חזרה למבחן **מבנה זכרון ושפת C++ – סמסטר ב' 2019 מדעי המחשב**

נכתב ע"י צבי מינץ
מייל: zvimints@gmail.com

מי שנעזר בסיכום מוזמן לתת **endorse** ב-LinkedIn
לא להתבייש לעשות Connect 😉, **תודה**!
https://www.linkedin.com/in/tzvi-mints-0ba18a180/

נכתב ע"י צבי מינץ
מייל: zvimints@gmail.com

הקדמה

	Java	C++	C
תכנות ברמת Low-Level	לא	כן	כן
תכנות ברמת High-Level	כן	כן	לא
תכנות גנרי	מוגבל	כן	לא
קושי	בינוני	קשה	קל
ניהול זיכרון	נשלט ע"י המערכת, תומך בתהליכים ו-Interfaces	ע"י המתכנת, בעזרת מצביעים, תומך בStructs ו-Unions	
הורשה	לא תומך בירשות מרובות	תומך בירושות מרובות	
העמסת אופרטורים	לא	כן	

תוכניות שרשומות ב++C : פייסבוק, בייטקוין, LibreOffice ועוד.

ב++C, הזכרון אשר משומש למבנה נתונים הוא הדוק, זאת אומרת, מה שאנחנו מבקשים זה מה שמקבלים – לעומת זאת בגאווה, מבנה הנתונים יכול לצרוך הרבה יותר זיכרון ממה שהוא צריך באמת.

ב-גאווה ישנו מצביע לטבלה וירטואלית, כך שבמקרה של ירושה, ניתן לדעת איזה להפעיל בזמן ריצה. יש מצביע כזה גם בC++ אך הוא מופעל רק במקרה של ירושה של מתודות וירטואליות.

בנוסף, ב++C יש שימוש רק בקומפיילר, בעוד שבגאווה יש גם קומפיילר וגם מפרש.

מחרוזות: std::string עם #include <iostream>
משתנים בולאנים: bool Parm
:Enums

```
enum class Season {
Winter,
Spring,
Summer,
Autumn
}
Season curr_seasson;  curr_season = Season::AUTUMN;
Curr_season = SUMMER; // Wont Compile!
```

מרחבי שם (Using Namespace)

ב-CPP כל התכונית נמצאת תחת שם אחד, לכן לא נוכל לקרוא לשני משתנים באותו השם מאחר ותהיה שגיאת קומפלצייה, גם אם משתמשים ב-2 ספריות שונות שבפנים יש להם אותו שם משתנה, לא נוכל להשתמש ב-2 הספריות יחד.
לכן בשביל בעיות מהסוג הזה נשתמש ב- using namespace {}, נגדיר שם חדש לקטע וכל פעם שנרצה לפנות לספרייה/משתנה מסויימת נציין תחילה את רחב השם.

לדוגמא :

```
namespace name {
   // code declarations
}
```

קריאה ע"י

```
name::code;  // code could be variable or function.
```

בגלל זה כשאנחנו משתמשים בספרייה iostream אז נציין לפני std:: כי היא תחת namespace std {} **מומלץ** לשים על המחלקות שאנחנו בונים **שם מרחב**.

שגיאות Exceptions :

יש 2 סוגי שגיאות אפשריות:

1. שגיאות בזמן ריצה
2. של המתכנת עצמוד

עבור טיפול מסוג הראשון ניתן לזרוק חריגות באופן הבא:

If (x<0) throw string ("Type Here");

ולאחר מכן אפשר לתפוס את השגיאות ע"י בלוק try-ctach באופן הבא:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

ליצור פונקצייה שזורקת שגיאה:

```
double myfunction (char param) throw (int);
```

אופצייה נוספת:

```
#include <stdexcept>
...
if (x<0)  throw std::out_of_range("x should be at least 0");
```

והתפיסה נראית כך:

```
try {
     func(-5);
} catch (const std::exception& ex) {
     cout << "   caught exception: " << ex.what() << endl;
}
```

עבור שגיאות מהסוג השני:

שגיאות שעלולות להופיע תוך פיתוח הקוד, ניתן להשתמש ב-Assert (בזמן ריצה) לדוגמא `assert(argc == 1);` אם TRUE ממשיך אם FALSE עוצר.

אם רוצים בזמן קופלציה אז ב- **Static_assert (הסבר למטה)**

לדוגמא:

```
#include <iostream>
#include <cassert>

int main() {
    assert(2+2==4);
    std::cout << "Passed" << std::endl;
    assert(2+2==6);
    std::cout << "Failed" << std::endl;
}
```

פלט:

```
tzvimints@Tzvi-Mints:/mnt/c/Users/Tzvi Mints/Desktop/CPP/Linux/Testing$ ./a.out
Passed
a.out: Exception.cpp:8: int main(): Assertion `2+2==6' failed.
Aborted (core dumped)
```

ליצור מחלקת Exception:

```
// Exception Example
#include <iostream>
#include <exception>
#include <string>

void Func(int);
class MyException : public std::exception {

    std::string m_error;
    const char* getMessage() { return "Hey"; }

public:
    // Constructor
    MyException(std::string error)
        : m_error(error)
    {}

    // Destructor
    ~MyException() throw() {}

    virtual const char* what() const throw() { // Virtual Adds Nothing
        return m_error.c_str(); // Transform to char* - C++11 version
    }
};

int main() {
    try {
        Func(-1);
    }
    catch(const std::exception& ex) {
        std::cerr << ex.what() << std::endl;
    }
    return 0;
}

void Func(int num) {
    if( num < 0 ) throw MyException("Invalid Number");
}
```


Bash היא מעטפת פקודה למערכות UNIX

Bash Scripting

בדומה לכל מעטפת פקודה סטנדרטית, גם התחביר של Bash מאפשר יצירה של קובצי אצווה (הוא קובץ המכיל שורות טקסט של פקודות למערכת ההפעלה). קבצים אלה נקראים Bash Scripts ולהם צורה מובנית כפי שמדגים הסקריפט הבא:

```
#!/bin/bash
if [ $# -lt 3 ] ; then
        echo "I expected at least 3 parameters for this script"
        exit 1
fi
#loop over all input arguments and print them in order
for x in $* ; do
        echo "the next argument is ${x}"
done
#loop over all input argument and print them in order
#but also shift them such that the second argument becomes the
first one, etc.
while [ $# -gt 0 ] ; do
        echo "the next argument is ${1}"
        shift
done
exit 0
```

Overloading in C++

מצב שבו יש כמה פונקציות באותו השם אך הפונקציה מקבל ארגומנטים שונים, הבחירה לאיזו פונקציה לקרוא מתבצעת על ידי הקומפיילר (ערך החזרה לא משפיע!)

איך הוא יודע?

1. מציאת כל הפונקציות עם השם המבוקש
2. מתוך התוצאות של (1) הוא מוצא את אלה עם **מס'** הפרמטרים המתאים ביותר
3. מתוך (2) **סוג** הפרמטרים המתאימים ביותר.

Classes in C++ (לשים לב לסגור עם ;)

ההבדל בין Struct עם Class הוא שברירת המחדל היא Public מול Private בהתאמה.

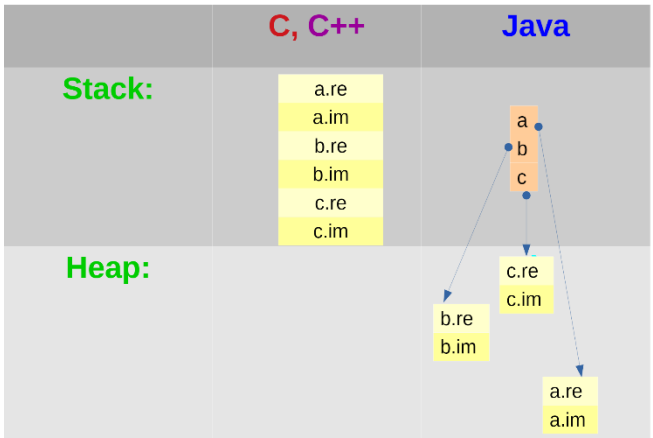
בנוסף:

שגיאת קומפלציה	תקין

`class Derived : Base { };`
`// is equilalent to:`
`class Derived : private Base {}`
 	
`d.x = 20; // compiler error`
`becuase inheritance is private`
	`struct Derived : Base { };`
	`// is equilalent to:`
	`struct Derived : public Base {}`
	
	`d.x = 20; // works fine becuase`
	`inheritance is public`

מבחינת זיכרון:

```
int main () {  Cplx a, b, c;  };
```



נכתב ע"י צבי מינץ
מייל: zvimints@gmail.com

ישנם 2 דרכים **למימש (Inline Vs Outline)** פונקצייה:
דרך ראשון: "Inline", בתוך הקובץ hpp.
דרך שנייה: "Outline", כלומר בקובץ hpp. נרשום Complex (double re, double im);
ובקובץ CPP למימש `Complex::Complex(double re, double im) { Code }`

הערה: ניתן לשים גם Inline על פונקציה לדוגמא': **inline** void foo() {}

כאשר אנחנו קוראים לפונקצייה (לא אנליין) אז תמיד יש **Overhead** כי הקומפיילר צריך לייצר הוראות כמו קריאה לפונקצייה (Call), לדחוף את המשתנים המקומיים למחסנית, לדחוף את ה- Ret Address ועוד, כל פעם שאנחנו קוראים לפונקצייה יש Overhead, ולכן כאשר אנחנו קוראים לפונקצייה הרבה מאוד פעמיים יש הרבה Overhead, ולכן ניתן להשתמש בInline כדי לחסוך זאת, כאשר יש פונקציה שנקראת { Code } inline void foo() אז כל פעם שיש קריאה ל()foo אז במקום זאת, כל ה-Code יועתק במקום הקריאה ל-()foo

הדילמה היא **זכרון** מול **זמן ריצה**.
כל פעם שרוצים להגדיר פונקציית Inline אז צריך לשים אותה ב-Header כי אם היא לא תהיה ב-Header File אז הקומפיילר לא יוכל לראות את הInline Code היות והוא יכול להסתכל על קובץ CPP **אחד** ועל הרבה קבצי HPP .

Header File

בקובץ זה נשים את כל נתונים המחלקה וחתימות הפונקציות.
הקובץ יראה באופן הבא:

Name.hpp	Name.cpp

`#ifndef NAME_H`
`#define NAME_H`
 	
`#include <iostream>`
`...`
`class name {`
 	
`}`
`#endif`
	`#include "Name.hpp"`
	`// Constructor`
	`Name::Name (int a ) : a(a) {}`
	`// Function`
	`<Return Param>`
	`Name::Name::<Function Name> () {`
	`}`

למה צריך קובץ **Header File**?

- להפריד בין מימוש להצהרות
- **קובץ הHeader עונה על השאלה "מה במחלקה" והCPP עונה על "איך".**
- חוסך זמן קומפלצייה כאשר יש קובץ Makefile טוב, היות ואם הכל בקובץ אחד אז כל פעם נצטרך לקמפל מחדש בכל שינוי.
- שומר על הקוד יותר מאורגן

Class Basics – Member / Static

אם יש מחלקה שבתוכה יש משתנה סטטי **לא** ניתן להאתחל אותו בשורה! השורה הבאה גוררת שגיאת קומפלצייה :

```
struct Object {
    static int var = 0;
};
```

נכתב ע"י צבי מינץ
מייל: zvimints@gmail.com

הסיבה לקח שאתחול כזה בקובץ hpp יגרור **לשכפול** האתחול בכל קובץ שקשור אליו.
ולכן חייב לאתחל אותו מחוץ למחלקה:

```
int Object::var = 1000;
```

הערה: קריאה תקינה למשתנה/מתודה סטטית היא באופן הבא:

```
    std::cout << Object::var << std::endl;
```

אולם השורה הבאה גם תתקמפל.

```
    std::cout << instance.var << std::endl;
```

לעומת זאת, ניתן לאתחל Static Const! (**חוקי**)

```
    static const int var = 1;
```

ניתן גם לאתחל עם Inline מ - C++ 12

```
    inline static int x = 0;
```

בנוסף לא ניתן להחזיר משתנה סטטי באמצעות **this->** כי הוא לא עצם של המחלקה.
לא ניתן לאתחל משתנה Const static ברשימת אתחול.

בנאים – Constructors

- ניתן לממש בקובץ הכותרת או בקובץ המימוש.
- יש בנאי ללא פרמטרים דפולטיבי **אם"ם** לא מגדירים בנאי אחר למחלקה.
- בנאי ברירת מחדל של המחלקה לא עושה כלום – הוא **לא** מאתחל את הזיכרון (בניגוד לגאווה) ולכן הערכים לא מוגדרים – יתכן שיש שם ערכי זבל.

מפרקים - Destructors

המתכנת אף פעם לא צריך לקרוא למפרק באופן ידני, זוהי האחראיות של הקומפיילר להכניס קריאה למפרק.
זה קורה כאשר העוצר נוצר על המחסנית ואז יש יציאה מהסקופ המתאים, או אם העצם נוצר על הערמה בעזרת new ואז צריך למחוק אותו ע"י delete.

אופרטור **new** – הקצאת זיכרון עבור עצם חדש למחלקה **וקריאה** לבנאי המתאים של המחלקה.
מחזיר 0 בשגיאה!
אופרטור **delete** – מבצע קריאה למפרק של המחלקה **ושחרור** הזיכרון שהוקצה עבור העצם.
אופרטור **new[]** – הקצאת זיכרון עבור מערך של עצמים מהמחלקה וקריאה לבנאי ברירת המחדל של כל אחד מהעצמים.
אופרטור **delete[]** – מבצע קריאה למפרק של כל אחד מהערכים במערך ומשחרר את הזיכרון שהוקצה עבור המערך.
כשמתאחלים מערך ע"י new[] חייב לשחרר אותו ע"י delete[] אם
נשתמש בdelete במקום ב-delete[]. (הקוד יתקמפל ויקר מערך)
ולכן תהיה דליפת זכרון.
דוגמא:
מה מוחקים?
כל דבר עם **new**.

```
class MyClass
{
public:
    MyClass();   // constructor
    ~MyClass(); // destructor
private:
    char* _mem;
};
MyClass::MyClass()
{
    _mem = new char[1000];
}
MyClass::~MyClass()
{
    delete[] _mem;
}
```

Reference

כשיוצרים רפרנס יש צורך **לאתחל אותו מייד (לא ב-null)**, זה אמור לצמצם את הסיכוי לשגיאות היות וכשיש רפרנס אנחנו יכולים להיות בוטחים שיש עצם ממשי ולא null.

בנוסף **לא ניתן לשנות** רפרפנס לאחר יצירות (מאותה סיבה – צמצום שגיאות).
<u>הערה:</u> רפרנס לא יכול להיות מטיפוס void – שגיאת קומפלציה.

```
int a = 10;
int *p = &a;
int &r = a;

 a = 10
 &a = 0x7ffff1551ed4
 p = 0x7ffff1551ed4
 *p = 10
 &p = 0x7ffff1551ed8
 r = 10
 &r = 0x7ffff1551ed4
```

נהוג לרשום **&** צמוד לשם כדי לדעת שהוא רפרנס. (כנל לגביי פוינטר)
השימוש העיקרי של רפרפנס הוא כשרוצים לכתוב פונקציות שמשנות את הארגומנטים שלהם.

Parameter passing

By value	By reference	By const reference
void f (Point x) {…}	void f (Point& x) {…}	void f (const Point& x) {…}
x is copied	x is not copied	x is not copied
Compiler lets f modify x, but changes have no effect outside	f can modify x	compiler does not let f modify x

- בקבלת עצמים גדולים עדיף לקבל אותם כרפרנס כי אז אין העתקה
- **<u>לא להחזיר רפרנס למשתנה לוקאלי אשר נהרס בסוף הפונקציה.</u>**
- ניתן להחזיר רפרנס של this* על מנת Call Chaining (שרשור פעולות)

<u>Lvalue & rvalue</u>
Lvalue (שמאל) – מציין דבר שיש לו כתובת בזיכרון - יש לו location)
Rvalue (ימין) – מציין דבר שלא ממשיך להתקיים אחרי שהביטוי מסתיים.

<u>לא</u> ניתן לאתחל reference עם משהו שהוא לא lvalue, כי רפרנס אמור להצביע לביטוי לא זמני.
ולכן הקוד הבא **<u>לא</u>** מתקמפל:

```
#include <iostream>
using namespace std;
const string foo() { return string(); }
int main() {
    const string a = foo();
    string b = foo(); // Will make core dump
    const string& c = foo(); // Will make core dump
    string& d = foo(); // Compile error [R Value Initialize]
```

```
    const string& d = "HEY"; // Will make core dump
}
```

אך, יש **מקרה יוצא מהכלל שאפשר להציב rvalue בתוך רפרנס**, וזה כמגדירים אותו כקבוע. ולכן הביטוי הבא חוקי: (פלט 9)

```
    int x = 3, y=6;
    const int &ref = x + y;
    cout << ref << endl;
```

וזה קורה בגלל שהדבר שמצביעים אליו מוגדר כ"קבוע", ולכן הקומפיילר ממילא לא ייתן לנו לשנות אותו ולכן הרפרנס יכול להצביע גם לדבר שאין לו כתובת בזכרון, במקרה זה, הקומפיילר **מאריך** את החיים של המשתנה הזמני, כך שישאר בזיכרון (בד"כ על המחסנית) למשך כל אורך החיים של הרפרנס.

החזרת רפרנס בפונקציה – למה?

- שיטה של מחלקה יכולה להחזיר רפרנס לשדה של המחלקה כדי לאפשר לקוראים לשנות את המחלקה.
- כדי לאפשר לקרוא לכמה שיטות-עדכון בשרשרת.

Composition – הרכבה

שימוש במחלקה עם אובייקט ממחלקה אחרת.
כאשר מחלקה מכילה מחלקה אחרת, האובייקט המוכל נולד לפני האובייקט המכיל.
סדר הריסת האובייקטים הפוך לסדר היצירה : קודם נהרס האובייקט המכיל ואז האובייקט המוכל.
באופן הבא:

```
In A
In B
In ~B
In ~A
```

דוגמא לחוסר בנאי ברירת מחדל (שגיאת קומפלציה) :

```
class A {
    int x;
    public:
    A(int x) : x(x) { cout << "In A" << endl; }
    ~A() { cout << "In ~A" << endl; }
};


class B {
    A a;
    public:
    B() { cout << "In B" << endl; } // Compiler Error!
                                    // Must call B() : a(0) {}
                                    // Because No Default Constructor!
    B(int x) : a(x) { cout << "In B" << endl; }
    ~B() { cout << "In ~B" << endl; }
};

int main() {
    B b(5);
    return 0;
}
```

רשימת אתחול – Initialization List

```
Clock::Clock(int hours, int minutes) : hours(hours), minutes(minutes)
{
    //hours = hours;
```

מחוץ לסוגריים: תכונה

בתוך הסוגריים: פרמטר

בנאי ברירת מחדל של מחלקה מורכבת קורא לבנאי ברירת המחדל של כל אחד מהרכיבים.
במידה ויש אובייקט שיש בו כמה אובייקטים מוכלים וכולם מאותחלים בשורת האתחול, סדר אתחולם יהיה לפי סדר הגדרתם במחלקה ולמי לפי סדר אתחולם בשורת האתחול.

דוגמא:

```
    A a1;
    A a2;
    B(int a1, int a2) : a2(a2), a1(a1) {}
```

הסדר יהיה קודם אתחול של a1 ואז אתחול של a2.

יתרונות:

- ניתן לרשום אותם שמות גם למשתני עצם וגם לפרמטרים שמקבלים (בלי שימוש בthis)
- חובה להשתמש ברשימת אתחול אם יש משתני מחלקה **קבועים** / **רפרנסים**.
- **מהיר יותר** – חוסך את האתחול ע"י בנאי ברירת מחדל.
- בטוח יותר – כי הוא מבטיח שבתוך הבנאי כל הרכיבים כבר בנויים עם פרמטרים נכונים.

העמסת אופרטורים:

סוגי אופרטורים: חשבוניים,קלט/פלט,סוגריים מרובים,סוגריים עגולים, השמה.

דוגמאות:

```
struct X {
    // Prefix
    X& operator++() {
        // Do Increment
        return *this;
    }
    // Postfix
    // You cannot return a reference to that local object
    // and hence it needs to be returned by value
    X operator++(int) {
        X temp(*this); // Copy
        opeartor++(); // Increment
        return tmp;
    }
}
```

```
// Implementation of [] operator.  This function must return a
// reference as array element can be put on left side
int &Array::operator[](int index)
{
    if (index >= size)
    {
```

```
        cout << "Array index out of bound, exiting";
        exit(0);
    }
    return ptr[index];
}
void Matrix::operator()()
{
    // reset all elements of the matrix to 0.0
    for (int row=0; row < 4; ++row)
        for (int col=0; col < 4; ++col)
            data[row][col] = 0.0;
}
    matrix(); // erase matrix
```

כי הם מחזירים *this

```
// arithmetic operators
const PhysicalNumber operator-() const; //[V]
const PhysicalNumber operator+() const; //[V]
const PhysicalNumber operator+(const PhysicalNumber&) const; //[V]
const PhysicalNumber operator-(const PhysicalNumber&) const; //[V]
PhysicalNumber& operator+=(const PhysicalNumber&);  //[V]
PhysicalNumber& operator-=(const PhysicalNumber&); //[V]
PhysicalNumber& operator=(const PhysicalNumber&); //[V]

// 6 comparison operators
friend bool operator==(const PhysicalNumber&, const PhysicalNumber&);
friend bool operator<(const PhysicalNumber&, const PhysicalNumber&);
friend bool operator!=(const PhysicalNumber&, const PhysicalNumber&);
friend bool operator<=(const PhysicalNumber&,const PhysicalNumber&);
friend bool operator>=(const PhysicalNumber&,const PhysicalNumber&);
friend bool operator>(const PhysicalNumber&,const PhysicalNumber&);

// Postfix: (A--)
const PhysicalNumber operator++(int); //[V]
const PhysicalNumber operator--(int); //[V]
// Prefix: (--A)
PhysicalNumber& operator++(); //[V]
PhysicalNumber& operator--(); //[V]
// I/O
friend std::ostream& operator<<(std::ostream&, const PhysicalNumber&);
friend std::istream& operator>>(std::istream&, PhysicalNumber&);

// Bonus:
PhysicalNumber& operator/(const PhysicalNumber&);
PhysicalNumber& operator* (const PhysicalNumber&);
PhysicalNumber& operator*=(const PhysicalNumber&);
PhysicalNumber& operator/=(const PhysicalNumber&);
```

וקריאה מהקובץ CPP באופן הבא:

```
const PhysicalNumber PhysicalNumber::operator-()
```

דוגמא לאופרטור קלט – Istream:

```
istream& operator>>(istream& input, Complex& number)
{
   input.ignore(); // ignore the first (
   input >> number.real;
   input.ignore(); // ignore the ,
   input >> number.imaginary;
   input.ignore(); // ignore the next )
   return input;
}
```

הערות:

`friend const Point operator++(Point & point, int dummy);`	`pt++ -> operator++(pt, 0)`	non-member friend
`const Point operator++(int dummy)`	`pt++ -> pt.operator++(0)`	member function
`bool operator==(const Point & rhs) const;`	translates "`p1 == p2`" to "`p1.operator==(p2)`"	*member function*
`friend bool operator==(const Point & lhs, const Point & rhs);`	`p1 == p2`" to "`operator==(p1, p2)`"	*non-member function*
`cout << p1 << endl;`	Translates to `operator>>(cin, p1);`	

prefix function **returns a reference** to this instance
postfix function returns a **const** object by value. A const value cannot be used as lvalue.

מצד שמאל מספר:

```
    friend Complex operator+(const int& real, const Complex& b);
```

```
Complex operator+(const int& real, const Complex& b) {
    return Complex(real + b.real, b.imaginary);
}
```

Const

קוראים מימין לשמאל!

חייב להיות מאותחל באותה שורה.

```
    // Error!
    int& const a = 1; // const reference to int (Some compilers warn
about it, other emit errors.)

// Both are same:
const int& d = y;
```

```
int const& b = x; // reference to const int (Wont change with reference
but can change x)
// b = 6; // Compile Error!
// x = 6; // Works!
// But
int& b2 = x;
b2 = 6; // Valid !

    // Error!
    const& int c = 3; // integer to a reference const (illegal)
```

- `int* const` is a `const` pointer to a [non-`const`] `int`
- `int const*` is a [non-`const`] pointer to a `const int`
- `int const* const` is a `const` pointer to a `const int`

```
    const A ref; // Call Default / Empty Constructor
    const int a; // Compiler Error
```

Int* **const** p1 = &a;

~~p1 = &b~~

Const int* p1 = &a

```
int const* p3; ⇕
```

~~*p1 = 5;~~

לעומת זאת, 2 הביטויים הבאים שקולים:

```
A const& a = f1;
const A& a2 = f2;
```

<u>שיטות קבועות</u>

כשפונקציה מוגדרת כקבועה, היא יכולה להקרא מכל סוג של אובייקט, **בעוד שאם פונקציה מוגדרת לא-קבועה אז רק משתנים לא-קבועים יכולים לקרוא לה.**

זוהי שגיאת **קומפלצייה**:

```
class TestConst {
    public:
    void foo() { cout << "Foo Const" << endl; }
};
int main() {
    const TestConst t;
    t.foo();
}
```

בעוד שהקוד הבא יתקמפל:

```
class TestConst {
    public:
    void foo()  { cout << "Foo" << endl; }
    // const void foo()  { cout << " Const Foo" << endl; }
    void foo() const { cout << "Foo Const" << endl; }
    // const void foo() const { cout << "Const Foo Const" << endl; }
};
int main() {
    TestConst a;
    const TestConst b;
```

```
tzvimints@Tzvi-M
Foo
Foo Const
```

```
    a.foo();
    b.foo();
    return 0;
}
```

הערה: כשפונקיה היא const אז הקומפיילר לא יתן לנו לקרוא מתוכה לשיטות אחרות שהן לא const.
לדוגמא:

```
class TestConst {
    public:
    void boo()  { cout << "Boo" << endl; }
    void foo() const { boo(); }
};
int main() {
    const TestConst b;
    b.foo();
}
```

הערה: כשמוסיפים את המילה const לשיטה היא הופכת לחלק מה"חתימה" של השיטה. מכאן אפשר ליצור שתי שיטות שונות עם אותו שם ואותו פרמטרים – אחת עם קבוע ואחת בלי, והקומפיילר ישתמש בשיטה הנכונה לפי ההקשר – אם משתמשים בשיטה lvalue הוא יקרא לשיטה בלי const ואם משתנים בשיטה כ-rvalue הוא יקרא לשיטה עם const.
זה שימושי במבנה נתונים כמו vector שנהוג להגיד 2 שיטות של get(i), אחת מיועדת לקריאה והיא מוגדרת כconst ומחזירה const&, והשנייה מיועדת לכתיבה – היא מוגדרת בלי const ומחזירה &.

למה זה חשוב?

- **עוזר לאתר באגים ותקלות.** אם שיטה מסויימת מוגדר כ-const, אפשר להיות בטוחים שהיא לא משנה את העצם ולכן אם העצם משתנה כנראה התקלה במקום אחר.
- אם בתוכנית הראשית מגדירים עצם כ-const, אפשר לקרוא על העצם הזה רק לשיטות שהוגדרו כ-const
- מאפשר לנו לקבל אזהרות על תופעות-לוואי לא רצויות (לדוגמא סוגריים מרובעים של map עלול לשנות את העצם!) ולכן אם מגדירים את השיטה כconst הקומפיילר יזהה את הבעיה ויזהיר אותנו.

Friend (מחלקה/פונקצייה)

לפעמיים יש פונקציה שהיא קשורה באופן הדוק למחלקה, אבל אי אפשר להגדיר בתוך המחלקה.
לדוגמה, אופרטור >> ו-<<.
בנוסף לפעמיים יש פונקציה שאפשר להגדיר בתוך המחלקה אבל נוח יותר להגדיר בחוץ, לדוגמא האופרטור +, במקום Complex operator+(Complex a,Complex b) אז:
friend Complex operator+(Complex a,Complex b)

מחלקה חברה יכולה לגשת לשדות פרטיים ושמורים של המחלקה שמוגדרת שהיא חברה שלה,
דוגמא היא שימוש בLinkedList שתהיה מחלקה חברה של-Node על מנת לגשת לשדות פרטיים.

```
class Node
{
Node() : key(10) {}
friend class LinkedList;
public:
  int key; // Will Be Junk without constructor
};
class LinkedList {
```

```
    Node head;
    public:
int getHeadData() { return head.key; } // If wasnt friend compile error
};
```

פונקצייה חברה נותנת גישה לשדות הפרטיים והשמורים של המחלקה, פונקצייה חברה יכולה להיות מתודה של מחלקה אחרת, או פונקצייה גלובלית.

```
class Node
{
    int key;
public:
    Node() : key(0) {}
            // ---------- Option 1 -> Inline ---------- //
   friend std::ostream& operator<<(std::ostream &os,const Node& other){
        return os << other.key;
   }
};


int main() {
    Node n;
    std::cout << n << std::endl;
}
```

```
class Node
{
    int key;
public:
    Node() : key(0) {}
        // ---------- Option 2 -> Outside of class ---------- //
    friend std::ostream& operator<<(std::ostream &os,const Node&
other);
};
// Note: Same Signature but without friend!!
std::ostream& operator<<(std::ostream& os,const Node& other) {
    return os << other.key;
}

int main() {
    Node n;
    std::cout << n << std::endl;
}
```

הערות:

- אם מחלקה A חברה של מחלקה B אז זה לא אומר שמחלקה B חברה של מחלקה A
- חברות לא עוברת בירושה
- עדיף שימוש מצומצם (לא רוצים לתת גישה לשדות פרטיים)

בנאי מעתיק

בנאי מעתיק למחלקה T הוא בנאי המקבל פרמטר אחד בלבד והוא const T&
הקומפיילר קורא לבנאי זה אוטומטית בכל פעם שצריך להעתיק עצם מהסוג T לעצם חדש, למשל, במעבר פרמטרים לפונקציות.

למה הוא const &T ולא פשוט T: כדי להעביר פרמטר מסוג T, הקומפיילר צריך לקרוא לבנאי מעתיק, אבל אנחנו מגדירים את הבנאי הזה עכשיו.

אופרטור השמה (=)
מגדיר איך להעתיק עצמים. חשוב במיוחד להגדיר אותו נכון כשהעצמים הם "עמוקים" (כוללים הקצאות דינמיות).
במקרה זה נרצה להגדיר שלוש שיטות:
1. בנאי מעתיק
2. מפרק
3. אופרטור =

כלל האמצע אומר שאם צריך אחד אז צריך את כולם.
בנאי מעתיק לעומת אופרטור השמה
העתקת עצמים מתבצעת בארבעת המקרים הבאים:

בנאי מעתיק ולא אופרטור השמה!

1. כשמגדירים עצם חדש מתוך עצם קיים, לדוגמא Complex b = a
2. כמעבירים פרמטר לפונקצייה by value
3. כשמחזירים ערך מפונקצייה by value
4. בפעולת השמה של עצם קיים לעצם קיים אחר (Complex a; a=b)

הקומפיילר מספק לנו copy c'tor במתנה אשר מבצע "**העתקה רדודה**": מעתיק תכונה-תכונה
העתקה רדודה Shallow Copy
2 מצביעים מכילים את אותה הכתובת, ואז יש תלות בין האובייקטים

המימוש המתקבל במתנה

```
}
Person(const Person& other)
{
    id = other.id;
    name = other.name;
}
```

הבעיה מתחילה להיות כשיש פוינטרים או הקצאות על הערמה, ואז שינוי בעצם אחד יגרור לשינוי בעצם האחר.
במקרה שלנו char* name.
ולכן יש צורך לדרוס את בנאי המעתיק ולממש מחדש **באופן הבא**:

```
Person(const Person& other)
{
    name = new char[strlen(other.name)+1];
    strcpy(name, other.name)
    id = other.id;
}
```

יתכן ונרצה למנוע מעבר בבנאי המעתיק
למשל: למנוע שיבוט בני-אדם
במקרה זה, נצהיר על בנאי המעתיק כפרטי ולא נממש או לחלופין להצהיר ולרשום:
Person(const Person& other) = **delete**;

```
B() b1,b2; // Constructor
B() b3 = b1; // Copy Constructor
```

```
    b1 = b2; // Operator (=)
    void func(B b); // Copy Constructor
    void fun(B& b); // Nothing!
```

שימוש נכון בבנאי מעתיק ואופרטור = :

בנאי מעתיק

```
    Stack(const Stack& other) {
        this->arr = new int[other.size];
        if(this->arr == 0)
// Allocation fails.
            throw("out of memory");
        else
        {
            size = other.size;
            for(int i=0; i<other.index; i++)
            {
             this->arr[i] == other.array[i];
            }
        }
    }
```

אופרטור =

```
Stack& Stack::opeartor=(const Stack& other) {
        if(this == &other) return *this;
        if(other.size != this->size) {
            delete[] this->arr;
            this>arr= new int[other.size];
            this->size = other.size;
        }
        for(int i=0;i<other.index;i++) {
            this->arr[i] = other.arr[i];
        }
        this->index = other.index;
        return *this;
    }
```

<u>התנהגות בלתי צפויה:</u>

```
class Vector {
    public:
    Vector(size_t num) {};
};
int sum(const Vector& v) {}

int main() {
    int i = 3;
    sum(i);
}
```

הקוד מקמפל! (אם נוסיף בבנאי Excplict זה לא יתקמפל)

<u>הערה:</u> כאשר קוראים לבנאי מעתיק אז סדר פעולות הוא קודם כל ללכת למחלקת הבסיס ולאתחל, לאמר מכן לאתחל את משתני העצם ואז להשתמש בCopy-Constructor.

לדוגמא:

```
struct A {
    A() { std::cout << "A()" << std::endl; }
};

struct B : public A {
    A a;
    B() { std::cout << "B()" << std::endl; }
    B(const B& other) { std::cout << "B(&)" << std::endl; }

};
```

```
int main() {
    B b;
    return 0;
}
```

פלט הינו:

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

Explicit

נסתכל על הדוגמא הבאה:

```
class String {
public:
   String(int n); // allocate n bytes to the String object
   String(const char *p); // initializes object with char *p
};
```

ונניח שנרצה לבצע את השורה הבאה:

```
String mystring = 'x';
```

אז מה שיקרה, זה המרה של x ל-int. במידה ולא נרצה שתהיה המרה אוטומטית (Implict) אז נשנה את הקוד ל:

```
class String {
public:
   explicit String (int n); //allocate n bytes
   String(const char *p); // initialize sobject with string p
};
```

ירושה

לעומת גאווה, ב-C++ ניתן לרשת מכמה מחלקות!
למה משתמשים בירושה?

1. כדי שהתוכנית שלנו תשקף את המציאות – כלל אצבע: "אם A **הוא** B" אז A יורש מB
2. כדי להשיג פולימורפיזם בזמן ריצה

הערה: הבנאים לא עוברים בירושה.
באופן כללי: אם מחלקה B יורשת ממחלקה A אז תהיה שגיאת קומפלצייה אם A לא יקבל אתחול, זאת אומרת שאם אין קריאה מפורשת יש נסיון גישה לבנאי דפולטיבי, במידה ואין, נקבל שגיאה.
דוגמאות לשגיאות קומפלצייה:

```
class A {
    public:
    A(int x) { std::cout << "In A Constructor \n"; }
};

class B  {
    public:
    A a;
    B() { std::cout << "In B Constructor \n"; }
};
```

```
class A {
    public:
    A(int x) { std::cout << "In A Constructor \n"; }
};
```

```
class B : public A {
    public:
    B() { std::cout << "In B Constructor \n"; }
};
```

לעומת זאת, הקוד הבא **מתקמפל**:

```
class A {
    public:
    A(int x) { std::cout << "In A Constructor \n"; }
};

class B : public A {
    public:
    B() : A(3), Other Initialize { std::cout << "In B Constructor \n";
}
};
```

על מנת למנוע שגיאות:
אפשרות א': לקרוא בפירוש לבנארי של המחלקה המורישה ברשימת האיתחול.
אפשרות ב': לא לקרוא ואז הקומפיילר יקרא אוטומטית לבנאי בלי פרמטרים אם קיים.

המפרק גם כן **לא** עובר בירושים – כשעצם מהמחלקה Programmer מתפרק, הקומפיילר מפרק אותו ואחר כך קורא למפרק של Person באופן אוטומטי.
אופרטור השמה – **עובר** בירושה, אבל אי אפשר להשתמש בו במחלקה המורישה כי הקומפיילר יוצר אוטומטית אופטרטור השמה חדש שמסתיר אותו.
לדוגמא:
אם במחלקה A:

```
A& operator=(const A& other) {
     std::cout << "In A (=) Oper." << std::endl; }
```

וב-B אין אופרטור שווה, אז בשורות האחרונה תתבצע קריאה לאופרטור = של מחלקת הבסיס (A).

```
    B b1,b2;
    b1 = b2;
```

הערה: אם B יורש את A אבל בA יש בנאי דפולטיבי ריק ואין קריאה מפורשת מהבנאים של B אז תהיה שגיאת קומפלציה!
סוגי הורשה:

```
class Programmer : public Person
class Programmer : protected Person
class Programmer : private Person    // this is the default
```

אם מחלקה B יורשת את מחלקה A על **Public** אז הכל נשמר
אם מחלקה B יורשת את מחלקה A על **Protected** אז כל משתנה שהוא יותר מProtected נהפך להיות Protected (זה אומר שאם קוראים עכשיו לפונקצייה מהמיין, לא ניתן לגשת אליה!!)
אם מחלקה B יורשת את מחלקה A על **Private** אז כל משתנה שהוא יותר מPrivate נהפך להיות Private
הערה: אם משתנה הוא Private אז רק למחלקה הנוכחית יש גישה אליו!

דריסה (Override) :
תזכורת:
Overloading - שתי פונקציות עם אותו שם אבל עם ארוגמנטים שונים, כמות ארגומנטים שונה.

Override - לדרוס מתודה אשר קיימת במחלקה המורישה, אם נרשום מתודה עם חתימה זהה לחתימה של המתודה אשר נמצאת במחלקת האב אז המתודה הזאת תדרוס את התוכן של המתודה במחלקת האב

בעיה בדריסה:

```
struct A {
    void foo() { LOG("A"); }
};

struct B : public A {
        void foo() { LOG("B"); }
};

int main(){
    A *a = new B();
    a->foo();
    delete a;
}
```

פלט: A (התנהות לא מצופה)
בעוד שאם נוסיף את המילה Virtual (בהמשך) נקבל כמו פלט B (כמצופה)
זה קורה כי הפויינטר לא יודע שהוא מצביע על B (יצרנו*A) ולכן נקבל פלט A.

אם אין העמסה אז ניתן לגשת לפוונקציה במחלקת הבסיס דרך האב לדוגמא B.foo() כאשר foo() מוגדרת במחלקת הבסיס.

אם יש חתימה X ועשינו **העמסה <u>איבדנו</u>** לגמריי את הפונקצייה במחלקת הבסיס (**גם אם היא ורטואלית!**) גם אם הארגומנטים שונים (!), כלומר אם ב A יש פונקציה X וב B **<u>אין</u>** העמסה/דריסהאז אפשר לקרוא דרך B לפונקיה X ואז A תקרא, אם ב B יש העמסה, אז אם **<u>אין</u>** פרמטרים תואמים אז תהיה **<u>שגיאת קומפלציה</u>**.

```
class A {
    public:
    A() { std::cout << "In A Constructor \n"; }
    void foo() { std::cout << "A foo" << std::endl; }
    void foo(int) { std::cout << "A foo(int)" << std::endl; }
    void foo(std::string) { std::cout << "A foo(String)" << std::endl;
}
};
class B : public A {
    public:
    B() { std::cout << "In B Constructor \n"; }
    void foo() { std::cout << "B foo" << std::endl; }
    void foo(double) { std::cout << "B foo(double)" << std::endl; }
};
int main() {
    B b;
    b.foo(3);
    b.foo(3.3);
    b.foo();
    b.foo("Hey"); // Compile Error!
```

```
}
```

על מנת לקרוא לפונקצייה במחלקת הבסיס אז נוכל מהמחלקה המורישה להשתמש בפקודה:

```
A::foo();
```

באופן כללי, ניתן לדרוס רק משהו שהוא וירטואלי.

בנאי מהעתיק בהורשה:

כאשר B יורש מA אז הCopy Constructor שמקבלים במתנה עובר ראשית Copy Constructor של הבסיס, אם נממש בעצמו את הCopy Constructor של היורש יש צורך לקרוא בשורת אתחול לבנאי כלשהו של הבסיס, לרוב, הCopy Constructor באופן הבא:

```
    B(const B& other) : A(other) {
     std::cout << "In B COPY" << std::endl;
    }
```

במידה **ולא** נקרא לבנאי אז יהיה נסיון לפנות לDefault Constructor ותתקבל שגיאת קומפלצייה.

```
void foo(A a) {
    std::cout << "Fooooo" << std::endl;
}

int main() {
    B b;
    foo(b);
    // will Output:
    In A Constructor
    In B Constructor
    In A COPY
    Fooooo
}
```

בעוד שאם במקום A a נרשום const A& a **לא** נקרא לבנאי מעתיק.

בעיית הורשה מרובה (פתרון – הורשה וירטואלית) :

```
//      A
// B         C
//      D
class A {
    void show() {}
};
class B : public A { // To Fix Change to virtual public
};
class C : public A { // To Fix change to virtual public
};
class D : public C, public B {
};
int main() {
    D d;
    d.show() // Compiler Error!
}
```

error: request for member ‘show’ is **ambiguous**

הפתרון הינו:

```
class B : public virtual A {
};
class C : virtual public A {
};
```

Virtual

כללים:

פונקצייה רגילה – לא לדרוס

פונקצייה וירטואלית – לדרוס ולהוסיף בחתימה Override באופן הבא:

```
    void foo() override { }
```

וירטואלית טהורה - **חייבת** להדרס, לא ניתן לייצר מאותה מחלקה אובייקטים (נקראת אבסטרקטית אבל פוינטרים מותר)

לדוגמא:

```
struct A {
    virtual void foo() = 0;
};
struct B : public A {
    void foo() override {} // without it - compiler error
};
int main(){
    A *a = new B();
    A a; // Compile Error
    a->foo(); // Will go to B::foo
    delete a;
}
```

בשאר השפות כל השיטות הן וירטואליות, משמע אין זכות בחירה כמו פה (שכן זהו בזבוז זמן ומקום)

בנאי וירטואלי: (הבדל בפויינטרים)

```
struct Base { ~Base() { cout << "~Base" << endl; } };
struct Der : public Base { ~Der() { cout << "~Der" << endl; }};
int main()
{
    Base* base = new Der;
    delete base;
    cout << endl;
    // Will Print : (First ~Der!)
    // ~Der
    // ~Base
    Base base2; // Will Print ~Base

    Der der; // Will Print ~Der ~Base

    IF BASE WASNT VIRTUAL THE OUTPUTS WAS:

    Base* base = new Der;
    delete base;
    cout << endl;
```

```
    // Will Print :
    // ~Base
    Base base2; // Will Print ~Base
    Der der; // Will Print ~Der ~Base
    return 0;
}
```

פולימורפיזם (המשך לוירטואל)

המושג פולימורפיזם

רב צורתיות, הרעיון הכללי הוא להתייחס לעצמים שונים בתור דברים דומים.
רב צורתיות מאפשר לנו לבצע פעולות מסוימות מבלי קשר ישיר לאובייקט עליו אנחנו מבצעים את הפעולה, כך שגם אם נשנה את האובייקט לאובייקט אחר זה לא ישנה את הדרך בה אנחנו מבצעים את הפעולה שלנו על האובייקט החדש, לדוגמא מערך של חתול,כלב,ציפור כאשר כל ממש את הממשק חיה, ניתן ליצור מערך של חיות וכל פעם להפעיל את המתודה $X()$, תוצאת המתודה תיהיה כתלות האובייקט עליו אנחנו מפעילים את הפונקצייה.

מבסיס: (לא טוב)

```
struct A {
    virtual void foo() { LOG("A"); }
};
struct B : public A {
    void foo() { LOG("B"); }
};
struct C : public A {
    void foo() { LOG("C"); }
};

int main(){
    A list[2];
    list[0] = B();
    list[1] = C();
    for(int i=0; i<2; i++)
        list[i].foo();
}
```

פלט: A A

אם נשנה ל:

```
    virtual void foo() = 0;
```

נקבל שגיאת קומפלצייה.

```
struct A {
    void foo() { LOG("A"); }
};
struct B : public A {
    void foo() { LOG("B"); }
};
struct C : public A {
    void foo() { LOG("C"); }
};

int main(){
    A list[2];
    list[0] = B();
    list[1] = C();
    for(int i=0; i<2; i++)
        list[i].foo();
}
```

פלט: A A

```
struct A {
    virtual void foo() = 0;
    // Or:
    // virtual void foo() { LOG("A"); }

};
struct B : public A {
    void foo() override { LOG("B"); }
};
struct C : public A {
    void foo() override { LOG("C"); }
```

```
struct A {
    void foo() { LOG("A"); }
};
struct B : public A {
    void foo() { LOG("B"); }
};
struct C : public A {
    void foo() { LOG("C"); }
};

int main(){
```

```
};

int main(){
    B b;
    C c;
    A* list[2];
    list[0] = &b;
    list[1] = &c;
    for(int i=0; i<2; i++)
        list[i]->foo();
}
```

פלט: B,C!

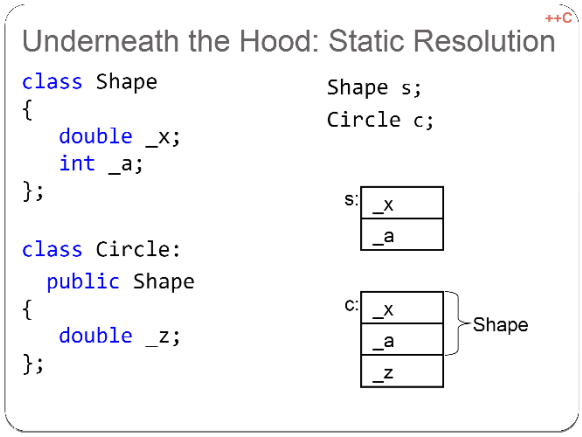
```
    B b;
    C c;
    A* list[2];
    list[0] = &b;
    list[1] = &c;
    for(int i=0; i<2; i++)
        list[i]->foo();
}
```

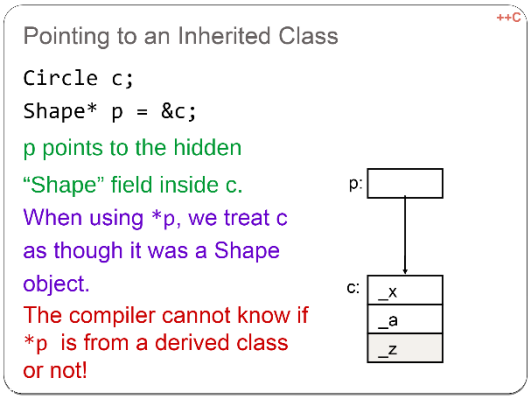
פלט: A A

Static Resolution:

נסתכל על הדוגמא הבאה:

- ידוע בזמן קומפלצייה



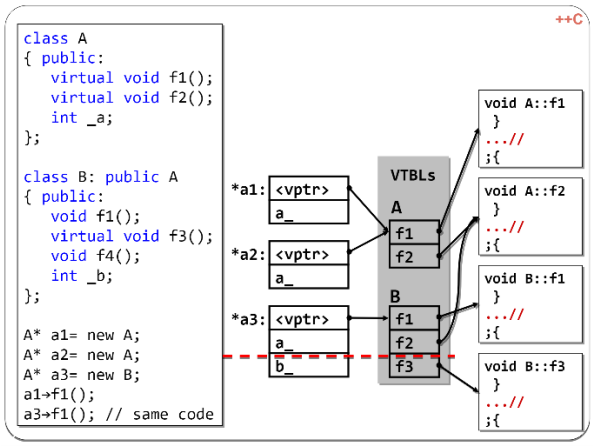


Dynamic Resolution:

- בזמן ריצה – נעזר ב V-Table כדי לדעת לאן ללכת.

איך זה קורה? בעזרת Virtual Table

- **כל אובייקט מכיל בתוכו פיינטר יחיד למערך של פונקציות**
- **מערך של פונקציות מכיל את המתודות המתאימות**
- **כל בכל טבלה של מחלקה בודקים איזה פונקציות הם Override וכך נשנה את הפוינטרים בהתאם.**



התנהגות של פוינטרים ורמפרנסים (הערה: אין שימוש בבנאי מעתיק):

```
struct A {
    virtual void foo() { LOG("A"); }
};
struct B : public A {
    void foo() { LOG("B"); }
};
int main(){
// B *b = new A; // Compile Error, B have more memory
A *a1 = new B;
B b;
A &a2 = b;
b.foo(); // B:FOO
a1->foo(); // if virtual B::foo, else, A:foo
a2.foo(); // if virtual B::foo, else, A:foo
}
```

בעיית המפרק

נסתכל על הקוד הבא ונראה שיש זליגת זכרון:

```
struct A {
 A() { LOG("A Constructor"); }
 ~A() { LOG("~A Destructor"); }
};
struct B : public A {
 B() { LOG("B Constructor"); }
 ~B() { LOG("~B Destructor"); }
};
int main(){
// B b; // Works Good!
A *a = new B;
delete a;
}
```

פלט:

```
A Constructor
B Constructor
~A Destructor
```

פתרון: להגדיר את המפרק כ-Virtual.

```
struct A {
 A() { LOG("A Constructor"); }
 virtual ~A() { LOG("~A Destructor"); }
};
struct B : public A {
 B() { LOG("B Constructor"); }
 ~B() { LOG("~B Destructor"); }
};
int main(){
// B b; // Works Good!
A *a = new B;
```

```
delete a;
}
```

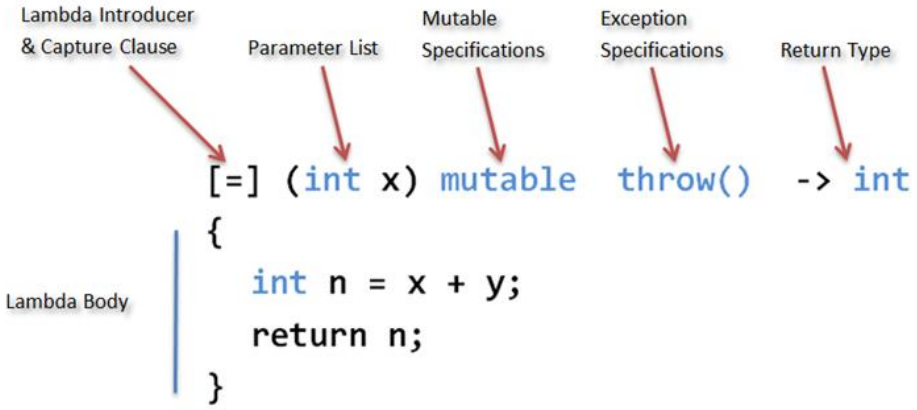
פלט:

```
A Constructor
B Constructor
~B Destructor
~A Destructor
```

לסיכום - כללי אצבע:

1. אם יש פונקציות וירטואליות במחלקה, הגדר את המפרק גם כוירטואלי.
2. בחיים לא לקרוא לפונקצייה וירטואלית בזמן הבנאי/מפרק.
3. להשתמש בפונקציות ורטואליות טהורות על מנת לממש מחלקות אבסטרקטיות/"ממשקים".

למבדה



איטרטור

איטרטור הוא אובייקט המאפשר מעבר על איברי קבוצה נתונה.

```
template <typename T>
class Container {
    T a,b;
public:
    Container(T a,T b) : a(a), b(b) {}

    struct iterator {
        // Constructor
        iterator(T type) : type(type) {}
        bool operator!=(const iterator& iter) const { return type !=
iter.type; }
        const T& operator*() const { return type; }
        iterator& operator++(){ ++type; return *this; }

        private:
        T type;
    };
```

```
    iterator begin() const { return iterator(a); }
    iterator end() const { return iterator(b); }

};

void for_each_function(int element) {
    std::cout << element << "  ";
}


int main() {
   Container<int> c(1,5);

    Container<int>::iterator it = c.begin();
    // Best way to iterator:
    for (const auto& element : c)
    {
        std::cout << element << "  "; // 1 2 3 4
    }
    std::cout << std::endl;
}
```

תזכורת: Erase מחזירה איטרטור אשר מצביע לאיבר **הבא** אחרי האיבר שנמחק.

Erasing during iteration (folder 3)

```
Container<...> c;
...
for(auto i= c.begin(); i!=c.end();/*no ++i*/ )
  if( f( *i ) ) { // some test
    i = c.erase(i);
  } else {
    ++i;
  }
```

כאשר אנחנו מוחקים איבר שאנחנו מצביעים עליו מתבצע:

- ברשימה, מפה וקבוצה - האיטרטור מכיל פוינטר המצביע למקום לא מאותחל בזיכרון - שגיאה חמורה.
- בוקטור - האיטרטור מצביע לאיבר הבא אחרי האיבר שמחקנו - לא שגיאה כל-כך חמורה, אבל עדיין לא מה שרצינו.

אז מה עושים? החל מ- C++11, השיטה erase מחזירה איטרטור מעודכן ותקין לאחרי המחיקה. צריך פשוט לשים את האיטרטור הזה באיטרטור שלנו. ראו דוגמה בתיקיה 3.

הדוגמא:

```
template<typename Container, typename Iterator>
void erase_odd_elements_FAIL(Container& c, Iterator b, Iterator e) {
    for (; b!=e; ++b) {
        bool is_odd = (*b)%2 != 0;
```

```
        if (is_odd)
            c.erase(b);
    }
}
template<typename Container, typename Iterator>
void erase_odd_elements_GOOD(Container& c, Iterator b, Iterator e) {
    for (; b!=e;) {
        bool is_odd = (*b)%2 != 0;
        if (is_odd)
            b = c.erase(b);
        else
            ++b;
    }
}

int main() {
        vector<int> v {1,2,4,7,11,16,22};
        erase_odd_elements(v, v.begin(), v.end());
}
```

לכל איטרטור יש גם גירסא שהיא const – גירסא המאפשרת לקרוא את הפריטים במיכל אבל לא לשנות אותם, ניתן לגשת אליהם באופן הבא:

```
std::vector<int> v = {0,1,2,3,4,5};
    std::vector<int>::iterator it = v.begin();
    std::vector<int>::const_iterator cit = v.begin();
    // *it = *it + 1; // Works
    // (****)  *cit = *cit + 1; // Fails
```

Iterators

begin cbegin	returns an iterator to the beginning (public member function)
end cend	returns an iterator to the end (public member function)
rbegin crbegin	returns a reverse iterator to the beginning (public member function)
rend crend	returns a reverse iterator to the end (public member function)

מיכלי STL

Vector: (זמן הכנסה בהתחלה/אמצע הוא לינארי, זמן הכנסה בסוף קבוע בבממוצע, וזמן גישה לאיבר באמצע הוא קבוע)

Insert,Emplace לוקחים גם איטרטור, push_back לא לוקח איטרטור

```
    std::vector<int> v;

    // It will create new object and then copy(or move) its value of
arguments.
```

```
    v.push_back(0);
    v.push_back(1);
    v.push_back(2);

    // insert copies objects into the vector. (Make Temporary Element)
   v.insert(v.begin(), 5); // 5 0 1 2
   v.insert(v.begin()+1,6); // 5 6 0 1 2

   // emplace construct them inside of the vector (Better)
      // MUST TAKE ITERATOR!
   v.emplace(v.begin(),0); // 0 5 6 0 1 2

   std::cout << v.end() - v.begin() << std::endl; // 6
   std::cout << v.size() << std::endl; // 6

   v.resize(10,5); // 0  5  6  0  1  2  5  5  5  5
   v.resize(1); // 0

    v.clear(); // Clear Content
  while (!v.empty())
  {
    v.pop_back();
  }
```

מעבר בעזרת אופרטור:

```
output_containers.hpp
```

```
template<typename T>
ostream& operator<< (ostream& out, const vector<T>& c) {
    for (int i: c)
        out << i << ' ';
    return out;
}
```

Average Time Complexity

If we inserted **n** elements we paid:
1+**2**+1+**4**+1+1+1+**8**+…+**n** =
O(n) + **1**+**2**+**4**+…+**n** =
O(n)

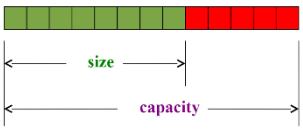
On average an each insertion cost O(1)

```
#include "output_containers.hpp"
int main() {
    std::vector<int> v = {0,1,2,3,4,5};
    auto i = v.begin();

    std::cout << v << std::endl;
}
```

כדי לחסוך זמן נגדיר קיבלת על ההתחלה עם (15) לדוג'

הערכים בוקטור מסוג Vector::value_type



- **The first "size" elements are constructed (initialized)**
- **The last "capacity - size" elements are uninitialized**
- push_back / emplace_back use the uninitialized elements until they are full; then, they **multiply the vector size by 2**.

Set: קבוצה של איברים **יחודיים ממויינים** >
MultiSet : קבוצה של איבר **ממויינים** >
Unordered_set : קבוצה של איבר **יחודיים**

ניתן להוסיף עם Insert או Emplace ללא איטרטור!

הערה: ניתן להגדיר סדר על המיכלים (Set,map וכו') לדוגמא:

```
struct SederYored {
    bool operator()(int x , int y) { return x > y; };
};
int main() {
    std::set<int,SederYored> s;
    // std::set<int,std::less<int> > s;
    // std::set<int, std::greater_equal<int> > s;
    // std::set<int , std::greater<int> > s;
    s.insert(1);
    s.insert(2);
    std::for_each(s.begin(), s.end(),[](const int& s) {
    std::cout << s << "  ";
    }); // Prints 2, 1
```

```
std::unordered_set<double> s_uo = {0,1,-5,3,0,0}; // Unorder
std::set<double> s_reg = {0,1,-5,3,0,0};  // Ordered + Without Coppies
std::multiset<double> s_ms = {0,1,-5,3,0,0}; // With Coppies
// s_uo = -5 3 1 0
std::for_each(s_uo.begin(),s_uo.end(),[](const double& d) { std::cout << d << " "; });
// s_reg = -5 0 1 3
std::for_each(s_reg.begin(),s_reg.end(),[](const double& d) { std::cout << d << " "; });
// s_ms = -5 0 0 0 1 3
std::for_each(s_ms.begin(),s_ms.end(),[](const double& d) { std::cout << d << " "; });

s_uo.insert(-1); s_ms.insert(-1); s_reg.insert(-1);
s_uo.insert(-1); s_ms.insert(-1);  s_reg.insert(-1);
s_uo.insert(-1); s_ms.insert(-1);  s_reg.insert(-1);
s_uo.insert(2.2);  s_ms.insert(-1);  s_reg.insert(-1);
// s_uo = 2.2 -1 -5 3 1 0
std::for_each(s_uo.begin(),s_uo.end(),[](const double& d) { std::cout << d << " "; });
// s_reg = -5 -1 0 1 3
std::for_each(s_reg.begin(),s_reg.end(),[](const double& d) { std::cout << d << " "; });
// s_ms = -5 -1 -1 -1 -1 0 0 0 1 3
std::for_each(s_ms.begin(),s_ms.end(),[](const double& d) { std::cout << d << " "; });
// Same for all
s_uo.erase(-1); // 2.2 0 1 2 3
s_uo.erase(s_uo.begin()); // 0 1 2 3
s_uo.erase(s_uo.begin(),s_uo.end()); // Empty
```

Map: קבוצה של מפתח:ערך **יחודיים ממויינים** <

Multi Map : קבוצה של מפתח:ערך **ממויינים** <

Unordered_ Map : קבוצה של מפתח:ערך **יחודיים**

```
std::unordered_map<double,std::string> m_uo = { {1,"A"}, {1,"B"}, {1,"B"}, {2.5,"C"}, {1.2,"C"} }; // Unorder
std::map<double,std::string> m_reg = { {1,"A"}, {1,"B"}, {1,"B"}, {2.5,"C"}, {1.2,"C"} };  // Ordered + Without Coppies
std::multimap<double,std::string> m_ms = { {1,"A"}, {1,"B"}, {1,"B"}, {2.5,"C"}, {1.2,"C"} }; // With Coppies
// m_uo = (1.2,C) (2.5,C) (1,A)
// m_reg = (1,A) (1.2,C) (2.5,C)
// m_ms = (1,A) (1,B) (1,B) (1.2,C) (2.5,C)
// Will NOT add
m_reg.insert(std::make_pair(1,"Hey"));
// Will ADD
m_reg.insert(std::make_pair(5,"Sup"));
// (1,A) (1.2,C) (2.5,C) (5,Sup)

// m_reg = (1,A) (1.2,C) (2.5,C)
std::cout << m_reg[1] << std::endl; // A
// Change
m_reg[1] = "Chaning";
std::cout << m_reg[1] << std::endl; // Chaning
std::cout << m_reg.at(1) << std::endl; // Chaning
std::cout << m_reg[100] << std::endl; // Undefined
std::cout << m_reg.at(100) << std::endl; // empty
// Print
for(std::map<double,std::string>::iterator it = m_reg.begin(); it != m_reg.end(); it++) {
        std::cout << "(" << it->first << "," << it->second << ") ";
}
std::cout << std::endl;
// Or:
std::for_each(m_reg.begin(),m_reg.end(),[](const auto& p)
{ std::cout << "(" << p.first << "," << p.second << ")" << " "; }); std::cout << std::endl;
```

Modifiers

clear	clears the contents (public member function)
insert	inserts elements or nodes (since C++17) (public member function)
insert_or_assign (C++17)	inserts an element or assigns to the current element if the key already exists (public member function)
emplace (C++11)	constructs element in-place (public member function)
emplace_hint (C++11)	constructs elements in-place using a hint (public member function)
try_emplace (C++17)	inserts in-place if the key does not exist, does nothing if the key exists (public member function)
erase	erases elements (public member function)
swap	swaps the contents (public member function)
extract (C++17)	extracts nodes from the container (public member function)
merge (C++17)	splices nodes from another container (public member function)

Lookup

count	returns the number of elements matching specific key (public member function)
find	finds element with specific key (public member function)
contains (C++20)	checks if the container contains element with specific key (public member function)
equal_range	returns range of elements matching a specific key (public member function)
lower_bound	returns an iterator to the first element *not less* than the given key (public member function)
upper_bound	returns an iterator to the first element *greater* than the given key (public member function)

deque: תור-דו כיווני – זמן הכנסה בהתחלה/סוף קבוע, וגם זמן הגישה לאיבר באמצע הוא קבוע אבל פחות יעיל מוקטור.
יש כמה מימושים, אחד המימושים הוא וקטור של וקטורים. הוקטור הראשי מכיל פוינארים לוקטורים המשניים ושומר מקום פנוי גם בהתחלה וגם בסוף.

```
std::deque<int> q = {1,2,3,4,1,1,1,1}; // 1 2 3 4 1 1 1 1
q.emplace(q.begin(),1); // Must take iterator
q.pop_back();
q.push_back(1);
q.pop_front();
q.push_front(1);
std::cout << q.at(2) << std::endl; // 2


std::for_each(q.begin(),q.end(),[](const int& i) { std::cout << i << "
"; });
//1 1 2 3 4 1 1 1 1
```

List: רשימה מקושרת (זמן הכנסה בהתחלה/אמצע/סוף הוא קבוע, אבל זמן הגישה לאיבר באמצע הרשימה הוא לינארי)
יש Sorting, Splicing, Removing!
אין Random Access!

```
std::list<int> list1 = {1,2,3,4,1,1,1,1}; // 1 2 3 4 1 1 1 1
std::list<int> list2 = {1,2,3,4,1,1,1,1};


list1.insert(list1.begin(),3); // Must take ITERATOR!!!!!
list1.emplace(list1.begin(),3); // Must take ITERATOR!!!!!
list1.erase(list1.begin());
// Before 3 3 1 2 3 4 1 1 1 1 After 3 1 2 3 4 1 1 1 1

list1.pop_back(); // 3 1 2 3 4 1 1 1
list1.pop_front(); // 1 2 3 4 1 1 1
list1.resize(10,5); // 1 2 3 4 1 1 1 5 5 5
list1.reverse(); // reverses the order of the elements 5 5 5 1 1 1 4 3
2 1
list1.sort(); // sorts the elements d 1 1 1 1 2 3 4 5 5 5
// list1.unique(); // removes consecutive duplicate elements
list2.swap(list1);
std::for_each(list2.begin(),list2.end(),[](const int& i) { std::cout <<
i << " "; });
// 1 2 3 4 5
```

אלגוריתמאי STL https://www.youtube.com/watch?v=2olsGf6JIkU
בספרייה התקנית של C++ יש 105 אלגורתמים נכון לשנת 2017.

```
std::minmax_element(v.begin(),v.end());
```

```
std::all_of(v.begin(),v.end(), is_even )
```

```
std::none_of(v.begin(),v.end(), is_even )
```

```
std::any_of(v.begin(),v.end(), is_even )
```

```
std::copy(v.begin(),v.end(), d1.begin());
std::transform(v.begin(),v.end(),
                v.begin(),op_increase);
std::find(v.begin(),v.end(),-1);

std::count(v.begin(),v.end(),1)
```

```
std::fill(v.begin(), v.end(), -1);
```

```
std::lower_bound(v.begin(), v.end(), 2);
```

```
std::upper_bound(v.begin(), v.end(), 5);
```

```
std::binary_search(v.begin(),v.end(), 1);
```

```
std::sort(v.begin(),v.end());
std::accumulate(v.begin(),v.end(),0,lambda);

    std::for_each(v.begin(), v.end(),
                  [](int &n){ n++; });
```

קודים לדוגמא:

```
    // Before:
    auto print = [](const int& n) { std::cout << " " << n; };
    std::for_each(v.begin(),v.end(),print);
    std::cout << std::endl;
    std::for_each(v.begin(), v.end(), [](int &n){ n++; });
    std::cout << std::endl;
    std::for_each(v.begin(),v.end(),print);
```

For each

```
int op_increase (int i) { return ++i; }
int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(-1);
    v.push_back(5);

int (*g)(int) = &op_increase; // pointer g to a function returning int
    std::transform(v.begin(),v.end(),v.begin(),*g); // 2 0 6
    // Could put op_increase instead of *g !!!!
    std::cout << v[0] << std::endl;
    std::cout << v.get(0) << std::endl;
    // v[0] makes operator[] is undefined behavior while at(0) throws
```

Transform

```
    v = [ 1 , -1 , 5 ]
    auto it = std::find(v.begin(),v.end(),-1);
    if( it != v.end()) {
        while(it != v.end()) {
            std::cout << *it << " ";
            ++it;
        } // Prints -1,5
        std::cout << std::endl;
    }
```

Find

אחר :

```
    // (*) -> [ V = 1 -1 5 3 ]
    //std::reverse
    // std::reverse(v.begin(), v.end()); // Reverse ->  3 5 -1 1

    // std::copys
    std::vector<int> d1(v.size());
    std::copy(v.begin(),v.end(), d1.begin());
    std::cout << std::equal(v.begin(),v.end(), d1.begin()) << std::endl; // True == 1

    //std::sort
    std::sort(v.begin(),v.end()); //  -1 1 3 5

    //std::binary_heap return boolean
    std::cout << std::binary_search(v.begin(),v.end(), -1); // True == 1
    std::cout << std::binary_search(v.begin(),v.end(), -2) << "\n"; // False == 0

    //std::lower_bound, std::upper_bound
    auto it1 = std::lower_bound(v.begin(), v.end(), 2); // >=
    auto it2 = std::upper_bound(v.begin(), v.end(), 5); // <

    std::cout << ( it1 != v.end() ? std::to_string(*it1) : "Not Found") << std::endl; // 3
    std::cout << ( it2 != v.end() ? std::to_string(*it2) : "Not Found") << std::endl; // "Not
Found"

   // std::any_of, none_of,
    auto is_even = [](int n) { return n%2 == 0; };
    std::cout << std::all_of(v.begin(),v.end(), is_even ) << std::endl; // False == 0
    std::cout << std::none_of(v.begin(),v.end(), is_even ) << std::endl; // False == 1
    std::cout << std::any_of(v.begin(),v.end(), is_even ) << std::endl; // False == 0
   // v.clear();
    v.push_back(-10);
    std::cout << std::any_of(v.begin(),v.end(), is_even ) << std::endl; // True == 1

    // std::min, std::max, std::min_max
    std::cout << *(min_element(v.begin(),v.end())) << std::endl; // -10
    std::cout << *(max_element(v.begin(),v.end())) << std::endl; // 5
    const auto result = std::minmax_element(v.begin(),v.end());
    std::cout << *result.first << "," << *result.second << std::endl; // -10,5
    std::fill(v.begin(), v.end(), 1);
    // accumulate function accumulate the container on the basis of
    // function provided as third argument
    auto lambda = [](int i,int j) { return i+j; };
    std::cout << std::accumulate(v.begin(),v.end(),0, lambda ) << std::endl; //5
```

אלגרותמים נוספים :

```
std::vector<int> diff;
std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
                    std::inserter(diff, diff.begin()));
```

```
std::size(Collection)
```

```
            << std::distance(v.end(), v.begin()) << '\n';
```

```
int main()
{
   int a = 5, b = 3;
   // before
   std::cout << a << ' ' << b << '\n';
   std::swap(a,b); // Get the Value! not Iterator!!!
   // after
   std::cout << a << ' ' << b << '\n';
}
```

לאפס מערך

```
int arr[5] = {-1};
    for(int i=0; i<5; i++) {
        std::cout << arr[i] << std::endl;
    }
```

פלט: [-1,Junk, Junk, Junk, Junk]

הדרך הנכונה הינה:

```
  int arr[5];
    std::fill(arr,arr+5,-1);
```

ניתן גם להשתמש באופן הבא:

```
    vector<int> v = {1,2,3,4,5};
    std::fill(v.begin(), v.end(), -1);

    for(int i=0; i<v.size(); i++) {
        std::cout << v.at(i) << std::endl;
    }
```

פונקציית הדפסה – מאקרו

```
#define LOG(x) std::cout << x << std::endl

int main() {
        LOG("Check Log Print");
```

מחרוזות:

```
  char c = 'a';
```

```
    std::string s;
    // s = std::to_string(c); // First Way

    std::stringstream buffer;  // Second Way
    buffer << c;
    buffer >> s;
    std::cout << s << std::endl; // a ;

    std::ostringstream foo;
    std::ostringstream bar(std::ostringstream::ate);  // out|ate
    foo.str("Test String");
    bar.str("Test String");
    foo << 404;
    bar << 404;
    std::cout << foo.str() << '\n'; // 404t String
    std::cout << bar.str() << '\n'; // Test String404
```

Tuples:

```
#include <tuple>
#include <string>
auto f() { return std::tuple<int,char,std::string>(5,'a',"SUP?"); }
int main() {
    auto x = f();
    std::cout << std::get<0>(x) << " " << std::get<1>(x) << " " <<
std::get<2>(x) << std::endl;
}
```

ריצה ב-**For Each** של מערך רגיל – עובד מ גרסא 11 ומעלה.

```
    int array[20];
    for(int i : array)
    {
        std::cout << "Here";
    }
```

עובד.

תבניות

נרצה להשתמש בתבניות במקרה שבהם נרצה לכתוב פונקצייה כללית המתאימה לטיפוסי משתנים שונים אבל תבצעת בצורה שונה לכל טיפוס.

- Templates הינם כלי מאוד שמושי לצורך הגדרת Containers.
- למרבה הצער המימוש של Templates ב - C++ מזכיר macro מכם ולכן כל הקוד של ה - Template נמצא ב - header file כפי שהוצג או כ - inline function.
- ישנן לא רק מחלקות שהן גנריות אלא גם פונקציות גנריות.

דוגמא:

```
template<typename T> void swap(T& a, T&b) {
    T temp = a;
    a = b;
    b = temp;
}
```

כשמגדירים תבנית, הקומפיילר זוכר את הגדרה אבל עדיין לא מייצר שום קוד. תבנית היא לא רק – היא רק מרשם לייצר קוד, הקוד שנוצר, רק כשמנסים להפעיל את התבנית ע"י קריאה מתאימה אז תהיה יצירה לפונקצייה עם הפרמטרים המתאימים ע"י הקומפיילר. בגלל שהקומפיילר יכול לראות רק קובץ CPP אחד בלבד, המימוש חייב להיות בhpp היות ואם נממש טמפייל באופן כלשהו, אז הקומפיילר לא יוכל לראות זאת בקובץ CPP אחר.

- אילו הנחות הניח major על T?
  - קיום copy c'tor (העברת פרמטר לפונקציה by value).
  - קיום operator=.

זה בגלל שצריך ליצור עצם חדש מסוג T ושאפשר לבצע העתקה.

איך הקומפיילר בוחר לאיזו פונקציה לקרוא?

כשיש כמה פונקציות עם אותו שם ואותו מספר פרמטרים, הקומפיילר בוחר ביניהם באופן הבא:

- קודם-כל, הוא בוחר את כל הפונקציות עם רמת ההתאמה הגבוהה ביותר (הכי פחות המרות סוגים).
- בתוך הקבוצה עם רמת ההתאמה הגבוהה ביותר, הוא בוחר את הפונקציות שהן לא תבניות, ורק אם אין כאלה - הוא מפעיל את התבניות.

הדבר מאפשר לכתוב תבנית כללית, ויחד איתה, פונקציה ספציפית יותר הפועלת באופן שונה על סוגים שונים. הקומפיילר יבחר את הפונקציה הספציפית אם היא מתאימה; אחרת - הוא יבחר את הפונקציה הכללית יותר. יש דוגמאות רבות במצגת. דוגמאות נוספות:

- swap - עבור טיפוסים מספריים - אפשר לבצע בעזרת פעולות חיבור וחיסור ובלי משתנה זמני (זה שימושי רק אם מאד חשוב לנו לחסוך במקום על המחסנית).
- swap - עבור מחלקות עם העתקה עמוקה - אפשר לבצע במהירות רבה יותר ע"י העתקה שטחית.

<u>תבניות של מחלקות:</u>

ניתן גם להגדיר תבניות עבור מחלקות באופן הבא:

```
template <typename T> class Stack {
    T data
}
```

<u>הערה:</u> class T **<u>זהה בדיוק</u>** ל- typename T

תבניות יכולות לקבל **<u>יותר</u>** מפרמטר אחד באופן הבא:

```
template <typename Key, typename Value> class pair {
    Key k;
    Value v;
}
```

תבניות יכולות לקבל פרמטרים שהם לא סוגרים אלא מספרים, למשל בשביל ליצור מחלקה המייצגת מערכך שהגודל שלו ידוע בזמן קומפלצייה.

למה זה טוב? כדי לקבל מבנה זהה לחלוטין למערך של C, וכדי שנוכל להוסיף לו שיטות שונות.

בנוסף, ניתן לבצע **<u>אופטמיזציות</u>**.

<u>דוגמא:</u>

```
template <typename T, int size> class array {
    T m_values[size];
};
```

```
int main(){
    array<int,20> arr;
}
```

או עם גודל קבוע:

```
template <typename T, int size = 20> class array {
    T m_values[size];
};
```

אפשר גם לחייב לפרמטר השני להיות מסוג T לדוגמא:

```
template <typename T, T size> class array {
    T m_values[size];
};
int main(){
    array<int,"Hey"> arr; // compile error
}
```

התמחות

הגדרת מקריים פרטיים לתבניות: Template Specialization

אפשר להגדיר מקרים פרטיים של תבנים עם מימוש שונה אשר מתאים לסוג מסויים.

לדוגמא:

Template class specialization (folder 3)

```
template <typename T> class Test {
  public:  Test()  {     cout << "General";  }
};

template <> class Test <int> {
  public: Test()    {     cout << "Specialized";    }
};

int main()  {
    Test<int> a;       // Specialized
    Test<char> b;   // General
    Test<float> c;   // General
}
```

ניתן להשתמש בתבניות בשביל Meta-Programming באופן הבא:

יתרון: יצירת תוכניות שלמות שרצות בזמן הקומפלצייה

Template Meta-Programming

```
// primary template computes 3 to the Nth
template<int N> class Pow3 {  public:
    static const int result =
        3*Pow3<N-1>::result;
};
// full specialization to end recursion
template<> class Pow3<0> {  public:
    static const int result = 1;
};
int main(){
  cout << Pow3<1>::result<<"\n"; //3
  cout << Pow3<5>::result<<"\n"; //243
  return 0;
}
```

```
template<typename T> void foo(T data) { std::cout << "Template" << std::endl; }
template<> void foo<int>(int data) { std::cout << "Int Template" << std::endl; }

int main()
{
    foo(3);
    foo(3.5);
}
```

סדר קריאות:

```
template <class T>
void foo(T num) { cout << "T" << endl; }

template <>
void foo<char>(char num) { cout << "T char" << endl; }

void foo(int x) { cout << "regular foo" << endl; }

int main() {
    foo(3); // Regular foo
    foo(3.3); // T
    foo('a'); // T Char
    foo("Hey"); // T
}
```

1. פונקציה רגילה
2. התמחות
3. תבנית

- לא מבצע המרה.

Decltype:
דוגמא:

```
int i = 33;
 decltype(i) j = i * 2;
```

Auto:

```
  auto c = a;
  // equivalent to:
  // decltype(a) c = a;
```

Static_assert –
גילוי שגיאות בזמן קופלצייה, אם יש שגיאה הקומפיילר יודע על כך.

- `static_assert( constant_expression, string_literal );`

דוגמא:

```
template <class T, int Size>
class Vector {
    // Compile time assertion to check if
    // the size of the vector is greater than
    // 3 or not. If any vector is declared whose
```

```
    // size is less than 4, the assertion will fail
    static_assert(Size > 3, "Vector size is too small!");

    T m_values[Size];
};

int main()
{
    Vector<int, 4> four; // This will work
    Vector<short, 2> two; // This will fail

    return 0;
}
```

פלט:

```
error: static assertion failed: Vector size is too small!
```

Mutable function

מאפשר לשנות משתני עצם מפונקצייה קבועה.

```
class A {
   mutable int x;
   int y;

   public:
     void f1() {
       // "this" has type `A*`
       x = 1; // okay
       y = 1; // okay
     }
     void f2() const {
       // "this" has type `A const*`
       x = 1; // okay
       y = 1; // illegal, because f2 is const
     }
};
```

Casting

Const_cast :

להוריד או להוסיף const לפוינטר

```
    int i = 3;                  // i is not declared const
    const int& rci = i;
    const_cast<int&>(rci) = 4; // OK: modifies i
    std::cout << "i = " << i << '\n'; // i = 4

    const int& j = 4;
    // j = 3; // Error
    const_cast<int&>(j) = 3; // OK: modifies j
    std::cout << "j = " << j << "\n"; // j = 3
```

ד

דוגמא:

Static cast :

static_cast< Type* >(ptr)

לוקח פוינטר ומנסה להמיר בצורה **בטוחה** בזמן **קומפלציה** לטיפוס Type*

לדוגמא:

```
class B {};
class D : public B {};
class X {};

int main()
{
  D* d = new D;
  B* b = static_cast<B*>(d); // this works
  X* x = static_cast<X*>(d); // ERROR - Won't compile
  return 0;
}
```

Dynamic class:

dynamic_cast< Type* >(ptr)

מנסה לקחת פוינטר ולהמיר בצורה **בטוחה** לפוינטר מטיפוס Type*, אבל זה מתבצע בזמן ריצה ולא בזמן קומפלצייה. בגלל שזה ממור בזמן ריצה, זה משומש בעיקר בפולימרופיזם.
ניתן לעשות זאת רק ע"י Polymorphic types, שבהם יש לפחות פונקצייה אחת וירטואלית.
לדוגמא הקוד הבא **לא** יתקמפל:

```
class Base {};
class Der : public Base {};
int main()
{
    Base* base = new Der;
    Der* der = dynamic_cast<Der*>(base); // ERROR - Won't compile

    return 0;
}
```

אבל זה **כן**:

```
class Base { virtual void foo() {} };
class Der : public Base {};
int main()
{
    Base* base = new Der;
    Der* der = dynamic_cast<Der*>(base); // ERROR - Won't compile
    return 0;
}
```

הערות נוספות

וקטור בזכרון:

```
vector<Type> vect;
```

will allocate the `vector`, i.e. the header info, on the stack, but the elements on the free store ("heap").

```
vector<Type> *vect = new vector<Type>;
```

allocates everything on the free store.

זליגת זכרון:

```
myPointer = new int;
myPointer = NULL; //leaked memory, no pointer to above int
delete myPointer; //no point at all
```

שגיאות זמן ריצה:

1. Segmentation fault
2. Core dump

שגיאות קומפצייה:

יהיה רשום מספר שורה ועמודה ביחד עם שם הקובץ

שגיאות לינקר:

יהיה רשום שם הקובץ out.

קריאה לפונקצייה const ע"י משתנה לא const

```
void g1(std::string& s);
void f1(const std::string& s)
{
  g1(s);          // Compile-time Error since s is const
  std::string localCopy = s;
  g1(localCopy);  // Okay since localCopy is not const
}
```

קודים לדוגמא:

```
struct A {
    A() { std::cout << "A" << std::endl; }
    ~A() { std::cout << "~A" << std::endl; }
    A(const A& other) { std::cout << "A copy" << std::endl; }
    A& operator=(const A& other) {
        { std::cout << "A =" << std::endl; }
        return *this;
    }
    void foo(A other) { std::cout << "A foo" << std::endl; }
    void fooRef(A& other) { std::cout << "A foo other" << std::endl; }
};
```

פלט:

```
A copy
A foo
~A

A foo
other
```

```
struct B : public A{};
int main() {
    A* a = new B;
    std::cout << std::endl;
    a->foo(*a);
    std::cout << std::endl;
    a->fooRef(*a);
    return 0;
}
```

```
#include <iostream>
using namespace std;
struct A {
    void draw() const { cout << "a" << endl; }
    void g() const { draw(); }
    void f1(const A other) const {other.draw();}
    void f2(const A& other) const { other.draw(); }
};

struct B : public A{
    void draw() const {cout << "b" << endl;}
};
int main() {
    B b;
    b.draw(); // b
    b.g(); // a
    b.f1(b); // a
    b.f2(b); // a
    cout << endl;
    A* a = new B; // a
    a->draw(); // a
    a->g(); // a
    a->f1(*a); // a
    a->f2(*a); // a
    delete a;
}
```

פלט:

```
b
a
a
a

a
a
a
a
```

```
struct A {
    virtual void draw() const { cout << "a" << endl; }
    void g() const { draw(); }
    void f1(const A other) const {other.draw();}
    void f2(const A& other) const { other.draw(); }
};

struct B : public A{
    void draw() const {cout << "b" << endl;}
};
```

פלט:

```
b
b
a
b

b
b
a
b
```

```
int main() {
    B b;
    b.draw(); // b
    b.g(); // b
    b.f1(b); // a
    b.f2(b); // b
    cout << endl;
    A* a = new B;
    a->draw(); // b
    a->g(); // b
    a->f1(*a); // a
    a->f2(*a); // b
    delete a;
}
```

```
#include <iostream>
using namespace std;
struct A {
    A() { draw(); }
    ~A() { cout << "~A" << endl; }
    virtual void draw() const { cout << "a" << endl; }
};

struct B : public A{
    void draw() const {cout << "b" << endl;}
    ~B() { cout << "~B" << endl; }
};
int main() {
    B b; // a
    b.draw(); // b
    cout << endl;
    A* a = new B; // a
    cout << endl;
    a->draw(); // b
    delete a; // ~A
    return 0;
}
```

פלט:

```
a
b

a

b
~A
~B
~A
```

פלט מצויין בהערות:

```
#include <iostream>
using namespace std;

class A {
      public:
            A() {cout << "A::A()" << endl;}
```

```
            A(const A& a) {cout << "A::A(A&)" << endl;}
            A& operator=(const A& a) {cout << "A::op=" << endl; return
*this;}
            ~A() {cout << "A::~A()" << endl;}
            virtual void f(const int& x) {cout << "A::f()" << endl;}
            void g() {cout << "A::g()" << endl;}
            virtual void h() {cout << "A::h()" << endl;}
            virtual int i() {cout << "A::i()" << endl; return 1;}
            void j(const int x) {cout << "A::j()" << endl;}
};
class B: public A {
      public:
            B() {cout << "B::B()" << endl;}
            B(const B& b) {cout << "B::B(B&)" << endl;}
            ~B() {cout << "B::~B()" << endl;}
            void f(int& x) {cout << "B::f()" << endl;}
            void g() {cout << "B::g()" << endl;}
            void h()const {cout << "B::h()" << endl;}
            int i() {cout << "B::i()" << endl; return 1;}
            virtual void k(char c) {cout << "B::k()" << endl;}
};
class C: public B {
      public:
            C() {cout << "C::C()" << endl;}
            ~C() {cout << "C::~C()" << endl;}
            void h() {cout << "C::h()" << endl;}
            int i(const int x) {cout << "C::i()" << endl; return x+1;}
            void k(char c) {cout << "C::k()" << endl;}
};
void f1(A a1){
       A a2 = a1;
}
B* f2(A& a) {
         B b;
         B* pb = new B(b);
         a = b;
         return pb;
}

int main() {
    cout << "1:" << endl; // 1
    A* pa; C* pc;
    pa = new A(); pc = new C(); // A::A()    ;     A::A(), B::B(),
C::C()
    cout << "2:" << endl; // 2
    A a; f1(a);  // A::A(), A::A(A&), A::A(A&), A::~A() A::~A()
    cout << "3:" << endl; // 3
```

```
    B* pb = f2(a); //
A::A(),B::B(),A::A(),B::B(&),A::oper=,~B::B(),~A::A()
     int x = a.i();  // A::i()
    cout << "4:" << endl;
    A* pab = pb;
    A* pac = pc;
    B* pbc = pc;
    cout << "5:" << endl;
    pab->f(1); // B::f()
    pab->g(); // A::g()
    pab->h(); // C::h()
    delete(pab); // ~A:A()
    cout << "6:" << endl;
    pac->h(); // C::h()
    pac->i(); // C::i()
    cout << "7:" << endl;
    pbc->f(x); // B::f()
    pbc->k('a'); // C::k()
    pbc->j(x); // A::j()
    cout << "8:" << endl;
    delete(pc); // ~C::~C() ~B::B() ~A::A();
    delete(pa); // ~A::A();
    return 0;
}
```

סדר קריאה:

1. בנאי בסיס
2. אתחול משתני עצם
3. בנאי המחלקה

```
B b;
```

מוציא פלט: A ctor, B ctor, B dtor, A dtor

```
    A* a = new B();
    delete a;
```

מוציא פלט: ~A() ~B() כאשר מפרק הבסיס וירטואלי!

אלגוריתם לכתיבת טבלת V-Table:

1. עבור כל פונקציה Virtual במחלקת הבסיס A לכתוב A::<func name> **כולל** מפרק **וכולל** פונקציות וירטואליות טהורות.
2. עבור כל מחלקה שיורשת ממחלקה אחרת, לרשום את כל הפונקציות של מחלקת הבסיס **חוץ** מהמפרק של מחלקת הבסיס(אם הוא וירטואלי), לאחר מכן לבדוק איזה פונקציות במחלקה היורשת הם Override, ולשנות את המצביע שלהם (B::: במקום A::), לאחר מכן **להוסיף** את המפרק אם הוא וירטואל.

NDEBUG

```
#ifdef NDEBUG
    #define assert(condition) ((void)0)
#else
    #define assert(condition) /*implementation defined*/
#endif
```

Uniform Initialization in C++

שימוש:

```
type var_name{arg1, arg2, ....arg n}
```

לדוגמא:

```
class A {
    int arr[3];

    public:
        // initializing array using uniform initialization
        A(int x, int y, int z)
            : arr{ x, y, z } {};
```

הערות נוספות:

1. בקריאה לפונקצייה שמקבלת Reference **אין** שום קריאה, בעוד שבפונקצייה שמקבלת By value אז יש קריאה לבנאי מעתיק **ולבסוף** למפרק.

2. אם class B : public A וב-A אין פונקצייה f() וב-B יש פונקציה virtual f() אז לא ניתן לגשת ל-f()→ עם פוינטר של A!

3. **איטרטורים:** אופרטור != מחזיר bool והוא **const**, אופרטור++ מחזיר &, אופרטור * הוא **const** ומחזיר &.

```
    B* b = new B();
```

מפרק את B ואז את A.

4. אם יש את המחלקות A,B,C,D עם יחס ירושה ובונים פוינטר מסוג X, אם באחת ממחלקות האב יש מפרק וירטואלי אז נפרק את D,C,B,A, אחרת, נפרק את X ומחלקות האב.

5. #Pragma Once פותר בעיות של קומפלציה **ולא** לינקר.

6. הקוד הבא **עובד** אבל אין גישה לשדות פרטיים:

```
struct A { int x;};
ostream& operator<<(ostream& os, const A& other) {return os << "Hey"; }
int main() {
    A a;
    cout << a << endl;
}
```

7. Set משתמש באופרטור > על מנת למיין, אם ניצור Set של אובייקט שאין לו את האופרטור > נקבל שגיאת קומפלציה כי לא נדע לסדר.

8. B* b = new C() מדפיס מפרקים **רק אם** קוראים ל-Delete, אם **אחד** המפרקים בדרך למעלה הוא וירטואל אז הוא ידפיס עד אגף ימין במפרקים (עד C), הקריאה לבנאים של C ביצירה.

9. בשאלות של מה הקוד מדפיס, **לא** לשכוח מפרקים ופירוק משתני עצם!

10. שאלות של אם יש דליפת זכרון – להוסיף Delete אם צריך **ומפרק** וירטואלי למחלקת הבסיס.

11. זיכרון: שורות אתחול אז המשתנה שמקבלים הוא או באיזור הקוד (אם זה מחרוזת, או אם זה ערך שנמצא על המחסנית אז זה במחסנית), Static נמצא בגלובלי.

36) char pass[] = "5678";
37) puser = new User("**Sara**", pass);

12. בפונקיות שמתעסקות עם מיכלים לבדוק אם המיכל **ריק** בתחילה: (אם איטרטור עם טיפוס T להוסיף typename) ולא Const אלא Reference!

```
void counter_min(std::vector<T>& v) {
    if(v.empty()) return;
    T min = v.at(0);
    int counter = 0;
    for(typename vector<T>::iterator it = v.begin(); it != v.end();
++it ) {
        if(*it < min) {
            min = *it;
            counter = 1;
        }
        else
        {
            if(*it == min) counter++;
        }
    }
    std::cout << "Counter = " << counter << " Min = " << min << endl;
}
```

13. אם ידוע מה הסוג אפשר לעשות במקום Template אפשר:

```
void func(int* begin1 ,int* end1, int* begin2, int* end2, int* ans) {
while(begin1 != end1) {
    assert(begin2 != end2);
    end2--; // Must!!!
    *ans = *begin1 + *end2;
    begin1++;
    ans++;
}
Assert(begin2 == end2)
}
```

14. קבועים: ע"י **const** float INCH_TO_CM;

15. פוינטרים לא מדפיסים כלום, רק מה שבצד ימין מדפיס. לעומת זאת לא פוינטרים אז יכול להיות מבנאי, בנאי מעתיק או אופרטור =.

```
    A* a1 = new B(); // Print A,B
    std::cout << std::endl;
    A* a2; // Prints Nothing
    a2 = new B(); // Prints A,B
    std::cout << std::endl;
    A* a3; // Prints Nothing
```

```
    B b; // Prints A, B
    a3 = &b; // Nothing
    std::cout << std::endl;
    A a4,a5; // Prints A A
    a4 = a5; // Prints A = (Operator =)
    std::cout << std::endl;
    A a6; // Prints A
    A a7 = a6; // Prints A Copy
```

.16

```
bool has_pair_with_sum(std::vector<int> vec, int sum) {
    unordered_set<int> set(vec.begin(),vec.end());
    for(const auto& p : vec) {
        if(set.find(sum - p) != set.cend()) return true;
    }
    return false;
}
```

.17

שגיאת קומפלציה:

```
    const vector<int> v = {0,1,2,3,4,5};
    v[0] = 1;
```

.18 נוסף פוינטר של 8 ל-Table V.

```
struct A {
    int a;
    virtual void foo() {}
};
```

שווה ל-16

.19 גישה למצביע המכיל זבל בשחרור תגרור להתנהגות בלתי צפויה. (לא חייב להיות זבל, אלא לזכרון שכבר שוחרר)

.20 בנאי לא מאתחל משתני עצם אלא אם כן הוגדר לו לאתחל!

.21

```
    std::vector<int> v = {0,1,2};
    try { std::cout << v.at(3) << std::endl; }
    catch(const exception& e) { std::cout << "At() Throw Exception"; }

    // Undefined Behaivor : -564245800
    std::cout << v[12] << std::endl;
```

.22

לזרוק Struct:

```
struct negative_value{ void sup(){ std::cout << "Hey" << std::endl; }};
```

```
    try {
```

```
        throw negative_value();
    }
    catch(negative_value& other)
    {
        other.sup();
    }
```

23. אופרטור = לא לשכוח לבדוק אם this==&other, ואז למחוק את מה שיש ולבצע העתקה.

24. `std::runtime_error("The input value correspond to inserted value in the tree.");`

25. Pair vs Make_Pair

26. לשים לב באופרטור קלט לא לשים את istream כקבוע אחרתת אי אפשר להכניס למשתנה לא קבוע

27. לשים לב באופרטור אם פונקציה היא חברה אז לא צריך בקובץ מימוש להוסיף :: שם המחלקה.

```
    pair<int,std::string> pair(1,"Hey");
    cout << pair.first << "," << pair.second << endl;

    auto pair2 =  make_pair(1,"hey");
    cout << pair2.first << "," << pair2.second << endl;
```

28. ליצור מערך 2 מימדי עם אופרטור [][]

```
struct MyMatrix {
    int rows;
    int cols;
    int ** arr;
    MyMatrix(int rows,int cols) : rows(rows), cols(cols) {
        arr = new int*[rows];
        for(int i=0; i<rows; i++)
            arr[i] = new int[cols];
    }
    ~MyMatrix() {
        for(int i=0; i<rows; i++)
            delete[] arr[i];
        delete[] arr;
    }

    struct MatrixAt {
        friend class MyMatrix;

        int row;
        MyMatrix& parent;

        MatrixAt(MyMatrix& parent, int row) : parent(parent), row(row)
{}

        int& operator[](int col) {
```

```
            return parent.arr[row][col];
        }
    };
    MatrixAt operator[](int row) {
        return MatrixAt(*this,row);
    }
};
```

29. סדר אתחול:

1. מקבל ערך מבנאי (אם הוא לא & אז בנאי מעתיק)

2. לפי סדר המשתנים. אם מאתחל משתנה מורכב עם רשימת אתחול אז משתמשים בבנאי מעתיק!

30. *ptr++ = *(ptr++) = *ptr, ++ptr

31. סוגי שגיאות:

```
    static_assert(false,"Sup"); // Compile time
    assert(true) // Run time
```

מבחן 3

פתרון שאלה 6:

```
double twice(double x) { return 2*x; }

template <typename F>
double integral(F f, double intervals) {
    double ans = 0;
    for(double x=0; x<=intervals; x+=0.01) {
        double y = f(x);
        ans += y;
    }
    return ans*0.01;
}
int main() {
    cout << integral(twice,20) << endl;
    cout << integral([] (const double& x) -> double
                        { return 4*x; }
                        ,20) << endl;
    return 0;
}
```

מבחן 4

פתרון שאלה 7:

```
template <typename I1, typename I2>
void minmax(I1 begin1, I1 end1, I2 begin2, I2 end2) {
```

```
        while(true) {
        if(begin1 == end1 && begin2 == end2) break;
        else if(begin1 != end1 && begin2 == end2)
            throw "Sequence 1";
        else if(begin1 == end1 && begin2 != end2)
            throw "Sequence 2";
        else {
             if(*begin1 > *begin2)
            std::swap(*begin1,*begin2);
            begin1++;
            begin2++;
            }
        }
}
```

וקובץ הMain:

```
    try {
            minmax(vec1.begin(), vec1.end(), arr1, arr1+6); //
exception: "sequence 2 is longer"
    }
    catch(const char* msg) {
        std::cout << msg << std::endl;
    }
```

מבחן 5

```
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;
struct negative_value {};
class Proxy  {
    double& _d;
    public:
    Proxy(double& other) : _d(other) {}
    Proxy& operator=(double d){
        if (d < 0) throw negative_value();
        _d = d;
        return *this;
    }
    operator double() const  { return _d; }
};

class PositiveMatrix{
    double* vals;
    int rows, cols;
    public:
    PositiveMatrix(int rows, int cols) :
```

```
    rows(rows), cols(cols), vals(new double[rows*cols]) {
        for(int i=0; i<rows*cols; ++i)
            vals[i] = 0.0;
    }
    ~PositiveMatrix() { delete[] vals; } // [A]

    PositiveMatrix (const PositiveMatrix& other)  // [B]
    : rows(other.rows), cols(other.cols), vals(new double[rows*cols])
    {
        for(int i=0; i<rows*cols; ++i)
            vals[i] = other.vals[i];
    }
    // [C]
    PositiveMatrix& operator=(const PositiveMatrix& other) {
        if (this != &other)
        {
            delete[] vals;

            rows = other.rows;
            cols = other.cols;
            vals = new double[rows*cols];

            for(int i=0;i<rows*cols; i++)
                vals[i] = other.vals[i];
        }
        return *this;

    }
    Proxy operator() (int x,int y) {
        return vals[x*cols + y];
    }
};

int main() {

    PositiveMatrix m1(10,4);
    cout << m1(1,2) << endl;
    m1(1,2) = 3.0;
    cout << m1(1,2) << endl;

    PositiveMatrix m2(m1);
    m2(1,2) = 5;
    cout << m1(1,2) << endl; // Should print 3

    m1 = m2;
    m1(1,2) = 7;
    cout << m2(1,2) << endl;
```

```
    m1(1,2) = -5.0;

    return 0;
}
```

מבחן 6

```
#include "Computer.hpp"
class Macintosh : public Computer {
    std::string color;
    public:

    Macintosh(std::string name, std::string color) : Computer(name),
color(color) {}

    virtual void print() const override {
        std::cout << "Macintosh" << std::endl;
        Computer::PrintName();
        std::cout << "Color: " << this->color << std::endl;
    }
};
```

```
#include "Computer.hpp"
class PC : public Computer {
    int weight;
    public:
    PC(std::string name, int weight) : Computer(name), weight(weight)
{}
    virtual void print() const override {
        std::cout << "PC" << std::endl;
        Computer::PrintName();
        std::cout << "Weight: " << this->weight << std::endl;
    }
};
```

```
#pragma once;
#include <string>
#include <iostream>

class Computer {
    std::string name;
    public:
    Computer(std::string name) : name(name) {}

    virtual void print() const = 0;
```

```
    void PrintName() const {
        std::cout << this->name << std::endl;
    }
};
```


שימוש באיטרטור לדוגמא:

```
template <typename T>
struct MyArr {
    T* _ptr;
    int size;
    MyArr(int size) : _ptr(new T[size]), size(size) {
        for(int i=0; i<size; i++)
            _ptr[i] = (i % 3);
    }
    ~MyArr() { delete[] _ptr; }

    struct MyIterator {
        T* ptr;
        int size;
        MyIterator(T* ptr,int size) : ptr(ptr), size(size) {}
        bool operator!=(const MyIterator& other) const {
            return (other.ptr != ptr || size != other.size);
        }
        MyIterator& operator++() {
            size ++;
            return *this;
        }
        const T& operator*() const {
            return ptr[size];
        }
    };
    MyIterator begin() const { return MyIterator(_ptr,0); }
    MyIterator end() const { return MyIterator(_ptr,size); }

};



int main() {
    MyArr<double> arr(10);
    int i = 0;
    for(MyArr<double>::MyIterator it = arr.begin(); it != arr.end();
++it, i++ )
```

```
    {
        std::cout << i <<"'th Element = " << *it << std::endl;
    }
    return 0;
}
```