Derived* ולא את זו שנמצאת ב-*Base*.

הערה 1: ה-*virtual* במחלקה *Derived* לא חיוני. זה היה עובד גם בלעדיו.

הערה 2: אם נרצה לקרוא לשיטה *bar()* של *Base* דרך המחלקה *Derived*, נוכל לעשות זאת עם *Base :: bar()* או מחוץ למחלקה עם *s-> Base :: bar()* (יתקמפל כי השיטה *public*)

פתרון רביעי (ונכון):

אם נגדיר את המחלקות כמו שהראנו בפתרון השלישי **וגם** נוסיף *virtual* בחתימת השיטה של *Shape :: Draw()*, רצף הפקודות הבא:

```
Shape *s  = new Circle;
s-> Draw();
```

ידפיס *'c'* כמו שהיינו רוצים.

מצביעים למחלקות יורשות

```
Derived b;
Base *p  =  &b;
```

p הוא מצביע מטיפוס *Base* שמצביע לטיפוס *Derived*. בכל פעם שנבצע *dereference* (*p*),

הקומפיילר יתייחס לאובייקט ש-*p* מצביע עליו כאובייקט מטיפוס *Base*.

הקומפיילר לא יכול לדעת במקרה הזה מה הטיפוס של האובייקט שיושב בפועל בזיכרון.

רפרנס למחלקות יורשות

במקרה של רפרנס זה עובד בצורה שונה. עבור המבנים הבאים ופונקציית $main$ הבאה:

```
struct B {
        virtual void f() { cout << "B" << endl; }
};
struct D : public B {
        void f() { cout << "D" << endl; }
};
int main() {
        D d1;
        B b1 = d1;
        b1.f(); // prints B
        D d2;
        B& b2 = d2;
        b2.f(); // prints D
}
```

השורות הראשונות ב-$main$ מייצרות משתנה חדש מטיפוס B בעזרת האובייקט $d1$.

בזיכרון מוקצה מקום לאובייקט מטיפוס B, וכל המשתנים שמוגדרים ב-D ולא ב-B לא יועתקו.

השורות הבאות מגדירות רפרנס מטיפוס B לאובייקט מטיפוס D שיורש מ-B.

במקרה זה אמרנו שאפשר לחשוב על רפרנס כמצביע קבוע (לרוב ממומש כך),

ומכיוון שהמצביע קבוע, הקומפיילר כן יכול לדעת מה טיפוס האובייקט ש-$b2$ מצביע עליו בפועל,

ולכן עבור רפרנסים כל תופעל השיטה f שנמצאת במבנה D.

קריאה לפונקציה וירטואלית מתוך פונקציה של $Base$

```
struct B {
        virtual void f() { cout << "Base f()" << endl; }
        void g() { f(); }
};
struct D: public B {
        virtual void f() { cout << "Derv f()" << endl; }
};
int main() {
        D d;
        d.g();
}
```

יצרנו אובייקט מטיפוס D שיורש מ-B, ואז הפעלנו את השיטה $g()$. השיטה g מוגדרת רק ב-B לכן $g()$ :: B תופעל, ובתוכה אנחנו מפעילים את $f()$. אבל איזה $f()$ תופעל? נזכר שהקומפיילר שולח לשיטה את $this$,שהוא מטיפוס D , ולכן $f()$ :: D תופעל ויודפס "$Derv\ f()$". מאותה סיבה, לא נשתמש במתודות וירטואליות בבנאים ומפרקים מכיוון שלמדנו שסדר הפעלת הבנאים היא ממחלקת האב כלפי מטה בעץ הירושה, ולכן הקומפיילר יוסיף לבנאי את $this$ שכרגע הוא מאותו טיפוס של המחלקה בה מוגדר הבנאי, כלומר אין משמעות ל-$virtual$.

תנאים הכרחיים לקיום פולימורפיזם:

- נקרא למתודה דרך מצביע או רפרנס של מחלקת אב ($base\ class$) כלשהי, ובפועל המצביע או הרפרנס מצביעים לאובייקט מסוג מחלקה יורשת ($derived\ object$) .
- המתודה מוגדרת כ-$virtual$.
- המחלקה היורשת חייבת לדרוס את המתודה הרצויה עם אותה חתימה בדיוק.

איך המנגנון של מתודות וירטואליות עובד ב-CPP?

גישה אפשרית: אם $foo()$ היא מתודה וירטואלית , אז לכל אובייקט יש מצביע לשיטה $foo()$ שרלוונטית אליו. (דומה לפתרון שהצגנו עבור בעיית הצורות ב-C)

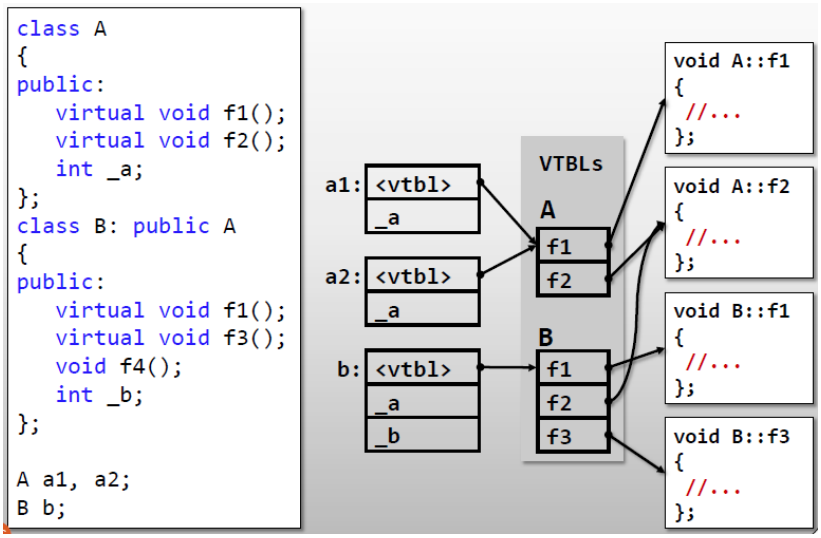
עלות: עבור מספר פונקציות וירטואליות, כל אובייקט צריך להחזיק מספר מצביעים לפונקציות הרלוונטיות , כלומר בזבוז זיכרון.

גישה נוספת: כל אובייקט יחזיק מצביע בודד למערך (טבלה) של מצביעים לפונקציות.

עלות: לכל מחלקה (לא אובייקט) נשמור טבלה בודדת של מצביעים לפונקציות, ולכן במקום מספר מצביעים לפונקציות בכל אובייקט, יש מצביע בודד בכל אובייקט ומערך בודד במחלקה כולה.

ואכן מתודות וירטואליות ב-CPP ממומשות בעזרת טבלת מתודות וירטואליות.

לכל אובייקט יש טבלה עם מצביעים לפונקציות שמוגדרות כ-$virtual$:



מפרק (*destructor*) וירושה

עבור קטע הקוד הבא:

```
class Base {
public:
        ~Base();
};
class Derived : public Base {
public:
        ~Derived();
};
Base *p = new Derived;
delete p;
```

איזה מפרק יופעל אחרי *delete p*? המפרק של *Base*.

היינו רוצים ש-*delete p* יקרא למפרק של *derived*, כי האובייקט שלנו הוא מסוג *derived*.

מפרק הוא כמו כל מתודה אחרת, ולכן נוכל לסמן אותו כ-*virtual*.

<u>כלל אצבע</u>: נסמן את המפרק במחלקה שלנו כ-*virtual* כל עוד יש לנו מתודות וירטואליות במחלקה.

מחלקות אבסטרקטיות

מחלקה אבסטרקטית היא מחלקה שלא ניתן לייצר ממנה מופעים, וניתן להגדיר בה שיטות ללא מימוש (לצד שיטות אחרות). מחלקות אלה נקראות *Pure Virtual Classes*. נראה דוגמה:

```
class Shape {
public:
        virtual ~Shape(); //Virtual destructor
        virtual void Draw() = 0; //Pure virtual
        virtual double Area() = 0; //Also
};
```

השיטה *Draw()* סומנה כ-*pure virtual* בכך שהוספנו לה "= 0".

כל מחלקה שמכילה שיטת *pure virtual* היא מחלקה אבסטרקטית,

כלומר לא ניתן לייצר מופעים שלה, אך כן ניתן לייצר מצביעים ורפרנסים שלה:

```
Shape * p; // legal
Shape s; // illegal
Circle c; // legal
p = &c; // legal
Shape& s = c; // legal
p = new Shape; // illegal
```

Override

בעבודה עם מחלקות יורשות, קל לטעות ולייצר בטעות מתודה וירטואלית חדשה במחלקה היורשת, כשבעצם התכוונו לדרוס מתודה קיימת כזו במחלקת אב. למשל במקרים הבאים:

- כשחתימת השיטה במחלקה היורשת לא מתאימה בדיוק לחתימת השיטה במחלקת אב, למשל עבור משתנה מסוג int במחלקת אב אך מסוג $float$ במחלקת בן.
  או למשל כאשר המתודה במחלקת אב היא $const$ ושכחנו לרשום $const$ במחלקת בן.
- כשאנחנו מוסיפים פרמטר לשיטה במחלקת אב אך שוכחים להוסיף במחלקה היורשת.

קשה לעלות על הטעויות הנ"ל מכיוון שהן לא המחשבה הראשונה שלנו.

לכן נשתמש ב-$override$. שיטה שמסומנת כ-$override$ חייבת לדרוס פונקציה במחלקת אב, ואם היא לא עושה זאת, הקומפיילר יתלונן. כלומר זו דרך נוספת להעביר שגיאות זמן ריצה לשגיאות קומפילציה, לכן נרצה תמיד להוסיף את $override$ למתודות וירטואליות.

```
class Base {
        virtual void A( float = 0.0 );
        virtual void B() const;
        virtual void C();
        void D();
};
class Derived: public Base {
        virtual void C() override; // OK
        virtual void B() override; // compile error
        void D() override; // compile error
};
```

השורה הראשונה תקינה, כי אנחנו דורסים שיטה שקיימת במחלקת אב.

השורה השנייה תגרור שגיאת קומפילציה כי $B()$ במחלקת אב היא $const$ וביורשת לא,

והשורה האחרונה תגרור שגיאת קומפילציה כי $D()$ לא מתודה וירטואלית.

<u>הערה</u>: $override$ היא לא מילה שמורה, היא מקבלת התנהגות מסוימת רק אם היא נמצאת אחרי הארגומנטים של פונקציה, כפי שתיארנו לעיל.

Final

מחלקה שמוגדרת כ-$final$ מציינת בפני הקומפיילר ששום מחלקה לא יכולה לרשת ממנה:

```
// final identifier marks this class as non – inheritable
class Base final { };
// trying to override final class Base will cause a compiler error
class Derived: public Base { };
```

כלומר ניסינו לרשת ממחלקה שמוגדרת עם $final$ ולכן תהיה שגיאת קומפילציה.

נוכל באותו אופן למנוע דריסה של מתודה וירטואלית:

```
class Base {
        // final identifier marks this function as non – overrideable
        virtual void A() final;
};
class Derived: public Base {
        // trying to override final function
        // Base::A() will cause a compiler error
        virtual void A();
};
```

כלומר ניסינו לדרוס שיטה שמוגדרת עם $final$ ולכן תהיה שגיאת קומפילציה.

<u>הערה 1</u>: יש מספר סיבות לגיטימיות להשתמש ב-$final$, אך באופן כללי נשתדל להימנע מכך. במידה ואכן החלטנו להגדיר מתודה כ-$final$, נתעד זאת ונסביר למה כי ככל הנראה הסיבה לא תהיה ברורה למי שיורש מהמחלקה שלנו.

<u>הערה 2</u>: גם $final$ היא לא מילה שמורה, היא מקבלת התנהגות מסוימת רק אם היא נמצאת אחרי הארגומנטים של פונקציה, או בראש המחלקה כפי שתיארנו לעיל.

העתקה והמרה

באופן כללי, כשאנחנו רואים את הביטוי $foo(x)$, הקומפיילר מחפש אחר פונקציה foo עם ארגומנט שמתאים ל-x **או** פונקציה foo עם ארגומנט מטיפוס שאפשר להמיר את x אליו.

ראינו המרות סטנדרטיות: $int \to double, \quad char \to int$ וכו'.

וראינו המרות בעזרת בנאי המרה (המרות שהמתכנת בחר לממש)

נראה כעת המרת $Up - Casting$ (המרה ממחלקה יורשת למחלקת אב):

```
g(Base b) { ... }
f(Base *b) { ... }
h(Base& b) { ... }
Derived *d = ...;
g(*d);
f(d);
h(*d);
```

כולם חוקיים. באותו אופן נוכל לבצע העתקה ממחלקה יורשת:

```
class Base {
private:
        double _x;
        int _a;
};
class Derived : public Base {
private:
        double _z;
};
```

עבור הקוד הבא:

```
Base a;
Derived b;
...
a = b;
```

תתבצע העתקה **רק** של המשתנים $\_a$ ו-$\_x$ של b (שהוא מטיפוס $Derived$).

למה זה עובד? $Overload\ resolution$. הפעולה $a =$ היא קריאה לאופרטור:

```
Base& operator = (const Base& other);
```

וכפי שהסברנו לעיל, יתבצע $Up - Casting$ מהאובייקט b לאובייקט $Base$.

חיפוש הפונקציה הנכונה ($Overload\ resolution$) מתבצע לפי הסדר הבא:

- פונקציה שמקבלת המרה ממשתנה לא $const$ למשתנה $const$
- המרה ממשתנה לא $reference$ למשתנה $reference$
- $Up - Casting\ (Conversion)$

הערה: מכיוון שבוצע $Up-Casting$, השיטה $operator=$ לא יודעת שהאובייקט המקורי מסוג $Derived$, ולכן לא תתבצע העתקה של טבלת המתודות הווירטואליות של $Derived$.
כלומר, a יעתיק אליו את השדות של b, ולאחר מכן הוא יהיה אובייקט A לכל דבר,
עם מצביע לטבלת המתודות הווירטואליות של A, ולכן לא יתבצע פולימורפיזם לאחר מכן.

באג נפוץ: עבור המחלקות הבאות:

```
class A {
public:
        virtual void f1();
        virtual void f2();
        int _a;
};
class B: public A {
public:
        virtual void f1();
        virtual void f3();
        void f4();
        int _b;
};
```

השיטות:

```
void thatWhatIWant(A& a) { a.f1(); //polymorphic };
void iForgotTheRef(A a) { a.f1(); // non polymorphic };
```

וה-$main$ הבא:

```
A a;
B b;
iForgotTheRef(b); //will call A::f1()
```

הקריאה ל-$iForgotTheRef$ מבצעת העתקה של האובייקט b, כי היא מקבלת $A\ a$ ולא רפרנס,
ולכן תתבצע העתקה המרה, שמייצרת אובייקט A חדש ומההערה לעיל $a.f1()$ יפעיל את $f1$ של המחלקה A ולא של B כמו שהיינו רוצים שיקרה. כלומר – לא לשכוח רפרנס כשצריך !

הערה: עבור אובייקט b של מחלקה B שיורשת ממחלקה A שלה יש משתנה בשם $\_a$,
נזכור שאם אין בבנאי העתקה של B קריאה מפורשת לבנאי העתקה הנכון ב-A,
תהיה קריאה לבנאי הדיפולטיבי (אם אכן יש כזה, אחרת תהיה שגיאת קומפילציה).
וקריאה לבנאי הדיפולטיבי של A לא תעתיק את $\_a$ של b אלא תאתחל אותו לערך דיפולטיבי.
לכן נזכור תמיד להוסיף קריאה מפורשת לבנאי הרצוי של מחלקת האב בבנאים שלנו.

Functors

$Function\ Object\ (Functors)$ הם כל דבר שניתן "לקרוא" לו, באותו אופן כמו פונקציה. למשל:

מצביע לפונקציה, מחלקה שמממשת את $operator()$ וכו'. דוגמה:

```
class c_str_less {
public:
        bool operator()(const char *s1, const char *s2) const {
                return (strcmp(s1, s2) < 0);
        }
};
c_str_less cmp; // declare an object
cmp("aa", "ab");
...
c_str_less()("a", "b");
```

כלומר יצרנו מחלקה בשם c_str_less שמממשת את $operator()$,

ולכן יכלנו "להפעיל" את האובייקט cmp על המחרוזות "aa" ו-"ab".

השורה האחרונה בדוגמה הנ"ל מייצרת משתנה סתמי על ידי קריאה לבנאי הדיפולטיבי,

ולאחר מכן "מפעילה" אותו על "a" ו-"b".

<u>*Template Comparator*</u>

```
template < typename T >
class less {
public:
        bool operator()(const T& lhs, const T& rhs) const {
                return lhs < rhs;
        }
};
```

המחלקה $less$ יכולה להיות מופעלת באופן הבא:

```
less <int> cmp; // declare an object
cmp(1,2)
...
less <int>()(1,2)
```

ובכך היא יכולה לייצג $comparator$ (אובייקט / שיטה שמשווה בין 2 אובייקטים)

באותו אופן יכלנו לייצר מחלקת $greater$ שמשווה בעזרת >, ולבחור אם לשלוח את $less$ או את $greater$ לשיטות שמקבלות $comparator$ כמו למשל שיטות מיון.

שימוש ב-$Functor$ לעומת מצביע לפונקציה מאפשר לנו להשתמש ביתרונות של מחלקות:

- ירושה
- אפשרות לשנות התנהגות בזמן ריצה ולא רק בזמן קומפילציה
- סכימה של נתונים (למשתנה של המחלקה למשל) במהלך הקריאות
- אפשרות למספר "גרסאות" של הפונקציה (על ידי מופעים שונים)

<u>הערה</u>: לכל הסיבות הנ"ל יש גם חלופות שיתאימו לשימוש עם שיטות רגילות, אך לרוב הן יהיו מסורבלות יותר. בנוסף, יותר קל לבצע $inline$ לקריאות של $functors$ לעומת קריאות לפונקציות רגילות, ולכן הן נחשבות ליעילות יותר.

<u>$Functors$ ומצביעים לפונקציות</u>

- פונקציה שמקבלת מצביע לפונקציה, לא יכולה לקבל $functor$ במקום – ולהפך.
- אם נרצה לאפשר גם מצביע לפונקציה וגם $functor$, נצטרך להשתמש ב-$templates$. מכיוון ששניהם מופעלים בעזרת ( ), הקוד יתקמפל.

<u>דוגמה לשינוי התנהגות בזמן ריצה</u>

האלגוריתם $count\_if$ של STL מקבל איטרטור התחלה, איטרטור סוף, ופונקציה שמקבלת משתנה בודד ומחזירה $bool$. הפונקציה תספור את כל האובייקטים באיטרטור שהפונקציה מחזירה עבורם $true$ (כלומר שעומדים בתנאי הנתון). נניח שנרצה לספור את מספר המחרוזות באיטרטור מסוים, שהאורך שלהן הוא לפחות 5. נרצה להשתמש בשיטה שמקבלת 2 ארגומנטים – מחרוזת ואורך רצוי, ולשלוח אותה ל-$count\_if$. אך במקרה זה תהיה שגיאת קומפילציה מכיוון ש-$count\_if$ מפעילה את הפונקציה הנתונה עם ארגומנט בודד. כדי לפתור את הבעיה , נוכל להשתמש ב-$Functor$.

נבנה מחלקה / מבנה עם בנאי שמקבל ארגומנט בודד $length$, ונממש בו את $operator( )$:

```
struct ShorterThan {
        ShorterThan(int maxLength): _len(maxLength) { }
        bool operator( )(const std::string &str) const {
                return str.length() < _len;
        }
private:
        int _len;
};
```

לאחר מכן נוכל לייצר מופע עם האורך הרצוי:

```
ShorterThan st(length);
```

ולשלוח אותו ל-$count\_if$:

```
count_if(vec.begin(), vec.end(), st);
```

בצורה זו נקבל את התוצאה הרצויה, ובנוסף נוכל לבחור איזה אורך שבא לנו בזמן ריצה, נוכל למשל לאתחל את המופע לפי קלט מהמשתמש.

פונקציות lambda

פונקציות למדה הן פונקציות ללא שם שיכולות לקבל כארגומנטים משתנים שנמצאים באותו $Scope$ בו הן הוגדרו. הסינטקס שלהן הינו: [ ] ( ) –> { } . דוגמה:

```
int main() {
        int a = 3;
        auto f = [a](int x, int y)–> int {
                return (x + y) · a;
        };
        int b = f(5,2); // b = 21
}
```

[ ] (Capture Closure) - בסוגריים המרובעים נבחר איזה משתנים מה-$Scope$ הנוכחי נרצה שהפונקציה תכיר במימוש שלה (a בדוגמה הנ"ל)

נוכל לשלוח לפי רפרנס או לפי ערך. דוגמאות שימוש:

[=] – ישלח את כל המשתנים שנמצאים ב-$Scope$ הנוכחי $by\ value$.

[&] – ישלח את כל המשתנים שנמצאים ב-$Scope$ הנוכחי $by\ reference$

$[x, &y]$ – ישלח עותק ($by\ value$) של x , ורפרנס של y

$[=, &y]$ – ישלח עותק כל המשתנים מה-$Scope$ הנוכחי **למעט** y, ובנוסף רפרנס ל-y.

הערה: משתנים שנשלחו לשיטה $by\ value$, ישלחו כ-$const$, כלומר לא ניתן לשנות אותם בפונקציה:

```
auto f = [a](int x, int y)–> int {
        a = 4; // ERROR: assignment to read – only a
        return (x + y) · a;
};
```

אם בכל זאת נרצה לבצע בהם שינוי, נשמש במילה השמורה $mutable$ שלמדנו, כך:

```
auto f = [a](int x, int y) mutable –> int {
        a = 4; // OK
        return (x + y) · a;
};
```

(כמובן שהשינוי יהיה לוקאלי כי a הוא עותק)

( ) (Function Argument List) - בסוגריים העגולים נגדיר את הארגומנטים של השיטה.

בדוגמה הנ"ל בחרנו לשלוח 2 משתנים מטיפוס int.

–> (Return Type) – טיפוס ערך ההחזרה. בדוגמה הנ"ל ערך ההחזרה הוא int.

פונקציית הלמדה בדוגמה הנ"ל מקבלת 2 מספרים שלמים ומחזירה את תוצאה החיבור שלהם,

כלומר הקומפיילר יכול לנחש את ערך ההחזרה לבד ולכן נוכל להגדיר את הפונקציה גם כך:

```
auto f = [a](int x, int y) {
```

{ } – גוף הפונקציה.

ניתן לשלוח פונקציות למדה בצורה ישירה:

```
int main() {
        array <int, 5> a = {1,2,3,4,5};
        int sum = 0;
        for_each(a.begin(), a.end(), [&sum](int i){sum+= i;} ); // 15
}
```

ניתן לבצע קינון של פונקציות למדה:

```
int main() {
  int a = 1, b = 1;
  auto m1 = [a]() {
    auto m2 = [b]() {
    // Error: b not captured
      std::cout << b;
    };
    m2();
    std::cout << a;
  };
  m1();
}
```

```
int main() {
  int a = 1, b = 1;
  auto m1 = [a, b]() {
    auto m2 = [b]() {
    // OK
      std::cout << b;
    };
    m2();
    std::cout << a;
  };
  m1();
}
```

כדי שהפונקציה הפנימית תכיר את b, נצטרך לשלוח את b לפונקציה החיצונית.

(כי הפונקציה הפנימית נמצאת ב-$Scope$ משלה)

Template Functions

בדומה ל-$Generics$ ב-$Java$, תבנית יכולה לקבל סוגים שונים של טיפוסים, עבור אותו קוד בדיוק.
למשל עבור שיטת מיון $sort()$, נוכל לכתוב שיטה שמקבלת מערך גנרי, וכל עוד האובייקטים במערך מאפשרים לבדוק יחס בין 2 אובייקטים למשל על ידי מימוש האופרטורים $<$, $>$, המערך יעבור מיון.
כדי לאפשר שימוש בתבניות, נוספו לשפה 2 מילים שמורות: $template$ ו-$typename$
(כאשר את השני ניתן תמיד להחליף במילה השמורה $class$)
בשונה מ-$Java$ (שבה ניתן לדרוש שאובייקט יממש ממשק מסוים), ב-$C++$ אין אפשרות לדעת אם המחלקה של אובייקט נתון עומדת בדרישות ולכן יכולות להיווצר בעיות. רוב השגיאות בשימוש עם תבניות קורות בגלל חוסר הבנה בין דרישות התבנית למי שמשתמש בה. חשוב להבין שדרישות התבנית הן **חוזה** בלבד, ולכן לפני כל שימוש בתבנית מסוימת, נקרא את החוזה שלה. באותו אופן, אם נבנה תבנית בעצמנו, נצטרך להגדיר חוזה באותו אופן. (כלומר להוסיף דוקומנטציה ברורה)
<u>דוגמה</u>:

```
template <typename T> const T& min(const T& x, const T& y) {
        return x < y ? x : y;
}
```

השיטה הנ"ל מחזירה את האובייקט הקטן מבין 2 אובייקטים, מבלי לדעת מה הטיפוס שלו.
היא מניחה שהמחלקה של האובייקטים הנתונים מימשה את האופרטור $<$,
שמחזיר $boolean$ ומאפשר לבדוק יחס, ולכן ההנחה הזו חייבת להיות מתועדת היטב בתיאור.
נשים לב ש-T הוא $placeholder$ "שמשתנה" לטיפוס הנכון בזמן ריצה,
והשם שלו (T) הוא לא ייחודי, כלומר הצהרת הפונקציה הבאה בנוסף לקודמת:

```
template <typename P> const P& min(const P& x, const P& y) {
        return x < y ? x : y;
}
```

תגרום לשגיאת קומפילציה מכיוון שמדובר באותה שיטה. (אין חשיבות לשם)

<u>$Concepts\ and\ STL$</u>:

- $concept$ זו רשימת דרישות מטיפוס מסוים.
- STL מגדירה היררכיה של $concepts$ עבור מבני נתונים, איטרטורים וטיפוסים.
- $Concepts$ של טיפוסים ב-STL:
  - $\boldsymbol{Equality\ Comparable}$ – טיפוסים שמימשו את $operator\ ==$
  - $\boldsymbol{LessThan\ Comparable}$ – טיפוסים שמימשו את $operator\ <$
  - $\boldsymbol{Assignable}$ – טיפוסים שמימשו את $operator\ =$ ובנאי העתקה.
- טיפוסים פרימיטיביים גם יכולים להתאם ל-$concept$ מסוים.

Template Functions:

המילה השמורה *templates* מגדירה קטע קוד שיאותחל בכל פעם עם ארגומנטים מסוגים שונים.

```
template ⟨typename T⟩ // T is a "type argument"
void swap( T& a, T& b ) {
        T tmp = a;
        a = b;
        b = tmp;
}
```

כלומר השורה הראשונה מגדירה לקומפיילר ש-T הוא שם כללי של מחלקה.

השיטה הנ"ל מניחה שלאובייקטים יש בנאי העתקה (שורה ראשונה) ואופרטור = (שורה שניה).

כפי שכבר אמרנו, יכלנו באותה מידה לרשום ; *template* ⟨*class T*⟩ . (*class* במקום *typename*)

עבור קטע הקוד הבא:

```
int main() {
        int a = 2;
        int b = 3;
        swap( a, b ); // requires swap(int&, int&)
}
```

הקומפיילר יבדוק אחר שיטה $swap(int\&, int\&)$. אם אין כזאת, הוא יחפש $swap(T\&, T\&)$,

ואם אכן קיימת כזאת, הוא ייצר עותק של השיטה עם *int* במקום T ויקמפל אותה.

העותק של השיטה ייווצר פעם אחת בלבד, גם עבור אותן קריאות בהמשך,

ואפילו עבור קריאה דומה במחלקה אחרת לגמרי (שהיא חלק מהתוכנית שלנו כמובן).

נוכל לקרוא לשיטה $swap(a, b)$ עם *int* גם באופן מפורש: $swap\langle int\rangle(a, b)$.

נוכל לקרוא לשיטה באופן מפורש למשל במקרים בהם יש התנגשות בין 2 שיטות מתאימות.

באותו אופן, נוכל להצהיר על פונקציות עם טיפוסים ידועים מראש באופן הבא:

```
template void f⟨int⟩(int);
template void f(char);
```

כלומר אנחנו אומרים לקומפיילר להכין עותקים של השיטות האלה מראש.

אתחול הפונקציות מראש יכול לשפר את זמן הקומפילציה של התוכנית שלנו.

קריאה לפונקציה *swap* פעם אחת עם *ints* ופעם אחת עם *doubles* יגרום ליצירת 2 עותקים שונים.

(אפילו שיכולה להתבצע המרה לא מפורשת) . שימוש בעותקים רבים של שיטות גנריות יכול להגדיל

את הקוד שלנו, אך זה נחשב לקוד נכון יותר כי כתבנו את השיטה *swap* פעם אחת בלבד.

delete ו-*default*:

המילים השמורות הנ"ל מאפשרות לנו לשלוט בשיטות/הבנאים שממומשים עבורנו על ידי הקומפיילר:

```
class Cookie {
public:
        Cookie() = default;
        Cookie(const Cookie&) = delete;
        Cookie& operator = (const Cookie&) = delete;
};
```

השימוש ב-*defalt* סימן לקומפיילר לממש עבורנו את הבנאי הדיפולטיבי.

שימושי במקרה בו נגדיר בנאי כלשהו, ונרצה שהקומפיילר יממש עבורנו גם את הדיפולטיבי.

השימוש ב-*delete* עבור בנאי ההעתקה ואופרטור = מסמן לקומפיילר **לא** לממש עבורנו את השיטות הדיפולטיביות, ובכך אנחנו מונעים את האפשרות להעתיק אובייקטים מהמחלקה שלנו.

(במידה ומישהו ינסה, תהיה שגיאת **קומפילציה**)

הערה 1: פונקציית *template* היא רק הכרזה. היא לא יכולה להתקמפל כמו שהיא אלא היא צריכה לעבור את תהליך השינוי מ-T לטיפוס הרצוי. לכן מימוש השיטה חייב להיעשות באותו קובץ בו הגדרנו את התבנית. (כדי שהקומפיילר ידע מי זה T). רק לאחר השינוי הקומפיילר יבדוק שגיאות סינטקס.

הערה 2: אמרנו שפונקציות *template* יגדרו ב-*header*, אך להבדיל מפונקציות רגילות שמוגדרות שם, פונקציות *template* לא מסומנות לבד כ-*inline*. אם נרצה לסמן אותן כ-*inline* נעשה זאת בצורה מפורשת.

הערה חשובה 1: ראינו ב-*Quiz* 2 ששיטות *inline* ושיטות סטטיות הן ייחודיות לקובץ בו הן הוגדרו, ולכן בתהליך ה-*Linkage* לא תהיה שגיאה במקרה של 2 פונקציות דומות. גם עבור מספר פונקציות *template* עם אותו שם לא תהיה שגיאת מקשר, אך הפונקציות **לא** ייחודיות לקובץ בו הן הוגדרו. בסוף תהליך הקישור יישמר רק עותק אחד של מימוש הפונקציה ! איזה מימוש ייבחר? מה שהלינקר יחליט, אין התנהגות מוגדרת. לכן אם יש לנו מספר פונקציות *template* עם אותו שם, זה יתקמפל וככל הנראה תהיה התנהגות בלתי צפויה בתוכנית שלנו.

הערה חשובה 2: מהסיבה הנ"ל, נהיה חייבים להגדיר פונקציית *template* בקובץ *header*.
אחרת אם רק נצהיר על הפונקציה ב-*header*, ונממש אותה בכל קובץ בנפרד,
הלינקר ישתמש במימוש אחד אקראי מבין המימושים השונים ויגרום להתנהגות בלתי צפויה.

מה נעשה אם יש לנו הרבה פונקציות $template$ שנצטרך לממש ב-$header$?

למדנו בשבועות קודמים שקובץ הידר טוב יהיה קצר ותמציתי. נראה פתרון אפשרי לבעיה:

```
#ifndef SWAP_H
#define SWAP_H
template <class T>
void swap( T& a,T& b ) {
        T tmp = a;
        a = b;
        b = tmp;
}
#endif
```

נניח שהשיטה $swap$ לעיל היא פונקציה ארוכה שלא נרצה שתופיע ישירות ב-$header$.

ניצור קובץ חדש $swap.hpp$ שיכיל את המימוש לשיטה באופן הבא:

```
// swap.hpp
template < class T >
void swap( T& a,T& b ) {
        T tmp = a;
        a = b;
        b = tmp;
}
```

וקובץ ה-$header$ ישתנה ל:

```
// swap.h
#ifndef SWAP_H
#define SWAP_H
template < class T >
void swap( T& a,T& b );
#include "swap.hpp"
#endif
```

נעשה זאת לכל פונקציה ארוכה שנרצה להוציא מה-$header$.

<u>הערה</u>: ניתן לרשום את תוכן הקובץ $swap.hpp$ גם ישירות בקובץ $swap.h$ מחוץ להגדרת המחלקה,

אך עבור מספר גדול של פונקציות ארוכות, קובץ ה-$header$ שלנו יהיה גדול מדי,

ולכן נעדיף לפצל אותו לקבצים.

Overloading Template Functions

עבור הקריאה $f(a,b)$, הקומפיילר מתנהג לפי השלבים הבאים:

1. מחפש אחר פונקציה f רגילה עם הטיפוסים המתאימים של a ו-b.
2. מחפש אחר פונקציית $template$ בשם f , ומייצר מופעים לפי הפרמטרים a ו-b.
   תתבצע התאמה מדויקת בלבד. כלומר אם a הוא int ו-b הוא $double$,
   אז רק שיטת $template$ מהצורה $f(int\&, double\&)$ תתאים.
3. מסדר את כל הפונקציות שמצא לפי ההתאמה שלהן
   $(exact\ match > promotion > conversion)$
4. אם יש מספר פונקציות מתאימות, הקומפיילר יעדיף שיטות שהן לא $template$.

<u>הערה</u>: עבור $f\langle some\_type\rangle(a,b)$ $(explicit\ call)$ יתבצעו רק שלבים 2-4.

בהרצאה 10 , שקופיות 36-65 יש הרבה דוגמאות **חשובות** שמתארות את התהליך הנ"ל:
https://drive.google.com/drive/u/2/folders/1y8gMtyUfziabSgVdmzUqCBS5nrKla3nk

Template Classes

עד עכשיו ראינו איך לבנות פונקציות גנריות. מה לגבי מחלקות גנריות?

כשאנחנו מייצרים וקטור חדש מהצורה $vector\langle int\rangle\ vec;$ אנחנו מגדירים לקומפיילר שאנחנו רוצים לייצר וקטור שכל השיטות, הבנאים והשדות שלו שמסומנים עם T, יאותחלו עם int במקום T.

כדי לאפשר זאת, המחלקה $vector$ היא מחלקת $template$. נראה דוגמה:

```
template ⟨class T⟩
class genericClass {
        T _member;
public:
        void print() const;
};
```

כלומר הסינטקס דומה. בשורה הראשונה הגדרנו את T להיות משתנה גנרי,

ולכן כל ה-$Scope$ שבא אחריו (הגדרת המחלקה) מכיר בו.

אפשר לחשוב על זה כמשתנה זמני שמוחלף בזמן קומפילציה לטיפוס הנכון.

נייצר מופעים של המחלקה שלנו באופן הבא:

```
int main() {
        genericClass ⟨int⟩ intObj;
        genericClass ⟨float⟩ floatObj;
        ...
}
```

מספר משתנים גנריים

נוכל להגדיר מחלקה שמקבלת מספר משתנים גנריים, כמו map שמקבל גם מפתח וגם ערך:

```
template <class T, class U>
class genericClass {
...
```

ואתחול מופע של המחלקה שלנו יתבצע על ידי $genericClass\langle int, double\rangle(90,1.4)$ למשל.

(אפשר גם $genericClass\langle int\rangle(90,1.4)$ והקומפיילר יזהה לבד שהארגומנט השני הוא $double$)

ארגומנטים

בנוסף למשתנים גנריים, ניתן לשלוח גם $template\ arguments$. כלומר ערכים, ולא טיפוסים:

```
template <class T = char, int Size = 1024>
class Buffer {
        T m_values[Size];
};
Buffer <char> buff1;
Buffer <> buff2; // same as Buff1, needs <>
Buffer <int, 256> buff3;
```

המופע הראשון שיצרנו יאותחל עם משתנה $Size$ עם הערך 1024, והמשתנה הגנרי שלו יהיה $char$.

המופע השני שיצרנו הוא בדיוק כמו המופע הראשון, רק שלא ציינו במפורש שמדובר ב-$char$ כי זה ערך ברירת המחדל גם ככה. המופע השלישי מציין גם משתנה גנרי שונה וגם ערך שונה ל-$Size$.

הערות:

- ערכים דיפולטיביים ניתנים החל מ-$C++11$ בלבד.
- בדוגמה הנ"ל, $Buffer$ הוא לא סוג של טיפוס. $Buffer\langle char, 1024\rangle$ הוא כן.
  $Buffer\langle char, 1024\rangle, Buffer\langle char\rangle, Buffer <>$ הם 3 שמות לאותו טיפוס !
  $Buffer\langle char, 256\rangle$ הוא טיפוס שונה !
  לכן נשתמש ב-$template\ arguments$ רק אם הערך של $Size$ הוא יחסית קבוע,
  כדי למנוע מהקומפיילר לייצר הרבה עותקים לכל הטיפוסים השונים.
  החלופה היא פשוט לשלוח את $Size$ לבנאי.
- $template\ arguments$ יכולים להיות מחלקות, $templates$ (לא נרחיב על זה בקורס),
  פרימיטיביים, מצביעים (כולל $nullptr$) ורפרנסים.

Template specialization

במקרים מסוימים נרצה לממש עותק של פונקציה גנרית עם טיפוס ידוע. למשל עבור שיטה שמבצעת $transpose$ למטריצה גנרית, נרצה לממש שיטה גנרית עבור טיפוסים רגילים,

ושיטה ספציפית עבור מטריצות עם מספרים מרוכבים שמיוצגים על ידי מחלקת $Complex$.

השיטה הגנרית תהיה למשל:

```
template <typename T>
Matrix<T> Matrix<T>::trans() const { ... }
```

והשיטה הספציפית תהיה:

```
template <>
Matrix<Complex> Matrix<Complex>::trans() const { ... }
```

ונזכור בתוך מימוש הפונקציה שבכל מקום בו היה לנו T במימוש הגנרי, נרשום $Complex$.

בצורה זו, כפי שכבר הסברנו בעמודים קודמים, הקומפיילר יעדיף את השיטה הספציפית על הגנרית,

וכך נוכל לייצר שיטה גנרית לכל הסוגים, ושיטות ספציפיות למקרים מיוחדים. דוגמאות נוספות:

```
template <typename T1, typename T2>
class MyClass {
        ...
};
// partial specialization: same types parameters
template <typename T>
class MyClass <T, T> {
        ...
};
// partial specialization: second type is int
template <typename T>
class MyClass <T, int> {
        ...
};
// partial specialization: parameters are pointers
template <typename T1, typename T2>
class MyClass <T1*, T2*> {
        ...
};
```

ואז:

```
MyClass <int, float> a;  //uses MyClass<T1, T2>
MyClass <float, float> b;  //uses MyClass<T, T>
MyClass <float, int> c;  //uses MyClass <T, int>
MyClass <int*, float*> d;  //uses MyClass<T1*, T2*>
```

פולימורפיזם ותבניות

מתי נרצה להשתמש בירושה ומתי בתבניות? נראה יתרונות וחסרונות של שניהם:

	$Inheritance\ \&\ Polymorphism$	$Templates$
$Code\ size$	$Stays\ small$	$Could\ bloat$
$Compilation\ time$	$Faster$	$Slower$
$Running\ time$	$Slower$	$Faster$
$Type\ checking$	$Explicit\ in\ declaration$	$Implicit\ in\ related\ code$
$Subtyping$	Yes	No

<u>הערה לגבי השורה האחרונה</u>: שימוש ב-$Templates$ לא מאפשר לנו לאגד מספר סוגים שונים של טיפוסים תחת מבנה אחד. כי $vector\langle int\rangle$ ו-$vector\langle double\rangle$ הם 2 טיפוסים **שונים**.
להבדיל מירושה, אין ביניהם קשר (כמו מחלקת אב) שמאפשר לאגד אותם תחת מבנה אחד.

<u>כלל אצבע</u>: תבניות טובות יותר עבור מבנים ואלגוריתמים סטנדרטיים.
ירושה ופולימורפיזם טובים יותר עבור יצירת תוכנית עם קשר לוגי.
ניתן לשלב תבניות וירושה במספר דרכים, וזה כי תבנית היא מחלקה.

בהרצאה 11 עמודים 33-38 יש דוגמה מעולם המולקולות לגבי בחירה בין ירושה לתבניות.

Lvalue & Rvalue

נגדיר את המושגים ואחר כך נראה דוגמאות שיבהירו אותם:

<u>$lvalue\ (locator\ value)$</u> – ביטוי שמתייחס למיקום קיים בזיכרון. ניתן לקרוא ממנו ולכתוב אליו, ולכן הוא יכול להופיע גם בצד ימין של השמה (=) וגם בצד שמאל של השמה.

<u>$rvalue$</u> – כל מה שהוא לא $lvalue$. $rvalue$ הוא ביטוי שניתן רק לקרוא ממנו, כלומר לא ניתן לבצע לתוכו השמה ולכן הוא יכול להופיע רק בצד ימין של השמה (=). נראה דוגמאות:

```
int foo() {
        return 2;
}
int main() {
        foo() = 2; //ERROR: lvalue required as left operand of assignment
        return 0;
}
```

בתרגילים שפתרנו, במידה והיינו צריכים להחזיר משתנה לוקאלי כערך החזרה, היינו חייבים להחזיר אותו $by\ value$. לאחר מכן כשעשינו $int\ a = foo();$, העתקנו (בעזרת האופרטור =) את ערך ההחזרה של foo לתוך a.

לבסוף ערך ההחזרה **הזמני** של הפונקציה נמחק. בדוגמה הנ"ל אנחנו מנסים לבצע השמה לתוך ערך ההחזרה של הפונקציה, אך הוא כבר לא קיים. כלומר הביטוי $foo()$ הוא $rvalue$, כי לא ניתן לבצע לתוכו השמה. באותו אופן, אם הפונקציה $foo()$ הייתה מחזירה $int\&$, היינו מקבלים שגיאה דומה הפעם עם $rvalue$, בגלל שניסינו לשלוח כתובת בזיכרון של הקבוע 2 שהוא $rvalue$. (ואין לו כתובת בזיכרון) לעומת זאת, הדוגמה הבאה תקינה:

```
int globalvar = 20;
int& foo() {
  return globalvar;
}
int main() {
  foo() = 10;
  return 0;
}
```

וזה בגלל שהשיטה $foo()$ מחזירה רפרנס למקום **קיים** בזיכרון (המשתנה $globalvar$), כלומר המשתנה לא לוקאלי ולכן ערך ההחזרה לא "נמחק" ביציאה מהשיטה.

דוגמאות נוספות:

```
int a = 1;
a = 5; // Lvalue = Rvalue, Ok
a = a; // Lvalue = Lvalue, Ok
5 = a; // Rvalue = Lvalue Comp.error
5 = 5; // Rvalue = Rvalue Comp.error
(a + 1) = 5; // Rvalue = Rvalue Comp.error
```

המרות בין $lvalue$ ל-$rvalue$:

ניתן להמיר ביטויי $lvalues$ באופן בלתי מפורש:

```
int a = 1; // a is an lvalue
int b = 2; // b is an lvalue
int c = a + b;
```

כלומר למרות ש-a ו-b הם $lvalues$, מתבצעת המרה לא מפורשת ו-$a + b$ הוא ביטוי $rvalue$.

ניתן להמיר ביטויי $lvalues$ גם באופן מפורש:

```
int var = 10;
int *bad_addr = &(var + 1); // ERROR: lvalue required as unary '&' operand
int *addr = &var; // OK: var is an lvalue
&var = 40; // ERROR: lvalue required as left operand of assignment
```

כלומר האופרטור & ($address - of$) מאפשר לנו לבצע המרה מפורשת של ביטויי $lvalues$

לעומת זאת, ביטויי $rvalues$ ניתן להמיר **רק** בצורה מפורשת:

```
int arr[] = {1,2};
int *p = &arr[0];
*(p + 1) = 10; // OK: p + 1 is an rvalue, but *(p + 1) is an lvalue
```

כלומר האופרטור * ($dereference$) מאפשר לנו לבצע המרה מפורשת של ביטויי $rvalues$

$R\backslash L$ $value$ and $References$

ל-$non - const\ reference$ אפשר להכניס רק $non - const\ lvalue$.

לעומת זאת, ל-$const\ reference$ אפשר להכניס גם $lvalue$ וגם $rvalue$.

```
int lv = 1;
const int clv = 2;
int& lvr1 = lv;
int& lvr2 = lv + 1; // error! why?
int& lvr3 = clv; // error! why?
const int& cr1 = clv;
const int& cr2 = 5 + 5;
```

- השגיאה הראשונה מתקבלת כי $lv + 1$ הוא $rvalue$, ואנחנו מנסים לבצע השמה שלו ל-$rvalue$. (השתמשנו ב-& לבצע המרה של $lvr1$ מ-$lvalue$ ל-$rvalue$)
- השגיאה השנייה מתקבלת כי $lvr3$ הוא רפרנס רגיל (ולא $const$) ולכן הוא לא יכול להצביע ל-$const\ lvalue$.

Rvalue reference (move)

ראינו בעמודים קודמים שלא ניתן לייצר רפרנס ל-$rvalue$ (כי אין לו כתובת בזיכרון).

החל מ-$C++11$ ניתן לייצר $rvalue\ reference$, כלומר רפרנס "מיוחד":

```
int && r1 = 3;
int i = 3, j = 5;
int && r2 = i + j;
int && r3 = i; // error! (i is lvalue)
```

כלומר בעזרת && ניתן להצביע על משתני $rvalue$ (כמו קבועים) שאמרנו שלא ניתן להצביע עליהם.

```
int foo(int *input) {
        return *input;
}
int a = 3;
int& r = foo(&a); //error!
int&& r = foo(&a); //Ok!
```

ניתן לבצע $overloading$ לשיטות עם &&:

```
void foo( int& input ){...}
void foo( int&& input ){...}
int main() {
        int i = 3;
        foo(i); // will call the first foo
        foo(3); // will call the second foo
}
```

כלומר מספיק שהשינוי הוא & או && ומתבצע $overloading$.

בצורה זו אנחנו יכולים לייצר שיטה שמקבלת רק משתני $rvalue$.

<u>$std :: move$</u>

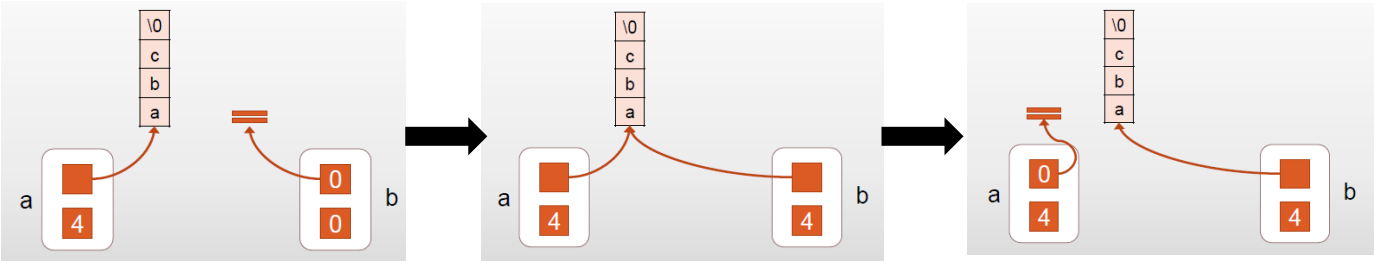
$move$ היא פונקציה של std המקבלת ביטויי $rvalue$ או $lvalue$ והופכת אותם ל-$rvalue$,

ובכך מאפשרת לנו לקבל $rvalue\ reference$. למה זה טוב? נראה בעמוד הבא.

Move Semantics

בעמודים קודמים תיארנו את השלבים שמתבצעים עבור הביטוי $int\ a = foo();$. באחד השלבים ביצענו העתקה מיותרת שיכלנו לחסוך אם הייתה דרך להשתמש בערך ההחזרה של הפונקציה (שכבר קיים), מבלי להעתיק אותו ואז למחוק אותו. אז יש דרך, והיא נקראת $move\ semantics$.
בעזרת $rvalues\ references$ נוכל לייצר שיטות שמקבלות משתנים זמניים (למשל ערכי החזרה של פונקציות), ולגנוב אותם. כלומר, במקום להעתיק ולמחוק נוכל פשוט להעביר אותם אלינו.
השימוש הנפוץ ביותר ב-$rvalue\ reference$ הוא ב-$move\ constructor$ ו-$operator\ =$.
ב-2 המקרים נוכל לממש שיטות נוספות שמקבלות && ל-$rvalue$ ואז במקום להעתיק את כל הערכים של האובייקט הנתון, נוכל להעביר אותם אלינו ולמנוע העתקות מיותרות. נראה דוגמה.

בעמודים 27-28 ראינו דוגמה למימוש מחלקה $MyString$ עם מצביע $(char\ *)$ למחרוזת, שהצריכה דריסה של המימוש הדיפולטיבי של בנאי ההעתקה, אופרטור ההשמה והמפרק.
כעת לאחר שהבנו מה זה $move\ semantics$, עבור מחרוזת **זמנית** a מסוג $MyString$ (שמייצגת למשל ערך החזרה של פונקציה), נרצה "לגנוב" את המחרוזת שלה מבלי לבצע העתקה:



כך המפרק של a לא ימחק את המחרוזת מהזיכרון וגם לא ביצענו העתקה מיותרת:

```
MyString::MyString(MyString&& other) { // move ctor implementation
        _string = other._string;
        other._string = nullptr;
        _length = other._length;
}
```

כלומר, יצרנו בנאי הזזה שמקבל $rvalue\ reference$ ומבצע שינוי מצביעים כנדרש.
בכך שהבנאי מקבל $MyString$&& אנחנו יכולים להיות בטוחים שמדובר במשתנה $rvalue$, כלומר משתנה שאף אחד לא משתמש בו, והוא רגע לפני פרידה מהעולם.
לכן אנחנו יכולים לגנוב ממנו ערכים בביטחון מלא שאף אחד לא יוכל לגשת אליהם אחר-כך.
בנוסף, מובטח לנו שעבור משתנים שהם לא $rvalue$, תתבצע קריאה לבנאי העתקה.

הערה: באותו אופן נצטרך לדרוס את $operator =$ $(move\ operator =)$
שמקבל $MyString\&\&\ other$ ומבצע את אותה פעולה.

בנוסף, נצטרך להרחיב גם את "חוק השלוש" שלנו ל"חוק החמש".
אם בנינו מחלקה והגענו למסקנה שצריך לממש את אחד מהבאים בעצמנו:
$destructor, copy\ constructor, operator =, move\ constructor, move\ operator =$
ככל הנראה אנחנו צריכים לממש את כולם !

אפשר לקרוא כאן: http://stackoverflow.com/a/3279550/2586599
על הדרך הנכונה לממש את כל החמישה הנ"ל בצורה נכונה עם מינימום שכפול קוד.

כל הקונטיינרים ב-STL מממשים $move\ semantics$,
כלומר הקוד שלנו כבר פועל לפי ההסבר הנ"ל בכל מה שקשור למבני הנתונים של STL.

מתי תתבצע קריאה לבנאי העתקה / בנאי הזזה?

- בקריאה מפורשת: $A\ a(b);$ תפעיל את בנאי ההעתקה אם b הוא $lvalue$,
  ותפעיל את בנאי ההזזה אם b הוא $rvalue$.
- כשפונקציה מקבלת ארגומנט $by\ value$,
  נוצר עותק חדש בעזרת בנאי העתקה אם מדובר ב-$lvalue$
  או בעזרת בנאי הזזה אם מדובר ב-$rvalue$
- אם ערך ההחזרה של פונקציה הוא $by\ value$,
  תתבצע קריאה לבנאי הזזה כל עוד יש כזה, אחרת תתבצע קריאה לבנאי העתקה.
- הפקודה $A\ a = b;$ כאשר b גם מטיפוס A,
  תפעיל את בנאי ההעתקה אם b הוא $lvalue$, ותפעיל את בנאי ההזזה אם b הוא $rvalue$.

לא תתבצע קריאה לבנאי העתקה / הזזה במידה ומדובר בערך החזרה $by\ reference$,
או בארגומנט שהפונקציה מקבלת $by\ reference$.

הערה חשובה: הקומפיילר יממש עבורנו את:
$move\ constructor$
$move\ assignment\ operators$

רק בתנאי שלא מימשנו אף אחד מהשיטות הבאות:
$copy\ constructors$
$destructors$
$copy\ assignment\ operators$

Exceptions

ב-$Java$ למדנו והשתמשנו בחריגות. חריגות ב-C פועלות באופן דומה אך קצת פחות מתוחכם,
ועם דגש של ניהול זיכרון שאין ב-$Java$. סדר הפעולות לשימוש בחריגות:

- הגדרת מחלקות של חריגות (עם היררכיה)
- זריקת חריגה ($throw$)
- תפיסת חריגה ($catch$)
- ריצת התוכנית ממשיכה לאחר ה-$try - catch$.

דוגמה לזריקת חריגה ושימוש ב-$try - catch$:

```
#include <iostream>
using namespace std;
int main () {
        try {
                throw 20;
        } catch (int e) {
                cout << "Exception occurred: " << e << endl;
        }
        return 0;
}
```

במקרה של חריגה, יודפס: $Exception\ occurred$: 20

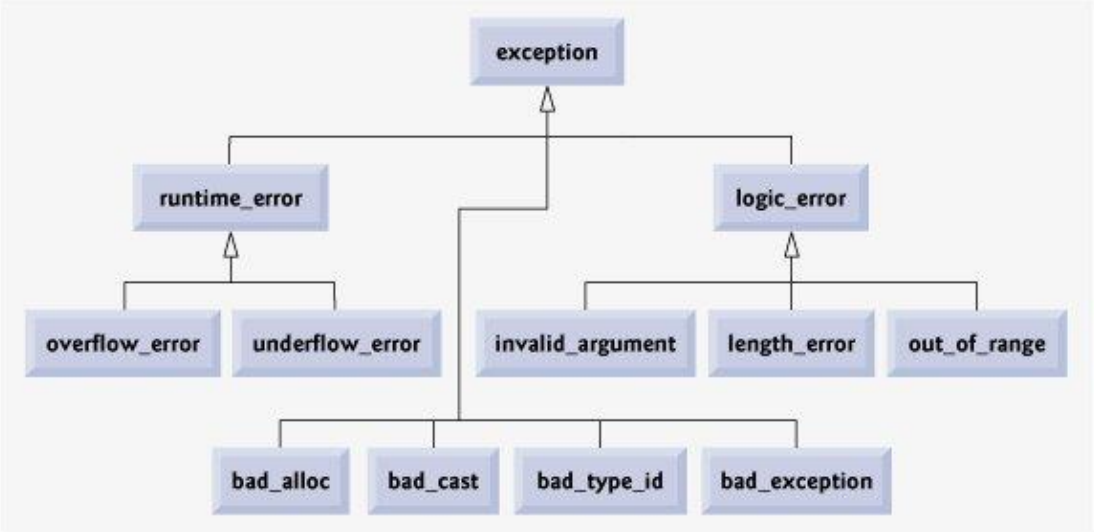
<u>הערה</u>: ב-$C++$ ניתן "לזרוק" הכל. פרימיטיביים, מצביעים, אובייקטים (עם בנאי העתקה) וכו'.

```
#include <iostream>
using namespace std;
int main() {
        cout << "1";
        try {
                cout << "2";
                if (true) throw 2;
                        cout << "3";
        } catch (int num) {
                cout << "4";
        }
        cout << "5" << endl;
}
```

בדוגמה זו יודפס "1245" . באותו אופן כמו $Java$, במקרה של חריגה בתוך בלוק $try - catch$,
לא נמשיך הלאה בביצוע אלא נצא ישר ל-$catch$. ולכן 3 לא הודפס.

בדרך כלל נרצה להגדיר מחלקות מיוחדות של חריגות שלרוב מוגדרות תחת היררכיה מסוימת.

חריגות של הספרייה הסטנדרטית:



דוגמה לתפיסת חריגה של הספרייה הסטנדרטית:

```
int main(void) {
        std::vector<int> vec(10);
        try {
                vec.at(20) = 100;
        } catch (const std::out_of_range& ex) {
                std::cerr << "out of range err:" << ex.what() << std::endl;
        }
        return 0;
}
```

נשים לב שתפסנו את האקספשן כ-$const\ reference$. מעבר ליעילות, נתפוס שגיאות בעזרת רפרנס או מצביע כדי לאפשר פולימורפיזם של חריגות. כלומר גם חריגות שיורשות מהחריגה שתפסנו , ייתפסו באותו $catch$. אם היינו תופסים את החריגה $by\ value$ זה לא היה מתאפשר, והיינו תופסים רק את החריגה מאותו סוג ספציפי. בנוסף, לתפוס חריגות עם $const$ זה הרגל טוב.

```
try {
        ...
} catch (CollectionE& e) {
        e.printErrorMessage();
        throw; // re - throw the exception
}
```

בדוגמה זו השתמשנו ב-$printErrorMessage()$ כדי להדפיס את הודעת השגיאה,
ולאחר מכן רשמנו $throw;$ כדי לזרוק את השגיאה הלאה.
(רק תחת $catch$ ניתן לרשום $throw$ בלי כלום אחריו)

הערות:

- ניתן להוסיף יותר מ-$catch$ אחד ובכך לתפוס מספר חריגות. נזכור רק שסדר ה-$catch$ חשוב, ואם נרצה לתפוס חריגה שיורשת מחריגה אחרת, החריגה היורשת תהיה ראשונה.
- ניתן לתפוס כל סוג של חריגה בעזרת הסינטקס הבא:

```
catch (...) { // catch ANY exception
        //...
}
```

כלומר 3 נקודות בתוך ה-$catch$ יגרמו לכך שנתפוס כל חריגה אפשרית.
במקרה זה לא נוכל להשתמש בחריגה שתפסנו (למשל לקבל את פירוט השגיאה),
כי אנחנו לא יודעים איזה חריגה תפסנו.

- ניתן לבצע $try - catch$ מקוננים.
- חריגה שלא תיתפס תגרום לקריסת התוכנית.
- בשונה מ-$Java$, בזמן חריגה לא יודפס למסך מידע נוסף (כמו למשל איפה החריגה נזרקה), ניתן להשתמש ב- backtrace (בשימוש עם gcc) או ספריות ייעודיות אחרות בשביל זה.
- כשנזרקת חריגה מתוך פונקציה, עבור כל המשתנים של הפונקציה הזו מתבצעת קריאה ל-$destructor$ (בסדר הפוך), אך עבור קבצים פתוחים או זיכרון דינמי נצטרך לטפל בהם בעצמנו בתוך ה-$catch$. (לסגור קבצים או להוסיף $delete$)

בשונה מ-$Java$, ב-$C++$ פונקציות לא חייבות להצהיר איזה חריגות הן זורקות.
עבור $int\ f();$ ניתן לרשום $int\ f()\ throw\ (BadArg\ int\ ^*);$
ואז $f()$ יכולה לזרוק רק $BadArg$ ו-int^* , אך זה לא מומלץ !
כלומר כל עוד כן יכולה להיזרק שגיאה מהפונקציה – לא נרשום כלום.
לעומת זאת, כדי לציין שפונקציה מסוימת לא יכולה לזרוק שגיאה, נרשום:
$void\ g()\ throw();$ או החל מ-$C++11$ כך: $void\ g()\ noexcept;$, נעדיף את האופציה השניה.

חריגות בבנאים:

במקרה של חריגה מהפקודה הבאה: $MyClass\ ^*m = new\ MyClass();$

- החריגה תיזרק על ידי האופרטור new.
- אובייקטים שהבנאי בנה כבר, יהרסו על ידי קריאה למפרקים שלהם.
- אם האובייקט עצמו נבנה ב-$heap$ (כלומר על ידי new) אין צורך לקרוא ידנית ל-$delete$, כיוון שהאופרטור new דואג למחוק את הזיכרון שהוקצה.

הערה חשובה: האופרטור new לא קורא ל-$destructor$ של $MyClass$ בדוגמה זו,
אלא הוא פשוט משחרר את הזיכרון שהוא עצמו הקצה עבור האובייקט.
מסיבה זו, אם הבנאי מקצה בעצמו אובייקטים על ה-$heap$, לא נוכל לבנות על המפרק שישחרר אותם, נצטרך לטפל באפשרות של חריגה כבר בבנאי עצמו. כלומר הבנאי צריך להתמודד עם החריגות שהוא אולי יכול לייצר. נראה דוגמה בעמוד הבא.

```
struct Incomplete{
        Incomplete() : _ip{nullptr}{
                cerr << "Allocated some memory\n";
                try{
                        _ip = new int;
                        // ...more code that can throw
                } catch(...) {
                        delete _ip;
                        throw;
                }
        }
        ~Incomplete() {
                cerr << "Destroying the allocated memory\n";
                delete _ip;
        }
        int *_ip;
};
```

כלומר הוספנו $delete\ \_ip$ בבנאי למרות שיש גם במפרק.

מה אם השגיאה תיזרק ברשימת אתחול? למשל עבור $Incomplete() : \_ip\{new\ int\{0\}\}\ \{$

נשתמש בסינטקס הבא:

```
Derived ()
try     // function – try block begins before the function
        // body, which includes init list
        : Base{789}, x{0}, y{0}
        { /* constructor body ...*/ }
catch (...)  {
        /* exception occurred on initialization */
}
```

<u>הערה 1</u>: בבנאי נרצה לתפוס חריגות בעזרת $catch(...)$ כי במקרה זה לא משנה לנו מה השגיאה, אנחנו רוצים לתפוס אותה ולשחרר את הזיכרון שהקצנו.

<u>הערה 2</u>: נזכור שניתן להוסיף $new(std::nothrow)$ ובכך למנוע מ-new לזרוק חריגה. במקום זאת , הערך המוחזר יהיה $nullptr$. במקרה זה, רק שגיאות מ-new ייתפסו, ולא שגיאות שהבנאי זורק !

<u>הערה 3</u>: לא נזרוק חריגות מתוך ה-$destructor$!! (גם אם הן נזרקות על ידי פונקציות אחרות, נתפוס אותן ונמנע מהן להמשיך הלאה) כלומר, נרצה שהמפרקים יהיו תמיד $noexcept$ (אין צורך להוסיף זאת ידנית, הקומפיילר מוסיף לבד)

<u>הערה 4</u>: גם ב-$move\ constructor$ לא נזרוק חריגות, אחרת ספריית ה-STL לא תשתמש בו אלא בבנאי העתקה, וזה פחות יעיל. במקרה זה כן נוסיף ידנית $noexcept$.

(גם לבנאי הזזה וגם ל-$swap$ אם יש)

חריגות הקצאת זיכרון

במקרה של שגיאה בהקצאת זיכרון על ידי new, תיזרק שגיאת $std::bad\_alloc$

```
int main() {
try {
        while(true)
                new int[100000000];
} catch (std::bad_alloc e) {
        std::cout << "allocation failure\n";
}
```

אם נרצה מידע נוסף נוכל להשתמש ב-$e.what()$

באותו אופן, עבור מחלקות $Exception$ שנכתוב בעצמנו, נרצה לדרוס את השיטה $what()$:

```
class myex: public exception {
virtual constexpr decltype(auto) what() const noexcept {
        return "My exception happened";
}
};
```

במקרה זה ניתן להחליף את $decltype(auto)$ עם $\mathbf{const}\ char^*$.

ירושה מרובה

מחלקה יכולה לרשת ממספר מחלקות:

```
struct inputFile {
        inputFile() { cout << "inputFile ctor"; }
};
struct outputFile {
        outputFile() { cout << "outputFile ctor"; }
};
struct ioFile: public inputFile, public outputFile {
        ioFile() { cout << "ioFile ctor"; }
};
```

$ioFile$ יורשת גם מ-$inputFile$ וגם מ-$outputFile$. יודפס:

$$inputFile\ ctor, outputFile\ ctor, ioFile\ ctor$$

כלומר, סדר האתחול הוא לפי סדר רשימת המחלקות.

במידה ול-2 המחלקות $inputFile$ ו-$outputFile$ יש שיטה עם אותו שם - $open$,

הקוד הבא יגרום לשגיאת קומפילציה:

```
ioFile f;
f.open(); // Error!
```

אך ניתן לציין במפורש את המחלקה הרצויה:

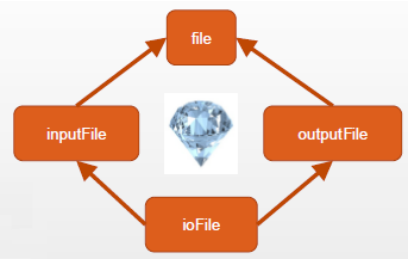
```
f.inputFile::open(); // Ok!
```

איך נפתור את הבעיה הזאת ונשמור רק מימוש אחד של $open$?

ירושת יהלום

כדי לפתור את הבעיה הנ"ל, נוציא שיטות ומשתנים עם מימוש דומה למחלקת אב נוספת $File$,

ונדאג ש-$inputFile$ ו-$outputFile$ ירשו ממנה:



אך ירושה רגילה לא תספיק. נצטרך לרשת את המחלקות בעזרת $virtual$:

```
struct file {
        char *name;
        void open();
};
struct inputFile: virtual public file {
        void read();
};
struct outputFile: virtual public file {
        void write();
};
struct ioFile: public inputFile, public outputFile {...};
```

בצורה זו, ל-$ioFile$ יהיה מימוש בודד של $open$, ומשתנה בודד $name$. ללא $virtual$ היינו נשארים עם אותה בעיה כי $ioFile$ היה יורש את $open$ ו-$name$ מ-2 מחלקות שונות.

בעיה: נניח שהמחלקות $inputFile$ ו-$outputFile$ מאתחלות כל אחת מופע של $File$.

אתחול של אובייקט מסוג $ioFile$ יאתחל 2 מופעים של $File$ עם אותו קובץ, למרות שאנחנו רוצים לפתוח את הקובץ פעם אחת. נפתור את הבעיה על ידי אתחול מופע של $File$ ב-$ioFile$,

המחלקה הנמוכה ביותר בעץ הירושה:

```
struct file {
        file(const char *name) {...}
        char *name;
};
struct inputFile: virtual public file {
        inputFile(const char *name): file(name) {} };
struct outputFile: virtual public file {
        outputFile(const char *name): file(name) {} };
struct ioFile: public inputFile, public outputFile {
    ioFile(const char *name): file(name), inputFile(name), outputFile(name) {}
};
```

<u>סדר הפעלת הבנאים</u>:

1. כל מחלקות האב הוירטואליות, בסדר בו הם מופיעים לפי DFS (כלומר מימין לשמאל) בעץ הירושה, כאשר הסדר מימין לשמאל בעץ הוא בהתאם לסדר בו הם מופיעים ברשימת הירושה של המחלקה.
2. כל מחלקות האב הישירות (לא וירטואליות) בהתאם לסדר בו הם מופיעים ברשימת הירושה של המחלקה.
3. המחלקה היורשת.

<u>*Virtual Base Class – Functions Overriding*</u>

```
struct A {
        virtual void f() { cout << "A::f()"; }
};
struct B1 : virtual A {
        void f() override { cout << "B1::f()"; }
};
struct B2: virtual A {
        void f() override { cout << "B2::f()"; }
};
struct C: B1, B2 {
        void f() override { cout << "C::f()"; }
};
int main() {
        C c;
        c.f(); // C::f()
        c.B1::f(); // B1::f()
        c.B2::f(); // B2::f()
        c.A::f(); c.B1::A::f(); c.B2::A::f(); // A::f()
}
```

<u>הערה</u>: במידה ולא היינו דורסים את f במבנה C, תהיה שגיאת קומפילציה מהסיבה שיהיו ל-C 2 שיטות עם אותו שם, מ-2 מחלקות האב $B1, B2$.

כדי לפתור את הבעיות שראינו לעיל, במקרה של ירושת יהלום נשתדל לעבוד כך:

- ניצור מחלקת אב "אמיתית" אחת
- ניצור מחלקה "וירטואלית" אחת או יותר כך ש:
  - המחלקה תכיל פונקציות $pure\ virtual$ (פונקציות $virtual$ ללא מימוש)
  - המחלקה לא תכיל שדות
  - המחלקה לא תכיל פונקציות $virtual$ נוספות

C++ Casting, RTTI

ב-C היינו ממירים אובייקט מטיפוס אחד לשני בעזרת $(type)$. למשל:

```
double d = 3.0;
int i = (int) d;
```

כמעט הכל אפשרי ב-C, ואחריות המתכנת לוודא שההמרה חוקית.

ב-$C++$ ניתן לבצע המרות באותה צורה, אך נוספו המרות נוספות חכמות יותר:

```
static_cast <type>(expression)
const_cast <type>(expression)
reinterpret_cast <type>(expression)
dynamic_cast <type>(expression)
```

<u>האופרטור $static\_cast$</u>:

נשתמש בו לכל:

- $cast$ שהיה עובד גם בצורה לא מפורשת.
- עבור המרת $up-casts$ סטנדרטית.
- כשאנחנו בטוחים שההמרה תצליח.

(למשל $down-cast$ לאחר $up-cast$)

שימוש ב-$static\_cast$ הוא בטוח יותר (למשל לא יאפשר המרה מ-int^* ל-$char^*$)

שגיאות המרה יהיו שגיאות קומפילציה !

```
int i = static_cast <int>(12.45); // OK
float *i = static_cast <float*>(&i); // ERROR
```

ההמרה השנייה נמנעת בזמן קומפילציה כי אנחנו מנסים להמיר מצביע $float^*$ ל-int^*,

אך המרה זאת יכולה לגרום להתנהגות בלתי רצויה.

<u>האופרטור $const\_cast$</u>:

נשתמש באופרטור זה כדי להמיר אובייקט $const$ ללא $const$. (או הפוך)

```
void g(C *cp);
void f(C const *cp) {
        g(const_cast <C*>(cp));
}
```

בדרך כלל נתכנן את הקוד שלנו בצורה כזאת שלא נצטרך להשתמש ב-$const\_cast$.

שימושי (אולי) למקרים שנרצה להחזיר משתנה שהפכנו ל-$const$ להיות לא $const$.

נזכור ש-$const$ מסמן לקומפיילר מתי אובייקט לא אמור להשתנות והקומפיילר יוודא זאת.

שימוש ב-$const\_cast$ יגרום לקומפיילר להתעלם מהמשתנה הנוכחי עליו ביצענו את ההמרה.

יכול ליצור בעיות חמורות.

```
int main() {
        int i = 3; // i is not declared const
        const int& cref_i = i;
        const_cast <int&>(cref_i) = 4; //OK: modifies i

        const int j = 3; // j is declared const
        int *pj = const_cast <int*>(&j);
        *pj = 4; // undefined behavior!
}
```

במקרה הראשון אין בעיה (הוספנו $const$).

במקרה השני, אנחנו מנסים לבטל $const$ למשתנה שמראש הוגדר כ-$const$,

ובמקרה זה ההתנהגות לא מוגדרת. מנגד, ניתן להפוך אובייקט ל-$const$

ואז לבטל את ה-$const$ וזה תקין (כי אנחנו יודעים שהמשתנה לא אותחל כ-$const$).

<u>האופרטור $reinterpret\_cast$</u>:

ה-$cast$ "המסוכן" ש-C מאפשר לנו. עוקף את ה-$type\ checking$ ומאפשר לבצע כל המרה שנרצה.

נשתמש בו כמה שפחות, מסוכן מאוד.

<u>האופרטור $dynamic\_cast$</u>:

למדנו ש-$up-casting$ ממחלקת בן למחלקת אב זו המרה חוקית. $Shape\ *s = new\ Circle();$

אך $down-casting$ יכול לגרום לבעיות. הנכונות לא יכולה להיבדק בזמן קומפילציה.

$dynamic\_cast$ מאפשר לנו לבצע $down-casting$ ולוודא בזמן ריצה האם ההמרה צלחה או לא.

עבור מצביעים, יוחזר המצביע הרצוי, אחרת יוחזר $nullptr$.

עבור רפרנסים, תיזרק שגיאת $std::bad\_cast$ במקרה של שגיאה.

```
Shape *s = container.pop();
Circle *c = dynamic_cast <Circle*>(s);
if ( c != nullptr ) // c is a circle
        c-> doSomething();
else
        handle(); // handle unexpected event
```

או

```
try {
        Shape& s = container.peek();
        Circle& c = dynamic_cast <Circle&>(s);
        c.doSomething(); // c is a circle
} catch (std::bad_cast) {
        handle(); // handle unexpected event
}
```

הערות:

- נשתמש ב-$dynamic\_cast$ רק על מצביעים ורפרנסים.
- ניתן להשתמש בו בשביל $up-cast, down-cast$ ו-$cross-cast$.
- רק לטיפוסים פולימורפיים, כלומר שיש להם $virtual-functions$, כי טבלת הפונקציות הוירטואליות שדיברנו עליה בעמודים קודמים מכילה את הטיפוס האמיתי של האובייקט.
  כך האופרטור ידע לבדוק בזמן ריצה האם ההמרה חוקית או לא.
- במידה ואין לאובייקט פונקציות וירטואליות, תהיה שגיאת קומפילציה.

$RTTI: typeid\ operator$

האופרטור $typeid()$ מקבל אובייקט, מחלקה או ביטוי (בדומה ל-$sizeof$),

ומחזיר $const\ std::type\_info\&$ - רפרנס לאובייקט סטטי שמחזיק מידע על ה-$type$ הרלוונטי.

```
#include <typeinfo>
int main() {
        Dog d;
        Cat *pc;
        void f(int, double);
        cout << typeid(Dog).name(); // 3Dog
        cout << typeid(d).name(); // 3Dog
        cout << typeid(pc).name(); // P3Cat
        cout << typeid(f).name(); // FvidE
        cout << typeid(main).name(); // FivE
}
```

קיבלנו רשימה של שמות פנימיים ($mangled$) בהערות בירוק, שלרוב לא יגידו לנו כלום.

כדי לפרש אותם לשמות מובנים עבורנו נשתמש ב- $c++filt$ או $\langle cxxabi.h \rangle$

גם $typeid$ דורש שהטיפוס יהיה פולימורפי (עם פונקציית $virtual$ אחת לפחות)

הערה: נעדיף להשתמש ב-$typeid$ רק אם אין פתרון אחר.

Smart Pointers

אחת מהבעיות העיקריות ב-$C/C++$ היא ניהול זיכרון דינמי, כאשר השאלה העיקרית היא האם הזיכרון שוחרר בסוף השימוש שלו או לא. נקודה חשובה נוספת היא הקצאה ושחרור בצורה יעילה מבחינת מקום ומבחינת זמן (new זו פעולה יקרה אך זה מחוץ לסקופ של הקורס)

אסטרטגיות לניהול זיכרון:

- $\underline{Fixed\ Ownership}$ – אסטרטגיה זו משאירה את אחריות שחרור המצביע למי שהקצה אותו - פונקציה או אובייקט. בדוגמה הבאה, הפונקציה foo מקצה זיכרון ומשחררת אותו.

```
void foo() {
        char *mem = new char[1000];
        //...
        delete [] mem;
}
```

דוגמה נוספת היא מחלקת $String$ שמחזיקה מצביע לרצף של תווים בזיכרון, היא מנהלת את המצביע, רק לה יש גישה אליו והיא זו שמשחררת אותו.

- $\underline{Dynamic\ Ownership\ (Compile\ time)}$ – בדומה לאסטרטגיה הקודמת, למצביע יש בעלים כמו למשל פונקציה או אובייקט. ההבדל הוא שבאסטרטגיה זו, הבעלים יכול להתחלף. כלומר, מי שאחראי על שחרור הזיכרון משתנה בזמן קומפילציה. למשל $strdup$ (מספריית $string$ של C) מקצה מקום בזיכרון ומחזיר מצביע למיקום הזיכרון, אך הוא לא זה שמשחרר אותו. כדי להימנע מדליפות זיכרון עקב שגיאות של המתכנת, נרצה לפעול לפי הגישה הבאה: מי שמקצה זיכרון אחראי גם על שחרורו (כלומר $fixed\ ownership$)

- $\underline{Dynamic\ ownership\ (run\ time)}$ – במקרים מסוימים לא נוכל או לא נרצה לזכור מי אחראי על מה בזמן קומפילציה. האם נוכל להגדיר מצביע חכם שידע לשחרר את עצמו לבד בזמן ריצה? התשובה היא כן ! נוכל לבנות מחלקה עוטפת (בסיסית) למצביע:

```
class pointer {
public:
        pointer(T*);
        pointer(pointer const&);
        pointer& operator = (pointer const&);
        T& operator*() const;
        T* operator-> () const;
}
```

כשאנחנו נייצר מופע של המחלקה, אנחנו נשלח לבנאי שלו מצביע T^* שקיבלנו מ-new, וכשהמופע יקרוס, הוא ישחרר את הזיכרון ב-$destructor$.

כמובן שמצביע חכם הוא משהו שימושי לכל מי שמתכנת בשפה, ולכן הוא כבר מומש עבורנו.
בעבר היו משתמשים ב-$auto\_ptr$, מחלקה עוטפת למצביע, ובכל פעם שפונקציה או אובייקט מקבל את המצביע הזה, הוא הופך להיות הבעלים החדש על המצביע. מי שלפני זה היה הבעלים, כבר לא מצביע לשם. הסיבה לכך היא למנוע מצב ש-2 מצביעים יצביעו לאותו מיקום בזיכרון, וישחררו אותו פעמיים. לכן $auto\_ptr$ מבצע **סוג** של $move$ למצביע, כלומר "גונב" אותו מהבעלים הקודם.
החל מ-$C++11$, $auto\_ptr$ לא בשימוש יותר והקומפיילר אמור להתריע לנו על שימוש בו.
במקומו נשתמש ב-$unique\_ptr$ הפועל באותו אופן כמו $auto\_ptr$ אך בטוח יותר.
נראה איך $auto\_ptr$ עובד, ולאחר מכן נגיע ל-$unique\_ptr$.

```
#include ⟨memory⟩
struct bar { /* ... */ };
void foo() {
        auto_ptr⟨bar⟩ myBar(new bar()); // use myBar as a raw pointer
} // myBar is deleted automatically
```

בצורה זו, גם במקרה של שגיאה, האובייקט $myBar$ יימחק על ידי קריאה ל-$destructor$,
ולכן גם $new\ bar()$ ישוחרר מהזיכרון. כלומר לא צריך לדאוג מבחינת שחרור זיכרון בחריגות.
איך נראית המחלקה $auto\_ptr$ (בצורה מופשטת)?

```
template ⟨typename T⟩
class auto_ptr {
public:
        explicit auto_ptr(T* t) : _ptr(t) {}
        auto_ptr( auto_ptr & );
        auto_ptr& operator = ( auto_ptr & );
        T& operator*() const { return *_ptr; }
        T* operator-> () const { return _ptr; }
        ~auto_ptr(){ delete _ptr; }
        //other methods
private:
        T* _ptr; // the actual pointer to the data
};
```

נסתכל על הדוגמה הבאה:

```
void foo() {
        auto_ptr⟨bar⟩ myBar(new bar());
        auto_ptr⟨bar⟩ myOtherBar = myBar;
} //calls destructors of myBar, myOtherBar
```

יצרנו מצביע $myBar$ מסוג $auto\_ptr$, ולאחר מכן העתקנו את המצביע ל-$myOtherBar$.
בגלל ש-$auto\_ptr$ לא מאפשר ל-2 מצביעים להצביע לאותו מקום, $myOtherBar$ גונב את המצביע ואז משנה את המצביע $myBar$ להצביע ל-$NULL$. כדי לאפשר זאת,
נשים לב ש-$operator =$ במחלקה של $auto\_ptr$ לא מקבל את המצביע השני כ-$const$.

כך ממומש למשל בנאי ההעתקה של $auto\_ptr$:

```
auto_ptr( auto_ptr& rhs ) : _ptr( rhs.release() ) {}
T* release() {
        T* tmp = _ptr;
        _ptr = NULL;
        return tmp;
}
```

ו-$operator =$ ממומש כך:

```
auto_ptr& operator = ( auto_ptr& rhs ) {
        reset( rhs.release() );
        return *this;
}
void reset(T* p = NULL) {
        if (_ptr != p) delete _ptr;
        _ptr = p;
}
```

כשאנחנו נכנסים לבנאי העתקה, האובייקט רק נוצר ולכן מספיק להגדיר את $\_ptr$,
ולגרום למצביע השני להצביע ל-$NULL$. במקרה של אופרטור= , המצביע הנוכחי כבר מצביע למיקום כלשהו בזיכרון, לכן נצטרך לשחרר אותו לפני ההחלפה. בנוסף, כפי שלמדנו בשלב מוקדם יותר בסיכום, אופרטור= צריך תמיד לוודא שהוא לא משחרר את עצמו בטעות, למשל במקרה הבא:

```
bar *newBar = new bar();
auto_ptr<bar> myBar1(*newBar);
auto_ptr<bar> myBar2(*newBar);
auto_ptr<bar> myBar1 = myBar2;
```

לכן הוספנו בדיקה בשיטה $reset$ כדי לוודא זאת.

כעת נראה איך $unique\_ptr$ פועל באותו אופן אך בטוח יותר, ונזכור לא להשתמש ב-$auto\_ptr$!
דבר ראשון, ל-$unique\_ptr$ אין בנאי העתקה ואין אופרטור=. הסיבה לכך היא למנוע מקרים בהם מצביע אחר "יגנוב" את המצביע שלנו בלי שהתכוונו לכך, ואחר-כך ננסה להשתמש בו.
כדי להעביר את המצביע שלנו למצביע אחר, נהיה חייבים לציין זאת במפורש:

```
// OK: argument is an rvalue
std::unique_ptr<int> p(new int);
// ERROR: p is an lvalue, no copy - constructor
std::unique_ptr<int> p2 = p;
// OK: argument is an rvalue
std::unique_ptr<int> p2 = std::move(p);
```

כלומר צריך לרשום במפורש $std :: move(p)$, וכך אנחנו בטוחים שאנחנו יודעים מה אנחנו עושים.
אחרת, הקוד לא יתקמפל !

4. <u>*Reference counting*</u> - $uniqur\_ptr$ ו-$auto\_ptr$ היו $Dynamic\ ownership\ (run\ time)$,
נראה 2 מצביעים נוספים שפועלים בעזרת ספירת מצביעים. מה נעשה במקרים בהם נצטרך שמספר אובייקטים או שיטות יצביעו לאותו מיקום? פתרון אחד הוא להחזיק עותקים רבים של אותו אובייקט ולהצביע עליהם בעזרת $unique\_ptr$, אך זה כמובן פתרון בזבזני. נראה פתרון נוסף. $\boldsymbol{shard\_ptr}$ הוא מצביע משותף, המאפשר את הפונקציונליות הנ"ל בדרך כלל בעזרת ספירת מצביעים. כלומר, המצביע ישמור $counter$ פנימי שיעקוב אחר מספר הישויות שמצביעים עליו, ואם ה-$counter == 0$ הוא ישחרר את עצמו.

איך זה מבוצע בפועל? ה-$counter$:

- יאותחל ביצירה ל-1.
- יגדל ב-1 בהעתקה (בנאי העתקה ואופרטור =)
- יקטן ב-1 ב-$delete$
- יקטן ב-1 בהשמה מחדש.

נראה דוגמת שימוש ולאחר מכן מימוש אפשרי (ומופשט) של $shared\_ptr$.

```
class Dog {
        string name;
public:
        Dog(string name);
        Dog();
        ~Dog();
        void bark() const;
};
int main() {
        shared_ptr<Dog> pD(new Dog("Sushi")); //count = 1
        {
                shared_ptr<Dog> pD2 = pD; //count = 2
        } //count = 1
} //count = 0: "Sushi" is deleted
```

בשורה הראשונה של ה-$main$ יצרנו מצביע משותף pD לכלב בשם סושי, וה-$counter = 1$.
לאחר מכן, בתוך $Scope$ פנימי יצרנו מצביע נוסף בשם $pD2$ שמצביע גם הוא לסושי, וה-$counter$ עודכן ל-1 כי 2 יש 2 מצביעים על סושי. כשה-$Scope$ קורס, $pD2$ קורס, ולכן ה-$counter$ מתעדכן להיות שוב 1. כשה-$main$ קורס, pD קורס, $counter == 0$, ולכן הכלב סושי משוחרר מהזיכרון.

<u>הערה</u>: נזכור שמצביעים חכמים משחררים את הזיכרון לבד לכן לא נשחרר את הזיכרון עליו הם מצביעים באופן ידני (כלומר בעזרת $delete$), ובאותו אופן לא נאתחל 2 מצביעים חכמים לאותו מיקום בזיכרון כי הם יקראו פעמיים ל-$delete$. **(תקף גם ל-$\boldsymbol{unique\_ptr}$)**

נראה מימוש מופשט של $shared\_ptr$

```
template <typename T>
class SPtr {
private:
        int * _cnt;
        T * _p;
public:
        explicit SPtr(const T* &p): _p(p),
        _cnt(new int(1)) {}
        SPtr(const SPtr& o) { *this = o; }
        T& operator*() const { return *_p; }
        T* operator-> () const { return _p; }
        SPtr& operator = (const SPtr& o) {
                if (_p != o._p) {
                        (*_cnt)--;
                        if ((*_cnt) == 0) {
                                delete _p;
                                delete _cnt;
                        }
                        _p = o._p;
                        _cnt = o._cnt;
                        (*_cnt)++;
                }
                return *this;
        }
        ~SPtr() {
                (*_cnt)--;
                if ((*_cnt) == 0) {
                        delete _p;
                        delete _cnt;
                }
        }
};
```

מעבר לשיטות הרגילות שמדמות מצביע, $operator =$ צריך לבדוק גם את המקרה הבא:

אם המצביע הנתון שונה מהמצביע הקיים, נעדכן את ה-$counter$ הנוכחי להיות 1 פחות

(כי אנחנו כבר לא נצביע עליו אלא למישהו אחר)

אם היינו המצביע האחרון שהצביע על $\_p$, נשחרר אותו (ואת המצביע)

לאחר מכן נעתיק את המצביע ואת ה-$counter$ של המצביע הנתון, ונגדיל את ה-$counter$ ב-1.

באותו אופן, ב-$destructor$ הקטנו את ה-$counter$ ב-1, ושחררנו את הזיכרון אם היינו האחרונים.

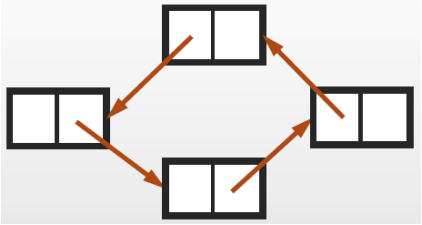
Custom Deleters

ראינו שמצביעים חכמים מקבלים כארגומנט מצביע בודד, אך אפשר לשלוח להם 2 מצביעים:
אחד למיקום הזיכרון הרצוי והשני מצביע לשיטה שמבצעת מחיקה. למדנו ששחרור מצביעים של מערכים צריכים להתבצע בעזרת *delete*[], אך ברירת המחדל של מצביעים חכמים היא *delete*.
לכן, אם נשלח למצביעים חכמים מערך, נצטרך לשלוח גם שיטה שמבצעת *delete*[]:

```
void arrayDeleter(Dog* p) { delete[] p; }
shared_ptr<Dog> pD_arr(new Dog[3], arrayDeleter);
```

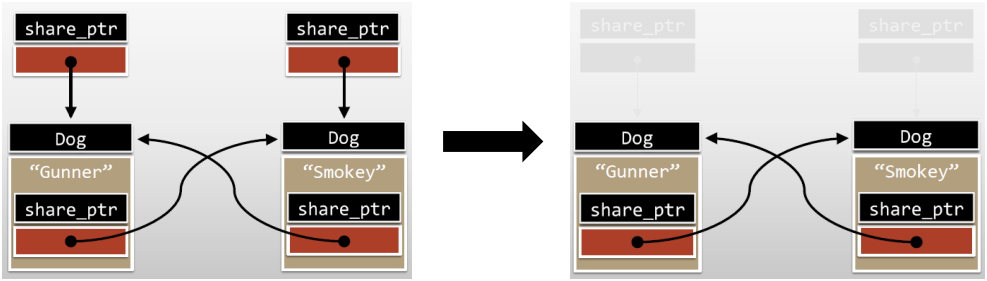
אחרת, תהיה דליפת זיכרון ! **(תקף גם ל-*unique_ptr*)**

cyclic reference problem



במידה וכל המצביעים בתמונה משמאל ממומשים בעזרת *reference counted*, לא יתבצע שחרור זיכרון,
כי *counter* לא יהיה 0 אף פעם.
נראה דוגמה ובעמוד הבא פתרון:

```
class Dog {
        shared_ptr<Dog> pFriend;
        string name;
public:
        Dog(string name);
        Dog();
        ~Dog();
        void bark() const;
        void makeFriend(shared_ptr<Dog> f) {
                pFriend = f;
        }
};
int main () {
        shared_ptr<Dog> pD1(new Dog("Gunner"));//count = 1
        shared_ptr<Dog> pD2(new Dog("Smokey"));//count = 1
        pD1-> makeFriend(pD2);//count = 2
        pD2-> makeFriend(pD1);//count = 2
}
```



נראה פתרון לבעיה הנ"ל:

weak_ptr – מצביע חכם נוסף שמאפשר להצביע על אובייקט שמנוהל בעזרת shared_ptr.

מצביע זה לא אחראי על האובייקט ולכן לא משפיע על ה-counter. קריסה שלו לא תשחרר את האובייקט עליו הוא מצביע, ובשונה ממצביע רגיל, weak_ptr לא מאפשר לשחרר את הזיכרון דרכו.

כדי למנוע שימוש במצביע לא קיים (למשל אחרי שה-counter של shared_ptr הגיע ל-0 והאובייקט שוחרר מהזיכרון), יש לו שיטות פנימיות שמאפשרות לבדוק האם הוא עדיין מצביע למיקום חוקי.

כדי לתקן את הבעיה בעמוד קודם, במקום shared_ptr⟨Dog⟩ pFriend; המחלקה תחזיק מצביע מסוג weak_ptr⟨Dog⟩ pFriend;.

make_shared:

בדוגמאות הנ"ל אתחלנו מצביע shared_ptr בעזרת:

```
shared_ptr⟨Dog⟩ p(new Dog("Gunner")); // count = 1
```

אך דרך מהירה וטובה יותר תהיה בעזרת פונקציה בשם std :: make_shared:

```
shared_ptr⟨Dog⟩ p = make_shared⟨Dog⟩("Gunner"); // count = 1
```

למה היא מהירה יותר? כי קריאה לבנאי של shared_ptr מבצעת לפחות 2 הקצאות זיכרון: לאובייקט T ולאובייקט shared_ptr שיצרנו. לעומת זאת std :: make_shared מאתחל מראש זיכרון רציף ל-2 האובייקטים בהקצאה אחת. נשים לב שלא השתמשנו ב-new באופציה השנייה.

באותו אופן, נשתמש בפונקציה std :: make_unique בשביל לייצר unique_ptr.

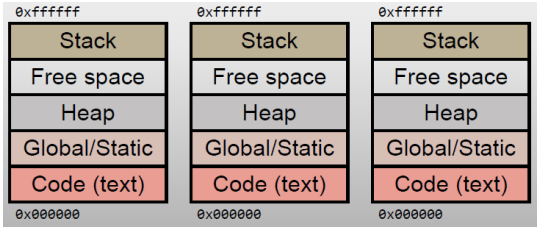
הערה: נשתמש ב-make_shared ו-make_unique רק אם שחרור הזיכרון מתבצע עם delete ולא עם פונקציה לא דיפולטיבית כפי שראינו לעיל.

Multithreading

תכנות אסינכרוני הוא תכנות המאפשר לבצע מספר פעולות במקביל. במחשבים עם יותר ממעבד אחד או מעבדים עם מספר ליבות, הפעולות באמת יכולות להתבצע בו-זמנית. במחשבים בעלי מעבד בודד עם ליבה אחת תחושת המקביליות מתקבלת באמצעות קפיצה מפעולה אחת לשנייה ואז חזרה לפעולה הראשונה וכן הלאה. נשתמש במושגים תהליכון ($thread$) ותהליך ($process$).
סדר הפעלת התהליכים לא מובטח ולכן נחלק לתהליכים שאינם תלויים אחד בתוצאה של השני.

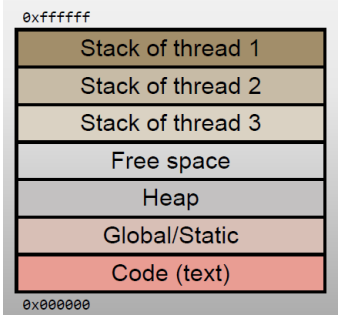
<u>איך נוכל להשיג תכנות מקבילי</u>?

1. נריץ מספר תהליכים במקביל:



כל תהליך מקבל זיכרון משלו, ולכן אם תהליך מסוים נתקל בשגיאה, שאר התהליכים ימשיכו לרוץ באופן תקין. בנוסף, ניתן להפעיל מספר תהליכים עם הרשאות שונות. התהליכים השונים לא יכולים לתקשר ביניהם - הם צריכים להשתמש במערכת ההפעלה בשביל זה.

2. נריץ מספר תהליכונים במקביל:



כל תהליכון מקבל מחסנית משלו, אך שאר הזיכרון משותף לכל התהליכונים,
לכן תהליכונים יכולים לתקשר ביניהם למשל דרך ה-$Global/Static\ Segment$.
מסיבה זו, שימוש בתהליכונים יכול לגרום לבעיות למשל בגישה לאובייקט משותף בו זמנית.

אופציה ראשונה:

```
#include <iostream>
#include <thread>
// This function will be called from a thread
void call_from_thread() {
        std::cout << "Launched by thread" << std::endl;
}
int main() {
        // Launch a thread
        std::thread t(call_from_thread);
        // main thread pauses until the created thread finishes
        t.join();
        return 0;
}
```

כדי לקמפל את הקוד הנ"ל נשתמש ב: $g++ -std=c++11 -pthread\ filename.cpp$

מה עשינו? יצרנו אובייקט מסוג $std::thread$ בשם t ושלחנו לבנאי שלו מצביע לפונקציה.

אחר-כך הפעלנו את $t.join()$, כדי שה-$main$ יחכה עד ש-t יסיים את ההרצה.

(אם לא, ה-$main$ יסתיים, יקרוס, האובייקט $std::thread$ יימחק ובפועל השיטה לא תספיק לפעול)

הערה: בדוגמה הנ"ל הפעלנו רק תהליכון אחד ובגלל זה השימוש ב-$join$ נראה מוזר (למה צריך לחכות שיסיים? מה מקבילי פה?), אך במקרה שנפעיל מספר תהליכונים, נוסיף את ה-$join$ שלהם בסוף באופן הבא:

```
std::thread t1(call_from_thread1);
std::thread t2(call_from_thread2);
t1.join();
t2.join();
```

ואז הם יפעלו במקביל.

אופציה שניה (וטובה יותר!):

```
#include <iostream>
#include <thread>
// This function will be called from a thread
void call_from_thread() {
        std::cout << "Launched by thread" << std::endl;
}
int main() {
        // Launch a thread
        auto result = std::async(call_from_thread);
        // main thread pauses until the created thread finishes
        result.wait();
        return 0;
}
```

1. השתמשנו ב-$std::async$ במקום ב-$std::thread$.
2. לא היינו צריכים לייצר אובייקט שונה לכל תהליכון.
3. $std::async$ מנהלת את כל התהליכונים בעצמה.
4. $std::async$ זורקת הרבה פחות חריגות מאשר $std::thread$.
   למשל אם יש אפשרות להריץ 8 תהליכונים במקביל לכל היותר,
   ואנחנו הפעלנו 9 פעולות במקביל, $std::thread$ תזרוק חריגה.
   לעומת זאת, $std::async$ תוסיף את המשימה התשיעית "לתור" של אחד מה-8.
5. קיבלנו (בעזרת $auto$!) את ערך ההחזרה של $std::async$ וכך הפעלנו את $wait()$,
   המקבילה של $join()$ מהאופציה הראשונה. גם כאן, נהיה חייבים להוסיף את $wait()$.

<u>הערה 1</u>: נוסיף את $wait()$ בכל מקום בו נהיה חייבים לחכות שהתהליכון יסיים. למשל לפני קריסה של ה-$main$, או למשל אם נצטרך לבצע פעולה שמסתמכת על פעולת התהליכון.

<u>הערה 2</u>: ערך ההחזרה של $std::async$ הוא מסוג $std::future\langle type\rangle$,
כש-$type$ הוא טיפוס ערך ההחזרה של הפונקציה ששלחנו. למרות זאת,
נעדיף תמיד לתפוס את ערך ההחזרה עם $auto$ כדי למנוע בעיות.

הערה 3: כדי לגשת לערך ההחזרה של הפונקציה נשתמש בפונקציה $get()$:
$result.get()$ בדוגמה הנ"ל.

<u>$Functors$</u>:

עד עכשיו דיברנו על $async$ בהקשר של פונקציות, אך למדנו שניתן להתייחס לאובייקטים כפונקציות, אם הם מממשים את $operator()$. על היתרונות של $Functors$ כבר דיברנו, נראה דוגמה:

```
struct SayHello{
        void operator()() const{
                std::cout << "hello" << std::endl;
        }
};
int main() {
        SayHello hello;
        auto res = std::async(hello);
        res.wait();
}
```

<u>הערה</u>: כשאנחנו שולחים $functor$ ל-$async$ הוא מועתק $by\ value$ לזיכרון הפנימי של $thread$ חדש, וה-$thread$ הזה מפעיל את $operator()$. האובייקט יכול להחזיק משתנים ופונקציות נוספות (חלק מהיתרונות).

איך נשלח ל-$async$ פונקציה שצריכה לקבל פרמטרים?

- בעזרת $functors$ (הפרמטרים יהיו משתנים של האובייקט, ו-$operator()$ ישתמש בהם)
- בעזרת $std::bind$

```
void greeting(std::string const& msg) {
        std::cout << msg << std::endl;
}
int main() {
        auto res = std::async(std::bind(greeting, "hi"));
        res.wait();
}
```

- פשוט לשלוח אותם:

```
void greeting(std::string const& msg) {
        std::cout << msg << std::endl;
}
int main() {
        auto res = std::async(greeting, "hi");
        res.wait();
}
```

הערה חשובה: המשתנים שנשלחים לפונקציה נשלחים $by\ value$!

$async$ ומתודות (פונקציות של מחלקה):

עד עכשיו שלחנו פונקציות גלובליות ל-$async$. במידה ונרצה להשתמש בפונקציות שמוגדרות בתוך מחלקה, נזכור לשלוח את $this$ בתור הפרמטר הראשון, כי הקומפיילר לא עושה זאת בשבילנו.

```
class Say {
public:
        void greeting(std::string const& msg) const {
                std::cout << msg << std::endl;
        }
};
int main() {
        Say x;
        auto res = std::async(Say::greeting, &x, "goodbye");
        res.wait();
}
```

שלחנו את $\&x$ בתור $this$. (שלחנו את הכתובת של x ! ולא רפרנס)

נזכור תמיד לוודא ש-x לא ימחק לפני שהתהליכון יסיים (כלומר לפני שהגענו ל-$res.wait()$)

למשל אם נגדיר במקום $Say\ x;$ את $std::shared\_ptr\ x = std::make\_shared\langle Say\rangle();$

ואז במקום לשלוח את $\&x$ נשלח את x.

העברת משתנים *by reference*

נשתמש ב-ref :: std ($wrapper\ for\ reference$) כדי לשלוח רפרנס ל-$async$:

```
void increment(int& i) { ++i; }
int main() {
        int x = 42;
        auto res = std::async(increment, std::ref(x));
        std::cout << "x = " << x << std::endl;
        res.wait();
}
```

שלחנו רפרנס ל-x, אך לא ניתן לדעת מה יודפס כי ההדפסה באה לפני $res.wait()$,

ובמקרה זה אי אפשר לדעת מה ירוץ קודם. כלומר x יכול להיות 42 ויכול להיות 43 בזמן ההדפסה.

אם נדפיס את x אחרי $res.wait()$ יודפס (בוודאות) 43.

תקשורת בין תהליכונים

אם התהליכונים שלנו לא צריכים לתקשר בינם לבין עצמם, ולא משתמשים במידע משותף, נוכל פשוט להריץ אותם ברקע ללא בבעיה. אם התהליכונים כן מתקשרים ביניהם, למשל על ידי גישה לאובייקט משותף, לא נוכל לדעת מתי כל תהליכון ייגש לאובייקט, ולכן יכול לקרות מצב בו 2 תהליכונים ינסו לשנות אותו במקביל. נצטרך למנוע גישה כל עוד מישהו אחר כבר ניגש אליו.

```
std::string s;
void append_with_no_lock(std::string const& extra) {
        s += extra;
}
int main() {
        s = "a";
        auto res1 = std::async(append_with_no_lock, "b");
        auto res2 = std::async(append_with_no_lock, "c");
        res1.wait();
        res2.wait();
        std::cout << s << std::endl;
}
```

כל תהליכון מוסיף תו למחרוזת. במקרה הטוב, נוסיף למחרוזת את "b" ו-"c" בלי לדעת באיזה סדר.

במקרה הרע, 2 הפונקציות יאתחלו מקום חדש בזיכרון בו זמנית, אחת מהן תספיק לשחרר את הזיכרון הקיים בזמן שהשנייה תנסה לגשת למיקום לא קיים.

כדי לפתור את הבעיה נוכל להשתמש ב-$Mutex$ – אובייקט שיכול להיות ב-2 מצבים, נעול או פתוח.
כשתהליכון מסוים ניגש לאובייקט הוא "נועל" אותו, וכל שאר התהליכונים יאלצו לחכות שהוא יסיים.
לאחר מכן תהליכון נוסף יוכל לנעול אותו ולעבוד עליו , וכן הלאה.

```
std::mutexm;  std::string s;
void append_with_manual_lock(std::string const& extra) {
        m.lock();
        s += extra;
        m.unlock();
}
int main() {
        s = "a";
        auto res1 = std::async(append_with_manual_lock,"b");
        auto res2 = std::async(append_with_manual_lock,"c");
        res1.wait(); res2.wait();
        std::cout << s << std::endl;
}
```

בצורה זו רק תהליכון אחד ניגש ל-s בו זמנית. אך עדיין יש בעיה. במידה ויש חריגה,
ניכנס למצב קיפאון ($Deadlock$), כי אחרי השורה $s+= extra$; נקפוץ לסוף הפונקציה,
לא נשחרר את m ולא יהיה מי שישחרר אותו. לכן פתרון אפשרי יהיה להוסיף $try - catch$,
ולעשות $unlock$ ב-$catch$. פתרון אפשרי נוסף (ומומלץ) הוא שימוש ב-$lock\_guard$:

```
void append_with_lock_guard(std::string const& extra) {
        std::lock_guard<std::mutex> lk{m};
        s += extra;
}
```

lk הוא אובייקט מסוג $lock\_guard$ שמקבל אובייקט מסוג $std :: mutex$,
והוא מבצע $lock$ לאורך כל חייו. כלומר מרגע קריאה לבנאי ועד קריאה למפרק.
אם יש שגיאה, תתבצע קריאה למפרק שלו וכך לא נגיע למצב קיפאון.

<u>$Atomic$</u>:

אובייקטים מטיפוס $Atomic$ הם האובייקטים היחידים ב-$C++$ שכתיבה וקריאה מהם על ידי
תהליכונים שונים מוגדרת. כלומר בזמן שתהליכון מסוים ניגש לאובייקט $Atomic$,
יתבצע $lock$ ו-$unlock$ בהתאם , בלי לעשות פעולות נוספות (לא נרחיב בקורס מעבר):

```
#include <atomic>
struct AtomicCounter {
        std::atomic <int> value;
        void increment() { ++value; }
        void decrement() { --value; }
        intget() { return value.load(); }
};
```