# סיכום סדנא
# ב-C + +

ע"פ חוברת הקורס של הרבנית החשובה

גב' **חני נדלר**

סוכם משיעוריו של הרה"ח

**אריה ויזן**

עם פירוש

# "רי"ח טוב"

נערך בחסדי ה'

מאיתי **יוחנן חאיק**

שנת 0001011010010001

ניתן להשתמש בסיכום באופן חופשי לכולם!!
להערות, הארות ותיקונים:
yohananha@gmail.com
@yohanan - בטלגרם

# תוכן עניינים


תכנות מונחה עצמים ..... 2
פונקציות חופפות והעמסת פונקציות ..... 9
Friend functions ..... 12
friend class ..... 15
ספריית String ..... 17
חריגות exceptions ..... 20
Static ..... 24
List ..... 27
ירושה ..... 30
רשימה דו-צדדית (עם שני קצוות) ..... 34
פולימורפיזם – ריבוי צורות ..... 36
מחלקות וירטואליות ..... 36
אבסטרקט ..... 39
הידור קבצים ..... 41
Virtual destructors ..... 44
Stack ..... 48
Queue ..... 51
תכנות תבניתי – Templates ..... 53
וקטור ..... 56
עצים ..... 58
עץ חיפוש בינארי ..... 60
איטרטורים ..... 62
Lambda ..... 65
ספריית אלגוריתמים (STL) ..... 69
פונקציות STL ..... 70
קבצים ..... 77
קריאה לא סדרתית מקובץ ..... 83
Void pointer ..... 85

# תכנות מונחה עצמים

בקורס זה נעבוד עם משתנה מטיפוס חדש – class (מחלקה)
המחלקה היא טיפוס שדומה מאוד למבנה שהיכרנו כבר בקורס המבוא, הא מסוגל להכיל משתנים מצורות שונות וכן פונקציות הפועלות על האיברים שבתוך המחלקה.
ההבדל הראשון אותו נראה הוא הגדרת פרטיות המחלקה. בעוד שברירת המחדל של הstruct היא הגדרה פתוחה לכל איבריה, במחלקה ניתן להגדיר רמות שונות של שיתוף המידע.
הבסיס הוא הגדרת **public** – ציבורי, יש גישה לכל החלקים הקיימים בו מהתוכנית הראשית והפונקציות השונות בתוכנית הראשית.
**Private** - החידוש הקיים במחלקה הוא האפשרות להגדיר איזור פרטי שאין אפשרות לגישה ישירה אליו מהתוכנית הראשית. אם רוצים להגדיר או לשנות את האלמנטים שמוגדרים פרטיים, ניתן לעשות זאת רק דרך פונקציות עקיפות (המוגדרות בדרך כלל בחלק הפומבי).
**Protected** – סוג של פרטיות המתייחס לירושה של מחלקות שנשתמש בה בהמשך, וברגע לא רלוונטית.

```
#include <iostream >
using namespace std;
class Rational
{
public:
	int mechane;
	int mone;
	//output
	void print()
	{
		cout << mone << '/' << mechane << '\n';
	}
};
int main()
{
	Rational num1, num2;
	num1.mone = 2;
	num1.mechane = 4;
	num2.mone = 3;
	num2.mechane = 6;
	num1.print();
	num2.print();
	return 0;
}
```

נתייחס לדוגמא למעלה:
**המחלקה** (class)– הגדרת שם המחלקה, נהוג לקרוא בשם עם אות ראשונה גדולה.
**סוג הרשאה** – Public . כל אחד יכול לשנות ולערוך משתנים במחלקה.
**סגירת המחלקה** – כמו בstruct, סוגריים מסולסלים ונקודה פסיק.
**בפונקציה הראשית** – מגדירים משתנים השייכים למחלקה, בצורה הרגילה הנהוגה. כאשר, על מנת להגדיר את החלקים של המחלקה בפני עצמה, מגדירים עם נקודה בין חלקי המשתנה. לדוג': "שם המחלקה.שם המשתנה" (num1.mone)
לאחר שמגדירים את הנתונים שבתוך המחלקה, ניתן להפנות לפעולה הקיימת בהגדרה (method), ואז זה ניתן לעשות רק כאשר המחלקה מוגדרת כPublic. **אם לא יוגדר הpublic תהיה שגיאת ריצה.**
את הקריאה לפעולה מגדירים עם סוגריים והכנסת ערך במקרה הצורך. בדוגמה שלפנינו, ההדפסה היא פונקטיות void של צריכה לקבל ערכים, ולכן הזימון במחלקה נכתב: (num1.print()) וזה מפנה לפונקציה הכתובה בתוך הclass.

כך שלמעשה, כל פעולה המזומנת מהמחלקה מוגדרת על ידי שם האובייקט, ועל ידי הארגומנט המוגדר בו.

**<u>דוגמא 2:</u>**

```
#include <iostream >
using namespace std;
class Rational
{
public:
	int mechane;
	int mone;
	//output
	void print();
	void mult(Rational num);
};
void Rational::print()
{
	cout << mone << "/" << mechane << '\n';
}
void Rational::mult(Rational num)
{
	mone *= num.mone;
	mechane *= num.mechane;
}
int main()
{
	Rational num1, num2;
	num1.mone = 2; num1.mechane = 4;
	num2.mone = 3; num2.mechane = 6;
	cout << "num1="; num1.print();
	cout << "num2="; num2.print();
	num1.mult(num2);
	cout << "num1*mum2=";
	num1.print();
	return 0;
}
```

בתבנית זו, יש זימון של פונקציה של הכפלה (mult) המשתמשת בערכים של המחלקה, ובנתונים מתוך מחלקה אחרת (בשורה המודגשת).

קריאת הפונקציה לוקחת את num1 ומכפילה בו את המונה והמכנה של num2. כך שnum2 נשאר ללא שינוי בכלל וnum1 שזימן את הפונקציה.

בנוסף, יש אפשרות לעשות רק קריאה/חתימה של פונקציה בתוך המחלקה, ואת הפונקציה עצמה לכתוב בין האובייקט לפונקציה הראשית. על מנת להגדיר פונקציה שכזו יש להשתמש בסימן "::" (Scope resolution operator) המגדיר את התחום ההגדרה של הפונקציה אליו מתייחסים.

ההגדרה מורכבת מ: **סוג הפונקציה,** התייחסות **לשם המחלקה**, הגדרת **שם הפונקציה**, הגדרת **הארגומנטים** הרלוונטיים (במידה ויש) לדוג': void Rational ::mult(Rational num)

**יש להקפיד על הגדרה נכונה של כל פונקציה על מנת למנוע טעויות**. ניתן להגדיר לכל סוג מחלקה פונקציית הדפסה שונה, אך יש לוודא מגדירים את שם הפונקציה בהתייחסות נכונה למחלקה המתאימה.

```
//Student.h:
class Student {
private:
	char name[20];
	int grade;
	int marks[10];
	float average;
	int sum();//
public:
	void setGrade();
	int getGrade();
	void setMarks();
	void setAverage();
	float getAverage();
};
#include <iostream>
#include "student.h"
using namespace std;
int main()
{
	Student me;
	// me.grade=3;
	//ERROR: cannot access private
member
	me.setGrade();
	// cout<<me.grade;
	//ERROR: cannot access private
member
	cout << me.getGrade() << endl;
	me.setMarks();
	me.setAverage();
	cout << me.getAverage();
	return 0;
}
```

```
// Student.cpp:
#include <iostream >
#include "student.h"
using namespace std;
void Student::setGrade() {
	int year;
	do {
		cout << "enter your grade ";
		cin >> year;
	} while (year<1 || year>12); //Validity
check
	grade = year;
}
int Student::getGrade() { // בלבד קריאה
	return grade;
}
void Student::setMarks() {
	cout << "enter 10 mark ";
	for (int i = 0; i<10; i++) {
		cin >> marks[i];
		if (marks[i]<0 || marks[i]>100)
		{
			cout << "ERROR\n";
			i--;
		}
	}
}
void Student::setAverage() {
	average = sum() / 10.0;
}
int Student::sum() {
	int s = 0;
	for (int i = 0; i<10; i++)
		s += marks[i];
	return s;
}
float Student::getAverage() {
	return average;
}
```

בקובץ המוגדר בסיומת ".h"(header) יש הצהרה ראשונית של המחלקה, כגון, משתנים ופונקציות רלוונטיות של המחלקה אותם נגדיר. את הקובץ הזה מעבירים הלאה לשימוש על ידי מתכנתים אחרים, כך שניתן יהיה להמשיך ולעבוד בלי להגיע לקובץ המקור- הcpp. כך שבעצם תיקיית קבצים מלאה של תוכנית תכיל: את כל ההצהרות של המחלקות השונות בקבצי h., מימושים של המחלקות והפונקציות הפנימיות שלה בקובץ cpp. וכמובן את הקובץ הראשי, המכיל קישור לקובץ הheader של המחלקה המוגדר על ידי מרכאות - "#include "student.h, ומכריז לתוכנית שעליה לקחת את הקובץ מתיקיית המקור.

בנוסף, ניתן להגדיר את המחלקה בשני חלקים שונים של פרטיות, ותחת כל הגדרה להכניס את הערכים הרלונטיים. כך שבתוך הפונקציות יש חלקים אותם ניתן לערוך, וחלקים אותם ניתן רק לקרוא. הפונקציות הקיימות בתוך המחלקה יכולות להגיע לכל הנתונים בתוך המחלקה, אף אם מדובר במשהו שנמצא מעבר לתחום הסיווג שלהם. אך אין אפשרות לגישה אל נתונים פרטיים, אם הגישה מגיעה מהתוכנית הראשית, או כל פונקצייה המוגדרת מחוץ למחלקה.

בדרך כלל נהוג לעשות פונקציות "get" על מנת לקרוא נתונים, כך שזה פונקציה שמחזירה ערך (int, float) , ופונקציות "set" על מנת להגדיר את הנתונים בתוך המחלקה, שזו פונקציה שלא מחזירה נתונים (void). יש לשים לב - הפונקציה המסכמת את כל הציונים בתוכנית, נמצאת באיזור

המוגדר כפרטי, אליו אין שום גישה מחוץ למחלקה; נהוג שהפונקציות הפנימיות בגון זו, ואחרות שאין בהם שימוש מחוץ למחלקה מוגדרות בתוך טווח הprivate.

```
#include <iostream >
using namespace std;
class Rational
{
public:
	int mechane;
	int mone;
	//constructor
	Rational(int myMone, int myMechane);
	//output
	void print() { cout << mone << "/" << mechane << '\n'; }
};
Rational::Rational
(int myMone, int myMechane) {
	mone = myMone;
	mechane = myMechane;
}
int main()
{
	Rational num1(1, 2),
		num2(3, 4),
		num3(5, 6);
	num1.print();
	num2.print();
	num3.print();
	return 0;
}
```

בתוכנית זו משתמשים בconstractor, האובייקט מוצהר על ידי שני מספרים שלכאורה לא מועברים לשום מקום. הקונסטרקטור הוא סוג של פונקציה, המחולקת להצהרה בתור המחלקה, המכיל את סוג הערך המוחזר. אך חייב לקיים שני תנאים: 1. לא מוגדר בתור טיפוס ערך מוחזר. 2. שם הקונסטרקטור הוא בדיוק כשם המחלקה.

אם יוגדר קונסטרקור בתור טיפוס מוחזר (Int\flaot), הא יפעל כפונקציה רגילה, ולא יגיב בראוי.

קיים גם default constractor, בנאי ש**יכול** שלא לקבל ארגומנטים.

על מנת ליצור default constractor, כאשר מגדירים בקובץ h. את הקונסטרקטור, מכניסים בו גם את ערכי ברירת המחדל. יש לשים לב שעושים את זה תחת הרשאת Public ולא private, אחרת התוכנית הראשית לא תוכל להגיע לערכים הללו.

דוגמא לבניית default constractor:

```
Fraction() // default constructor
{
m_numerator = 0;
m_denominator = 1;
}
```

הגדרת ערכי בסיס לשברים, כך שגם אם יכנס רק ערך המונה, המספר יתקבל פשוט כמספר שלם (מאחר ומתחלק בו), ואם לא יתקבל שום מספר, ברירת המחדל היא ס (0/1).

## דוגמא 1

```
#include <iostream>
using namespace std;
class Vector
{
private:
	int numbers[10];
public:
	Vector();
	int add();
	void add(int);
	void add(int, int);
	void print() const;
};
Vector::Vector()
{
	cout << "enter 10 numbers\n";
	for (int i = 0; i<10; i++)
		cin >> numbers[i];
}
int Vector::add()
{
	int sum = 0;
	for (int i = 0; i<10; i++)
		sum += numbers[i];
	return sum;
}
void Vector::add(int number)
{
	for (int i = 0; i<10; i++)
		numbers[i] += number;
}
void Vector::add(int number, int place)
{
	numbers[place] += number;
}
void Vector::print() const
{
	for (int i = 0; i<10; i++)
		cout << numbers[i] << ' ';
	cout << endl;
}
int main()
{
	Vector vec;
	cout << "add(): " << vec.add() << endl;
	vec.add(3);
	cout << "add(3): "; vec.print();
	vec.add(2, 5);
	cout << "add(2,5): "; vec.print();
	return 0;
}
```

## דוגמא 2

```
void f1(float f);
void f1(double d);
int main)({
int x;
f1(x);
return 0;
}
//ERROR: 'f1' : ambiguous call to overloaded
function
```

## דוגמא 3

```
class Vector {
private:
	int *numbers; int size;
public:
	Vector(int sizeVec = 10);
	Vector(int val, int sizeVec);
	Vector(int* vec, int sizeVec = 10);
	Vector(const Vector&);
	~Vector();
	void print() const;
};
Vector::Vector(int sizeVec) {
	size = sizeVec;
	numbers = new int[size];
	srand((unsigned)time(nullptr));
	for (int i = 0; i<size; i++)
		numbers[i] = rand() % 100;
}
Vector::Vector(int val, int sizeVec) {
	size = sizeVec;
	numbers = new int[size];
	for (int i = 0; i<size; i++)
		numbers[i] = val;
}
Vector::Vector(int* vec, int sizeVec) {
	size = sizeVec;
	numbers = new int[size];
	for (int i = 0; i<size; i++)
		numbers[i] = vec[i];
}
Vector::Vector(const Vector& V) {
	size = V.size;
	numbers = new int[size];
	for (int i = 0; i<size; i++)
		numbers[i] = V.numbers[i];
}
Vector::~Vector() {
	if (size) delete[] numbers;
}
void Vector::print() const {
	for (int i = 0; i<size; i++)
		cout << numbers[i] << ' ';
}
int main()
{
	int nums[10];
	for (int i = 0; i<10; i++)
		nums[i] = i;
	Vector vec1, vec2(6), vec3(4, 8),
		vec4(nums), vec5(vec3);
	cout << "vec1: "; vec1.print();
	cout << "\nvec2: "; vec2.print();
	cout << "\nvec3: "; vec3.print();
	cout << "\nvec4: "; vec4.print();
	cout << "\nvec5: "; vec5.print();
	return 0;
}
```

## דוגמא 4

```
#include <iostream>
using namespace std;
class Point
{
private:
	int x;
	int y;
public:
	//default constructor
	Point(int X = 0, int Y = 0);
	//copy constructor
	Point(const Point& p);
	int getX() const { return x; }
	int getY() const { return y; }
	void print();
};
Point::Point(int X, int Y) :x(X), y(Y)
{ }
Point::Point(const Point& p)
{
	x = p.getX();
	y = p.getY();
}
void Point::print()
{
	cout << '(' << x << ',' << y << ")\n";
}
int main()
{
	Point p1,
		p2(2, 8),
		p3(p2);
	cout << "p1: "; p1.print();
	cout << "p2: "; p2.print();
	cout << "p3: "; p3.print();
	return 0;
```

## העמסת פונקציות

```
//Rational.h
#include <iostream>
using namespace std;
class Rational
{
private:
	int mone;
	int mechane;
public:
	//constructor
	Rational(int myMone = 1, int myMechane
= 1)
		:mone(myMone),
mechane(myMechane) {}
	Rational(const Rational& num)
		:mone(num.getMone()),
mechane(num.getMechane()) {}
	//modify functions
	void setMone(int myMone) { mone =
myMone; }
	void setMechane(int myMechane)
	{
		mechane = myMechane;
	}
	//view functions
	int getMone() const { return mone; }
	int getMechane() const { return
mechane; }
	//operations
	void operator*=(Rational);
	Rational operator*(Rational);
	Rational operator+(Rational);
	Rational operator-(Rational);
	Rational operator/(Rational);
	bool operator==(const Rational&) const;
	Rational& operator=(const Rational &);
	Rational Rational::operator++();
	Rational Rational::operator++(int);
	Rational Rational::operator--();
	Rational Rational::operator--(int);
	//output
	void print() const {
		cout << mone << '/' << mechane
<< endl;
	}
};
//Rational.cpp
```

```
#include " Rational.h"
void Rational::operator *=(Rational num)
{
	setMone(mone*num.getMone());
	setMechane(mechane*num.getMechane());
}
Rational Rational::operator *(Rational num)
{
	Rational tmp;
	tmp.setMone(mone*num.getMone());
	tmp.setMechane(mechane*num.getMechane()
);
	return tmp;
}



Rational Rational::operator +(Rational num)
{
	Rational tmp;
	tmp.setMone(mone*num.getMechane() +
		num.getMone()*mechane);
	tmp.setMechane(mechane*num.getMechane()
);
	return tmp;
}
Rational & Rational::operator =(const Rational
& num)
{
	mone = num.getMone();
	mechane = num.getMechane();
	return *this;
}
bool Rational::operator==(const Rational &
num) const
{
	return (mone == num.mone && mechane ==
		num.mechane);
}
Rational Rational::operator++() {
	mone += mechane;
	return *this;
}
Rational Rational::operator++(int u) {
	Rational temp = *this;
	mone += mechane;
	return temp;
}
#include "Rational.h"
int main() {
	Rational num1(1, 2), num2(1, 4), num3;
	cout << "num1="; num1.print();
	cout << "num2="; num2.print();
	num3 = num1 + num2;  cout <<
"num1+num2="; num3.print();
	num3 = num1*num2;
	cout << "num1*num2="; num3.print();
	return 0;
}
type class_name::operator#(arg_list)
{
	body;
}
```

# פונקציות חופפות והעמסת פונקציות

בדומה להעמסת פונקציות רגילה שאנחנו מכירים, ניתן לממש העמסת פונקציות גם בתוך מחלקה, באשר הפונקציה מוגדרת באותו שם, והשוני בין הפונקציות השונות הוא בסוג בטיפוס המוחזר או בערכים הנשלחים לפונקציה.
בדוגמא שלפנינו, המטרה היא להשתמש במספרים רציונאליים באותו האופן כמו במספרי Int ו float.
על ידי השימוש במחלקה ובהעמסת אופרטורים. כך שניתן יהיה לחבר מספרים בעזרת הסימן '+'.
האופרטורים שאנו מכירים, כמו גם הפונקציה, היא סוג של פעולה מוגדרת. ההבדל ביניהם הוא שהפונקציה מוגדרת על ידי השם שלה, והאופרטור בעצם מבצע את הפעולה הנקראת לעבודה בעזרת הסימנים המוסכמים על ה"אופרנדים" (המשתנים).
כך שבפקודה: num3=num1+num2 שמות המשתנים הם האופרנדים, והפעולות חיבור ושווה הם אופרטורים.
האופרטורים מתחלקים ל: אונריים (unary), ובינאריים (binary) המעמיסים שני אופרנדים. ולעיתים רחוקות קיים גם הטרנרי (terenary) הפועל על 3 אופרנדים[1].
האופרנד הראשון בפקודה הוא האופרנד המזמן את פקודת האופרטור, כך אם נתייחס לפקודה למעלה בקריאה בתוך מחלקה נתייחס אליו כ num1.plus(num2) והתוצאה מוחזרת לNum3.
ובאופן מלא נאמר ש-

| | | |
|---|---|---|
| Num1. | plus | (num2) |
| Calling object | function | argument |

ובאותו אופן במחלקה:

```
Class Rational
{
        Public:
Rational plus(const Rational &num)const;
}
```

יש רשימה של סימנים אותם ניתן להגדיר כOperator כמו חיבור וכפל, והקומפיילר יודע לקחת את הקדימויות בין כפל וחיבור, וכן את ההבדל בין כפל לבין כוכבית של מצביע.

**:mone(myMone)**

בצורה כזו,ניתן לאתחל מספר שדות בתוך המחלקה, כאשר מוסיפים פסיק בין משתנה בו משתמשים. כך ששולחים את המשתנה המאותחל כארגומנט לתוך השדה. צורה זו הינה יעילה יותר , מאחר ואם משתמשים בצורה הרגילה של mone=myMone וכו' השדות נוצרים עם ערכי זבל ורק אז מוגדרים בערכיהם, אך פה ברגע שהאובייקט מוגדר, ישירות הוא מוגדר עם הערך המותחל. דרך זו נקראת **"Constructor initialization list "** (בנאי רשימה).

בעזרת הגדרת שם הפונקציות באופרטור, ניתן להכניס בפונקציה עצמה הרחבה של האופרטור הרגיל, כך שפה משתמשים באופרטור '=' לחבר שברים על ידי הכפלת מונה ומכנה אחד בשני, והעברת התוצאה למכנה משותף.

Rational & Rational::**operator** =( **const** Rational & num)

למה זה נשלח by refernce ולא value?
ברגע ששולחים בהעתק, נוצר "קופי קונסטראקטור", ועל ידי זה כל העתקה והחזרה של אובייקט משתמשת בעוד זיכרון ובעוד זמן.

[1] האופרנד היחיד הזה הוא בדיקת התנאי XX?XX:XX הבודקת נכונות אופרנד ומחליט לאן ללכת על פי התוצאה

## אופרטורים אונריים

**++** בכלל שכבר למדנו בעבר, יש לשים לב שיש הבדל בין ++ המופיע לפני הערך (הגדלה תחילית pre-increment) לבין אחד המופיע בסוף המשתנה (הגדלה בדיעבד post increment).

```
Int =5,y;
y = ++x; //x=6 y=6
y = x++ //x=6 y=5
```

באשר האופרנד מופיע לפני המשתנה הוא קודם מקדם אותו בו ורק אז מבצע את ההשמה לתוך הY.
ובאשר הוא אחרי הX, הוא קודם מכניס את הערך הנוכחי בY ורק אז מקדם את הX בו.

```
Rational Rational::operator++() {
	mone += mechane;
	return *this;
}
Rational Rational::operator++(int u) {
	Rational temp = *this;
	mone += mechane;
	return temp;
}
```

הפונקציה הראשונה מתייחסת ל x++, ואילו השני מציב את הערך ורק אחרי זה מקדם את אותו ומחזיר את הטמפ.

```
#include <iostream >
using namespace std;
class Rational
{
private:
	int mone;
	int mechane;
public:
	//constructor
	Rational(int Mone = 1, int Mechane = 1)
		:mone(Mone), mechane(Mechane) {}
	Rational(Rational& num) :
mone(num.getMone()), mechane(num.getMechane())
{}
	//modify functions
	void setMone(int Mone) { mone = Mone; }
	void setMechane(int Mechane) { mechane
= Mechane; }
	//view functions
	int getMone() const { return mone; }
	int getMechane() const { return
mechane; }
	//operations
	Rational operator+(Rational);
	Rational operator-(Rational);
	Rational operator*(Rational);
	Rational operator/(Rational);
	Rational& operator=(const Rational&);
	bool operator==(Rational);

	friend Rational operator *(Rational,
int);
	friend Rational operator *(int,
Rational);
	//input/output
	friend ostream& operator<<(ostream& os,
Rational num);
	friend istream& operator >> (istream&
is, Rational& num);
};

Rational Rational::operator +(Rational num)
{
	Rational tmp;
tmp.setMone(mone*num.getMechane() +
		num.getMone()*mechane);
	tmp.setMechane(mechane*
		num.getMechane());
	return tmp;
}
Rational Rational::operator -(Rational num)
{
	Rational tmp;
	tmp.setMone(mone*num.getMechane() -
		num.getMone()*mechane);
	tmp.setMechane(mechane*num.getMechane()
);
	return tmp;
}
Rational Rational::operator *(Rational num)
{
	Rational tmp;
	tmp.setMone(mone*num.getMone());
	tmp.setMechane(mechane*num.getMechane()
);
	return tmp;
}

Rational Rational::operator /(Rational num)
{
	Rational tmp;
	tmp.setMone(mone*num.getMechane());
	tmp.setMechane(mechane*num.getMone());
	return tmp;
}

Rational& Rational::operator =(const Rational&
num)
{
	mone = num.getMone();
	mechane = num.getMechane();
	return *this;
}

bool Rational::operator ==(Rational num)
{
	return mone == num.getMone() &&
		mechane == num.getMechane();
}
Rational operator *(Rational rat, int num)
{
	Rational tmp;
	tmp.setMone(rat.mone*num);
	tmp.setMechane(rat.mechane*num);
	return tmp;
}
Rational operator *(int num, Rational rat)
{
	return operator*(rat, num);
}
ostream& operator<<(ostream& os, Rational num)
{
	os << num.mone;
	os << '/';
	os << num.mechane;
	os << endl;
	return os;
}
istream& operator >> (istream& is, Rational&
num)
{
	is >> num.mone;
	char slash;
	is >> slash;
	is >> num.mechane;
	return is;
}
```

# Friend functions

```
friend ostream& operator<<(ostream& os, type name);
friend istream& operator >> (istream& is, type & name);
ClassName & ClassName::operator =(const ClassName & name)
```

פונקציות friend הינן פונקציות בעלות הרשאה מיוחדת. קודם הסברנו שפונקציות שהן חיצוניות למחלקה, לא יכולות לגשת לתוך החלקים המוגדרים תחת "פרטי", כמו משתנים וערכים, אך אם מגדירים פונקציה של friend, ניתן לגשת גם מהתכנית הראשית אל כל השדות (כמובן מה שמוגדר בתוך הפונקציה).
השימושים הנפוצים לפונקציות אלו, לפחות בסדנא, הם העמסות לאופרטורים של קלט ופלט, שבמקרה רגיל מקבלות רק ערך אחד, אך במחלקה צריכות להתייחס למספר משתנים שונים וטיפוסים שונים, כך שאופרטור פלט לא יספק את התוצאה הרצויה.
במקרים אלו, אנחנו מעמיסים את האופרטור בתוך הפונקציה, ומגדירים את הפלט פי הסדר שאנחנו רוצים, או לחילופין בקלט, אנחנו מקבלים בסדר הרצוי ויכולים לשלב גם פלט לפי הנדרש.
בעזרת פונקציות כאלה, ניתן גם ליצור פקודות שיעבדו בצורה שונה מהרגיל, כגון :
Rational n1(2,5);
N2=3*n1;
בעזרת כתיבת הפונקציות הבאות והגדרה שלהם כמו בדוגמא.

```
friend Rational operator *(Rational, int);
friend Rational operator *(int, Rational);
```

הפונקציה מוגדרת בצורה גלובלית

**פונקציות קלט ופלט**

```
friend ostream& operator <<(ostream& os, Rational num);
friend istream& operator >> (istream& is, Rational& num);
```

הפונקציות האלו חייבות להיות מועברות על ידי & בשביל להחזיר את הפלט המדויק שאותו יהיה אפשר לשרשר., אחרת יהיה שכפול של הערכים. וכך ניתן לפרוט פעולה שלמה למספר פעולות בסיסיות, כמו בדוגמה לפנינו:

```
ostream& operator<<(ostream& os, Rational num)
{
	os << num.mone;
	os << '/';
	os << num.mechane;
	os << endl;
	return os;
}
```

ניתן גם לכתוב את זה בצורה קצרה יותר –במקום לכתוב num.mone אפשר להחזיר:

```
Ostrea & operator << (ostream & os, const Rational & num)
{
 Return os<< num.mone << "\" << num.mechane<< endl;
}
```

וכן ניתן לעשות הצבה דומה על ידי:

```
Istream & operator >> (istream is, Rational & num)
```

```
{
 Char delim;
 Return is >> num.mone >> delim >> num/mechane;

}
```

את ההצבה ניתן לעשות בזכות פונקציית הfriend. (רק יש לזכור להגדיר את כל זה במחלקה כפונקציית friend, בתוך קובץ ההגדרה של ה cpp/main), וגם צריך שהפונקציה תחזיר את הערך המוגדר – במקרה הזה os/is.

```
#include <iostream >
using namespace std;
class CSquare;
class CRectangle
{
private:
	int width, height;
public:
	int area(void)
	{
		return (width * height);
	}
	void convert(CSquare a);
};
class CSquare
{
private:
	int side;
public:
	void set_side(int a)
	{
		side = a;
	}
	friend class CRectangle;
};
void CRectangle::convert(CSquare a)
{
	width = a.side;
	height = a.side;
}
int main()
{
	CSquare sqr; ←
	CRectangle rect;
	sqr.set_side(4);
	rect.convert(sqr);
	cout << rect.area();
	return 0;
}
```

# friend class

```
class B {
 friend class A; // A is a friend of B
private:
 int i;
};
class A {
public:
 A(B b) {
 b.i = 0; // legal access due to friendship
 }
};
```

**forward declaration**
בראש התוכנית, אנו מכריזים על מחלקה עם שם, אותה לא הגדרנו עדיין. במקרה כזה, הקומפיילר לא מגדיר פה כבר את המחלקה, וגם לא מקפיץ הודעת שגיאה, אלא רק מתייחס לשם המוגדר שברור שיוגדר בתוכנית בהמשך.

פעולה פחות נפוצה היא **friend class** –
אם אנחנו רוצים לקבל גישה ממחלקה אחת לאחרת, בלי להסתבך עם דיני ירושה, ניתן להגדיר בשדה המשתנים של המחלקה אליה רוצים להתייחס, את שם המחלקה שיכולה לגשת אליה וכך יש גישה **חד כיוונית** ממחלקה אחת לאחרת.
במקרה לפנינו, במחלקת הריבוע אנחנו מגדירים שהיא חברה לפונקציית מלבן – המלבן יכול לקחת ממנה מה שצריך.
בנוסף יש לוודא שאנחנו עושים בתוכנית הראשית הגדרה מראש של הפונקציות בסדר כזה שקודם כל מגדירים את המחלקה ממנה לוקחים משתנים/פונקציות, ואחר כך את המחלקה ה"חברה" שלוקחת את מה שהיא רוצה.
הפונקציה convert מקבלת את ערכי הריבוע ומגדירה רוחב וגובה למלבן על פי הצלע המוגדר. יש לשים לב, שהפונקציה ניגשת ישירות ולא על ידי set/get.
ניתן לעשות פונקציית friend כפולה כך שאחד "ידידותי" לשני אך לא לשלישי. רק מה שמוגדר בתוך המחלקה הוא ידידותי למתייחס אליו ולא יותר מזה.

```
// MyString.h
#include <iostream>
#include <string>
using namespace std;
class MyString
{
private:
	char * str;
	void setString(const char* s);
public:
	// constructor.
	MyString(char* s = nullptr);
	MyString(const MyString & s);
	~MyString();
	// view function.
	char* getString() const;
	// modify function.
	MyString & operator =(const MyString
&);
	// operators
	bool operator==(const MyString &)
const;
	MyString operator+(const MyString &);
	MyString operator* (int);
	int length() const;
	// print
	void print() const;
};
// MyString.cpp
#include " MyString.h"
```

```
MyString & MyString::
operator=(const MyString & s)
{
	if (str)
		delete[] str;
	setString(s.getString());
	return *this;
}
MyString MyString::
operator+(const MyString & s)
{
	int sizeI = strlen(str);
	int sizeII = strlen(s.getString());
	char* temp = new char[sizeI + sizeII +
1];
	strcpy_s(temp, sizeI + 1, str);
	trcpy_s(temp + sizeI, sizeII + 1,
s.getString());
	MyString x(temp);
	return x;
}
MyString MyString::operator*(const int num)
{
```

```
MyString::MyString(char* s)
{
	setString(s);
}
MyString::MyString(const MyString & s)
{
	setString(s.getString());
}
MyString::~MyString()
{
	if (str)
		delete[] str;
	str = nullptr;
}
char* MyString::getString() const
{
	return str;
}
void MyString::setString(const char * s)
{
	if (s)
	{
		int len = strlen(s) + 1;
		str = new char[len];
		strcpy_s(str, len, s);
	}
	else str = nullptr;
}
```

```
	char* temp;
	int len = strlen(str);
	temp = new char[len*num + 1];
for (int i = 0; i < num; i++)
	strcpy_s(temp + i*len, len + 1, str);
	MyString s(temp);
	return s;
}
bool MyString::
operator==(const MyString & s) const
{
	return !strcmp(str, s.getString());
}
int MyString::length() const
{
	return strlen(str);
}
void MyString::print() const
{
	if (str) cout << str << endl;
}
```

# ספריית String

במשך הקורס נעבור על מספר ספריות אותן ניתן להכליל בתוכנית ולהשתמש בהם על מנת לעשות פעולות פשוטות שעד עכשיו היינו צריכים לכתוב/להעתיק שורות קוד חדשות בכל הזדמנות. ספריות אלו מופיעות בהמשך בספריית הלגוריתמים/STL. בדוגמא כאן יש עבודה עם ספריית string (מחרוזות), בה עבדנו כבר בעבר ומכילה פונקציות לעבודה עם מחרוזות בצורה פשוטה.
במחלקה שהגדרנו mystring יש אפשרויות להשתמש במחרוזות בכל מיני צורות, ואפילו בצורה של הכפלת המחרוזת כמה פעמים (*Operator). כל הפעולות המתבצעות בתוך הפונקציות, משתמשות בפונקציות פנימיות של ספריית string שכבר נתקלנו בה בעבר, ובדאי לזכור את המימושים של כל פונקצית ספרייה.
הפונקציות השונות נותנות לעשות שדה שהוא פויינטר, שמצביע על מחרוזת או זיכרון דינאמי, שלא משתחרר עם שחרור הפויינטר. אך איך ניתן לשחרר את הזיכרון הדינאמי עם הפויינטר בצורה אוטומטית?
אחת מהפונקציות הבסיסיות של המחלקה נקראת destructor – הורס- שבדומה לבנאי שדואג לבניית בסיס של המחלקה, הdestructor עובד על "הריסת" המחלקה ומחיקת המידע הדינאמי.
הגדרת הדיסטרקטור נעשית על ידי קו גלי ()className~ ללא שום הגדרה של משתנה.
בעת היציאה מהחלקה הדיסטרקטור נכנס לפעולה וממלא את תפקידו ומוחק את כל המידע הדינמי.
הפקודה של הדיסטקרטור מחפשת אם יש מידע שלא השתחרר ומפעליה עליו פקודת []delete והגדרה שלו בnull.
ניתן גם להגדיר שחרור ספציפי ומחיקה של מידע לפי הצורך, כאשר ניגשים בין הסוגריים המסולסלים ומגדירים ידנית.

פקודות string שימושיות:

| Name | תיאור | syntax |
|---|---|---|
| Strcpy | העתקה לתוך מחרוזת ודריסת מה שהיה | char * strcpy ( char * destination, const char * source ); |
| strncpy | העתקה של כמות תויים מוגדרת למחרוזת חדשה | char * strncpy ( char * destination, const char * source, size_t num ); |
| Strcat(strncat) | העתקת תוים בקצה מחרוזת | char * strcat ( char * destination, const char * source ); |
| Strcmp | השוואת מחרוזת(1 – ראשון גדול. 0 שווה. -1 שני גדול | int strcmp ( const char * str1, const char * str2 ); |
| strlen | אורך מחרוזת לא כולל NULL | Int x = strlen(str); |
| strlwr | שינוי התוים לאותיות קטנות | strlwr(string) |
| Strupr | שינוי התוים לאותיות גדולות | strupr(string) |

```
#include<iostream>
using namespace std;
int main()
{
	int numerator = 10;
	int denominator = 0;
	int div = numerator / denominator;
	cout << "This text will not be
printed.";
	return 0;
}
!!!An unhandled exception of type
'System.DivideByZeroException' occurred in
exceptions.exe
```

---

**דוגמא 2 - catch**

```
#include <iostream>
using namespace std;
int main() {
	try {
		throw 20;
	}
	catch (int e) {
		cout << "Exception # " << e
			<< " occurred";
	}
	return 0;
}
```

---

**דוגמא 3 – העמסת חריגות**

```
#include<iostream>
using namespace std;
int level3() {
	cout << "Level 3 beginning.\n";
	int num1, num2;
	cout << "enter 2 numbers\n";
	cin >> num1 >> num2;
	if (!num2)
		throw "dividing by zero ";
	int result = num1 / num2;
	cout << "Level 3 ending.\n";
	return result;
}
void level2() {
	cout << "Level 2 beginning.\n";
	cout << level3();
	cout << "Level 2 ending.\n";
}
void level1() {
	cout << "Level 1 beginning.\n";
	try {
		level2();
	}
	catch (char*problem) {
		cout << "\nException message: "
			<< problem;
	}
	cout << "\nLevel 1 ending." << endl;
}
int main() {
	cout << "Program beginning.\n";
	level1();
	cout << "Program ending." << endl;
}
/*Output :
Program beginning.
Level 1 beginning.
Level 2 beginning.
Level 3 beginning.
enter 2 numbers : 5 0
Exception message : dividing by zero
Level 1 ending.
Program ending.*/
```

## דוגמא 4 – ריבוי תפיסות

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

class Product {
private:
	char name[10];
	int quantity;
	float price;
public:
	Product() {}
	void init(char* n, int q, float p);
	friend ostream& operator<<
		(ostream& os, Product p);
};
void Product::init(char* n, int q, float p) {
	if (strlen(n)>9)
		throw "error: name too long\n";
	strcpy(name, n);
	if (q<0 || q>100) throw q;
	quantity = q;
	if (p<0) throw p;
	price = p;
}
ostream& operator<<(ostream& os, Product p) {
	os << p.name << '\t';
	os << p.quantity << '\t';
	os << p.price << '\n';
	return os;
}
```

```
int main() {
	char name[100]; float price;
	int quantity, i; Product store[3];
	for (i = 0; i<3; i++) {
		try {

cout << "enter a product number " << i << " ";
	cin >> name >> quantity >> price;
	store[i].init(name, quantity, price);
	}
	catch (char* msg) {
		cout << msg;
		i--;
	}
	catch (float f) {
	cout << "error: price can't be negative\n";
		i--;
	}
	catch (int num) {
		if (num<0)
	cout << "error: quantity must be 0 or more\n";
		else
	cout << "error: quantity available up to 100\n";
		i--;
	}
	catch (...) {
	cout << "error: unknown error\n";
		i--;
	}
	}
for (int i = 0; i<3; i++) cout << store[i];
	return 0;
}
```

### דוגמא 5 – מחלקת exception עם ירושה

```
//standard exceptions
#include <iostream>
#include <exception>
using namespace std;
class myexception : public exception {
	virtual const char* what() const
throw()
	{
		return "My exception happened";
	}
} myex;
int main() {
	try {
		int y = 0;
		throw myex;
		int x = 4 / y;
	}
	catch (exception& e) {
		cout << e.what() << endl;
	}
	return 0;
```

# חריגות exceptions

כאשר התוכנית עובדת על פי סדר הפעולות המוגדר, יכול להיות שאחת מהפעולות שאנחנו רוצים לבצע, לא ניתנת לביצוע מכל מיני סיבות, למשל:
הזנת שני מספרים וחלוקה אחד בשני – אם המכנה = 0 לא ניתן לחלק את השבר.
העתקת קובץ לקובץ (אולי עם עיבוד של הקלט ביניים)- אם אחד הקבצים לא קיים במיקום הנדרש, התוכנית לא עובדת כמו שצריך.
קריאת נתוני תקשורת (אינטרנט וכדו'), מאחר ואנחנו תלויים בקיימות של הרשת.
לסיכום: מכל מיני סיבות יכול להיות כשלים בתוכנית **האמיתית** מכל מיני נתונים ומשתנים בעולם **האמיתי**.
אך הדרישה שלנו כמתכנתים היא לדאוג שלא תהיה שום קריסה ושהתוכנית תמשיך ותעבוד בכל מצב, או לפחות תגיב בהתאם לתקלה עד כמה שאפשר.
כל מצב כזה שאנו מבצעים פעולה חלופית (בדרך כלל הודעת שגיאה ופלט חדש) נקרא "חריגה" – Exception.
יש לציין, שאנחנו מדברים לא על שגיאה של התוכנית עצמה, אלא התמודדות עם מצבים חריגים מחוץ לתוכנית.

בדוגמא הראשונה, אין בכלל טיפול בחריגה, וברגע שמזהים שגיאה התוכנית יוצאת.
בדוגמא ה2 – רואים טיפול בחריגה על ידי הפקודה **try** – מגדירים איזור שהוא "ניסיוני", וברגע שנתקלים בפקודת **throw** בודקים את התנאי, ואם הוא לא מתקיים, זורקים את הפונקציה לקצה המסגרת כך ששום פקודה לא תתבצע מאותו רגע עד לסוף המסגרת.
התוכנית נזרקת עד שהיא מגיעה לפקודת "תפיסה" **catch(type name)** בה מגדירים פלט או פקודת התראה על כך שקפצה פה חריגה. כמובן כל זאת בהנחה שהפרמטר הנשלח מתאים להגדרת המשתנה בcatch. (20 ייתפס בint, אך 20.5 לא ייכנס לint ויקריס את התוכנית)
כמו כן ניתן להגדיר catch עם "(...)" כך שיכניס אליו כל ערך שנזרק הצידה ללא כל הגדרה מיוחדת.
בדוגמא 3 מתבצעת פקודה הבודקת אם מתבצע חילוק בס, אם יש 0 הפקודה קופצת לcatch שמוציא הודעת שגיאה, ומבטל את התוכנית. ברגע שיש קפיצה לthrow מתבצע דילוג מעל כל הפונקציות, לא משנה כמה עמוק זה קבור, בתוך פונקציה אחת בתוך השניה, הוא ישר יוצא החוצה, אל התפיסה.
למעשה התוכנית הראשית (main) נבנה בצורת מחסנית זימונים "call stack" פונקציה על גבי פונקציה כאשר ברגע שקופץ הthrow הוא מחפש באותה מחסנית של הפונקציה וברגע שהוא לא מוצא שם תפיסה, הוא מוציא את הפונקציה ועובר לשלב הבא בזימונים עד שיימצא תפיסה מתאימה על פי ההגדרה ורק שם הוא ימשיך לרוץ ולצאת לפונקציות המתאימות. כך שאם יגיע לסביבת ההרצה "Runtime Environment" ולא התבצעה שום פעולה התוכנה תקרוס.

| |
|---|
| ... |
| Level 3 |
| Level 2 |
| Level 1 |
| Main |

**<u>ריבוי תפיסות</u>**

<u>דוגמא 4</u> – נניח שיש מחלקה של מוצר product, המכיל מספר משתנים שונים. חלקם מספרים וחלקם תווים. וגם ביניהם יכו להתבצע חלוקה של Int (כמות) וdouble (מחיר). הפונקציה מקבלת ערכים עבור השדות האמורים, ומבצעת בדיקות תקינות עבור כל הערכים.
בכל מצב בו תהיה חריגה באחד הערכים, הפונקציה תיזרק אל ההתראה המתאימה ומשם תוציא הודעת שגיאה אל המשתמש ורק אז תמשיך הלאה.

כדאי לשים לב – מאחר ואנו נמצאים בתוך לולאת for המזיזה את הערך בעזרת ++, אם נחזור בצורה חדשה הערך ייכנס בתור מוצר חדש, כאשר אנו רוצים לבצע את הקליטה על **אותו** המוצר ולא על אחד חדש, ולכן בסוף הcatch מוסיפים – על מנת לאפס את הfor.

ברגע שיש בנאי ברירת מחדל עם פרמטרים, בעצם, יש התעלמות מהחריגות, ותמיד יהיו נתונים תקינים.

ברגע שונים אותו ללא שום הגדרה, התוכנית תתקמפל ותשאיר את האפשרויות לבדיקה של החריגות ויציאה במקרה הצורך.

דוגמא 5

הגדרת ספרית חריגות exception ואפשרות להגדת ()what בתוך המחלקה של הירושה[2], כך שמגדירים תוית של תפיסה, בצורה של ריבוי צורות.

[2] כל נושא הירושה יילמד בהמשך, אך כבר עכשיו נעבור על התוכנית ונבין אתה לגמרי בעוד מספר שיעורים.

## דוגמא 1 – הגדרת משתנה סטטי בפונקציה

```
#include <iostream>
using namespace std;
void showStaticValue(int n)
{
 static int value = 0;
 value += n;
 cout << "Static value is " << value << endl;
}
int main()
{
	for (int i = 0; i < 5; i++)
		showStaticValue(i);
	return 0;
}
//Output:
//Static value is 0
//Static value is 1
//Static value is 3
//Static value is 6
//Static value is 10
```

## דוגמא 2

```
#include <iostream>
#include<string>
using namespace std;
class student {
private:
	char name[20];
	int grade;
	static int numOfStudents;
	static int numOfFails;
public:
	student(char* n);
	void setGrade(int g);
	void percentageOfFailers();
};
int student::numOfStudents = 0;
int student::numOfFails = 0;
student::student(char* n) {
	strcpy_s(name, n);
	numOfStudents++;
}
void student::setGrade(int g) {
	grade = g;
	cout << grade << endl;
	if (grade<55)
		numOfFails++;
}
void student::percentageOfFailers() {
	cout << (float)numOfFails /
numOfStudents * 100;
	cout << "% of the students failed\n";
}
```

```
int main()
{
	char name[15];
	int mark;
	for (int i = 0; i < 10; i++)
	{
		cout << "enter students
			name and grade\n";
			cin >> name >> mark;
		student s(name);
		s.setGrade(mark);
		s.percentageOfFailers();
	}
	return 0;
}
```

## דוגמא 3 - ספריית פונקציות בקריאה סטטית ללא שימוש במשתנים

```
#include <iostream>
using namespace std;
class Calculator {
public:
	static int add(int, int);
	static int sub(int, int);
	static int mult(int, int);
	static float div(int, int);
};
int Calculator::add(int a, int b) {
	return a + b;
}
int Calculator::sub(int a, int b) {
	return a - b;
}
int Calculator::mult(int a, int b) {
	return a*b;
}
float Calculator::div(int a, int b) {
	if (!b) throw "cannot divid by zero\n";
	return (float)a / b;
}
enum options {
	STOP, ADD, SUB, MULT, DIV
};
int main() {
	int op, x, y;
	cout << "enter your choice\n";
	cin >> op;
	while (op) {
		cout << "enter 2 values\n";
		cin >> x >> y;
		switch (op) {
		case ADD:
			cout << x << '+' << y << '=';
			cout << Calculator::add(x, y) << endl;
			break;
		case SUB:
			cout << x << '-' << y << '=';
			cout << Calculator::sub(x, y) << endl;
			break;
		case MULT:
			cout << x << '*' << y << '=';
			cout << Calculator::mult(x, y) << endl;
			break;
		case DIV:try {
				float z = Calculator::div(x, y);
				cout << x << '/' << y << '=' << z << endl;
				}
				catch (char* msg) {
					cout << msg;
				}
				break;
		default: "no such option\n";
		};
		cout << "enter your choice\n";
		cin >> op;
	}
}
```

# Static

זיכרון סטאטי, כמו שעבדנו עליו עד עכשיו הוא זיכרון שמוקדש לפונקציה ונמחק בסוף הריצה. כך שאם בתוך אחת מהפונקציות הוגדר משתנה סטטי רגיל (instance variables) , הוא נמחק בעת החזרה לתוכנית הראשית או במעבר לפונקציה חדשה.
אך אם נגדיר את האתחול בסטטי (class variables), ניתן לשנות את הערך והוא יישמר גם במעבר לפונקציה אחרת.
בדוגמא 1 קיימת לנו פונקציה שבתוכה יש משתנה סטטי המאותחל כ-0., מתבצע בו שינוי, ומדפיסים אותו. אחרי שחוזרים לתוכנית הראשית, אנחנו נשלחים בחזרה לפונקציה, כאשר בשונה מפונקציה רגילה, המשתנה לא מאופס מחדש, אלא ממשיך מאותו ערך אותו הוא קיבל בפי האחרונה שנעשה בו שינוי.
דוגמא 2 כבר מתייחסת להגדרת משתנה סטטי בתוך המחלקה, הנספר בכל פעם שמתקבל ערך חדש ומחזיר ערכים שמתייחסים לערך הסטטי בפונקציות שונות. יש לשים לב, שאת המשתנה הסטטי מגדירים בקובץ המימוש – cpp – ולא בקובץ הheader.
דוגמא 3 מתייחסת לבניה של מחלקה שלימה שאין בה שום שימוש המועיל לתוכנית מלבד הערכים הסטטים שלה. ניתן לבנות מחלקה כזו, ולגשת לכל המשתנים/פונקציות הסטטיות שלה גם ללא שהגדרנו את המחלקה עצמה בתוך התוכנית. אם רוצים לגשת לפונקציה ניתן לזמן אותו על ידי '::'[3]

למעשה, כמו בדוגמא 3 לפעמים יוצא שמגדירים קובץ cpp רק בשביל להגדיר את המשתנים הסטטיים בפני עצמם. במקרה כזה, עוד לפני שמתחילים להצהיר על משתנים הם כבר נוצרים, וניתן להתחיל לעבוד עם המשתנים הסטטיים גם ללא שום אובייקט מזמן.

[3] Calculator::add(x, y) – במקרה לפנינו, קוראים למחלקת המחשבון, ומפעילים את הפוננקציה הסטטית בעזרת שני הערכים שהתקבלו.

```
//-----------------------------------------
// class List
// arbitrary size Lists
// permits insertion and removal
// only from the front of the List
//-----------------------------------------
class List
{
protected:
	//-------------------------------------
	// inner class link
	// a single element for the linked List
	//-------------------------------------
	class Link
	{
	 public:
	 // constructor
	 Link(int linkValue, Link * nextPtr);
	 Link(const Link &);
	 // data areas
	 int value;
	 Link * next;
	}; //end of class Link
 public:
	// constructors
	List();
	List(const List&);
	~List();
	// operations
	void add(int value);
	int firstElement() const;
	bool search(const int &value) const;
	bool isEmpty() const;
	void removeFirst();
	void clear();
protected:
	// data field
	Link* head;
};
//-----------------------------------------
// class Link implementation
//-----------------------------------------
List::Link::Link(int val, Link* nxt) :
value(val), next(nxt) {}
List::Link::Link(const Link& source) :
value(source.value), next(source.next) {}
//-----------------------------------------
// class List implementation
//-----------------------------------------
List:: List (): head(nullptr)
{
	// no further initialization
}
List::List(const List &l)
{
	Link *src, *trg;
	if (l.head == nullptr)
		head = nullptr;
	else
	{
		head = new Link((l.head)->value,
nullptr);
		src = l.head;
		trg = head;
		while (src->next != nullptr)
		{
			trg->next = new Link
			((src->next)->value,
nullptr);
			src = src->next;
			trg = trg->next;
		}
	}
}
List::~List()
{
	clear();
}
void List::clear()
{
	// empty all elements from the List
	Link* next;
	for (Link * p = head; p != nullptr; p =
next)
	{
	// delete the element pointed to by p
		next = p->next;
		p->next = nullptr;
		delete p;
	}
// mark that the List contains no elements 87.
head= nullptr;
}
bool List::isEmpty() const
{
// test to see if the List is empty
// List is empty if the pointer to the head
	// Link is null
	return head == nullptr;

}

void List::add(int val)
{
//Add a new value to the front of a Linked
List
	head = new Link(val, head); 101. if
(head == nullptr)
		throw "failed in memory
allocation";
}
```

```
int List::firstElement() const
{
	// return first value in List
	if (isEmpty())
		throw "the List is empty, no
first Element";
	return head->value;
}
bool List::search(const int &val) const
{
	// loop to test each element
	for (Link* p = head; p != nullptr; p =
p->next)
		if (val == p->value)
			return true;
	// not found
	return false;
}
void List::removeFirst()
{
	// make sure there is a first element
	if (isEmpty())
		throw "the List is empty, no
Elements to remove";
	// save pointer to the removed node
	Link* p = head;
	// reassign the first node
	head = p->next; 129. p->next = nullptr;
	// recover memory used by the first
element
	delete p;
}
```

USE:

```
#include <iostream>
#include "List.h"
using namespace std;
int main()
{
	int element;
	List ls1, ls2;
	try
	{
		for (int i = 0; i < 5; i++)
		{
			ls1.add(i);
			cout << i << " ";
		}
		ls1.removeFirst();
		for (int i = 0; i < 4; i++)
		{
			element =
ls1.firstElement();
			cout << element << " ";
			ls2.add(element);
		}
		cout << endl;
		cout << ((ls2.search(4)) ? "ls2
includes 4" :
			"ls2 doesn't include 4")
<< endl;
		cout << ((ls2.search(3)) ? "ls2
includes 3" :
			"ls2 doesn't include 3")
<< endl;
		ls2.removeFirst();
		cout << ((ls2.search(3)) ? "ls2
includes 3" :
			"ls2 doesn't include 3")
<< endl;
	}
	catch (char* problem)
	{
		cout << problem;
	}
	return 0;
}
```

# List

בתוכנית שלפנינו מוצג שימוש במבנה נתונים "רשימה" אותו למדנו בקורס. העץ עצמו מכיל שני חלקים חשובים: 1. מחלקה פנימית בשם "Link" – כל לינק שכזה, יכיל את הערך המרומי שלו ומצביע ללינק הבא.
2. פונקציות למימוש עץ.
הגדרת הלינק נעשית תחת הגדרת פרטיות protrected עליה נרחיב בענייני הירושה, אך בכלל, זו הגדרה שמכילה את כל מה שמתחתה בנגיש למחלקה עצמה ולכל המחלקות היורשות ממנה. המשמעות היא שהList יכול לגשת לתוך הLink והתוכנית הראשית לא מסוגלת לגעת במחלקה הפנימית.
למה צריך להגדיר את Link בתוך הlist?
אנחנו כותבים את התוכנית של הרשימה המקושרת – List, זאת אומרת שבכל מקום שנרצה להשתמש במחלקה הזאת, נרצה לשמור אוסף של אובייקטים מכל סוג (בדוגמא שלנו, שמירה של מספרים שלמים), האובייקט צריך להיות מסוגל להוסיף ולמחוק מספרים, למצוא מספרים ולהדפיסם. בתוכנית הראשית עצמה, אין לנו שום עניין בדרך שבה הרשימה עובדת ומה מקשר למה, אלא רק בשינוי והצגת הערכים. לכן כל האפשרויות האלו יוגדרו בתוך הרשימה עצמה ויהיו חלק אינטגרלי ממנה. עקרון זה הוא עיקרון חשוב מאוד בתכנות – אין צורך לחשוף את כל התוכנית, והלקוח לא צריך לדעת מה יש "מתחת למכסה המנוע", אלא לדעת רק מהmain מה האפשרויות שהוא מסוגל לעשות.
יש לזכור שבעצם אין לנו גישה ישירה לתוך הלינקים. כל שינוי ודחיסה ייעשה רק דרך הפונקציות המסוגלות לדחוף לינקים חדשים. המקסימום שאנחנו יכולים הוא להגדיר משתני עזר חדשים של הלינק ולעבוד איתם.
בדוגמא זו יש 6 אפשרויות לפונקציות: 1. הוספה (**add**) 2. **First element** – החזרה של הערך הראשון ברשימה 3. **Search –** מקבלת ערך ומחזירה האם הערך קיים ברשימה. 4. **isEmpty** – האם הרשימה ריקה. 5. **removeFirst** – מחיקה של הערך הראשון ברשימה. 6. **Clear** – מחיקה של כל הערכים ברשימה[4]
לא תמיד קיים ערך ראשון – לפעמים הרשימה ריקה ויש צורך בזריקה של חריגה אם הרשימה ריקה. הדבר היחיד שחדש כאן מבחינתנו הוא הגדרת המחלקה הפנימית, יש שני סטראקטורים שממשים אותם מחוץ למחלקה List (שורה 43) מאחר ומדובר במחלקה פנימית המוגדת בתוך המחלקה List יש להגדיר את המימוש בצורה כפולה List::Link::Link ואז הגדרת הבנאי עובדת בצורה רגילה.
יש לשים לב שכל הגדרה של שדה או פונקציה פנימית יש לעשות את ההגדרה הכפולה שלו.
בlist יש שני בנאים – ברירת מחדל ומעתיק.
<u>הdefault constructor</u> – מאפס את המצביע על הhead בראש הרשימה ומכניס שם רשימה ריקה כאשר הhead==NULL. יש לשים לב, שהאתחול הזה גורם לרשימה שלא יהיה בה ערך לא מאותחל המצביע על זבל, אלא מגדיר אותה מחדש כרשימה שאין בה כלום, ויש הבדל גדול בין השניים.
<u>Copy constructor –</u> מקבל "const link&" מתקבלת רשימת מקור אותה יש לעביר לזיכרון חדש בצורת העתקה עמוקה, כל השדות צריכות לעבור במלואן. יש להעתיק הכל לאובייקט מזמן חדש, המכיל שדה יחיד של ראש רשימה "Head", ראשית המעתיק בודק האם רשימת המקור ריקה. במידה וכן, היא מעתיקה לראש הרשימה הבא קישור לNULL. (שורה 55)
במידה ולא, ברור לנו שיש ברשימה המקורית לפחות שדה אחד. ומעתיקים את הHead לרשימה החדשה. וכך מתחילה לולאה הבודקת כל פעם האם יש ערך נוסף ומעתיק את הערך הקיים בו, בצורה של הגדרת new Link עד שהוא נתקל בNULL.

[4] הdestructor משחרר את הרשימה רק אם כל האובייקט נעלם מהscope ויפסיק להתקיים, אך לפעמים יש לנו צורך שהאיבר עדיין יהיה קיים אך לרוקן אותו – במקרה כזה נשתמש בפונקצית clear.

עקרונית צריך גם לבדוק בכל ההקצאות הדינמיות האם הם שוות לNULL ובמידה שכן להוציא התראה שאין מספיק זיכרון ולצאת מהתוכנית.
את ההעתקה עושים על ידי משתנים המוגדרים src (מקור) ו-trg (צומת) ובהם בודקים האם יש NULL בערך הבא, ובמידה והכל תקין מעתיקים את החוליות ליעד.
עד כאן הבניה של הרשימה וההעתקה שלה. ומכאן נעבור לששת הפונקציות של הרשימה.
ניתן לראות על פי סוג הפונקציות שמדובר על רשימה בצורה של מחסנית. כל השינויים עובדים רק על הערך הראשון ברשימה.

**Add** – מעבירים את הnext של הערך החדש שיצביע על הערך הראשון, ואת ומעבירים את הhead שיצביע על הערך החדש.

**firstElement –** בדיקה האם הרשימה ריקה, ואם לא החזרת הערך שבמקום הראשון. (לפני שמשחררים מצביע שמשפיע על דברים אחרים, יכול להיות שהשחרור של האובייקט יגרום לdestructor לעבוד ולשחרר אובייקטים נוספים, לכן כדאי להציב בו NULL לפני המחיקה כך שלא ישפיע על דברים אחרים.)

**search –** החזרה בוליאנית האם הערך הנתון מופיע ברשימה. אין סיבה להכניס פה חריגה, מאחר וגם במידה והרשימה ריקה, התשובה ממילא היא שהנתון לא מופיע ברשימה.

**isEmpty –** בדיקה בוליאנית האם יש ערך ראשון או לא. אפילו ללא תנאים אלא פשוט החזרה של אמת רק במידה ו head==NULL

**removeFirst, –** בדיקה האם קיים בכלל ערך ראשון, אם הוא ריק יש לזרוק חריגה. השמה של ערך ה headעל הערך הבא שאחרי הראשון.

**clear** – מעבר על **כל** הערכים ומחיקתם אחד אחרי השני.

התוכנית הראשית מגדירה שני אובייקטים של הרשימה המקושרת. הרשימות מאותחלות על ידי הבנאי- ב"מ שמכניס בהם head=NULL. הלולאה הראשונה מכניסה ברשימה הראשונה לפי הסדר את הספרות 1-0 בצורת מחסנית ומיד מדפיסים אותם. סדר ההדפסה יהיה לפי הסדר, כאשר במחסנית עמה הם יהיו מסודרים הפוכים.
לאחר מכן מעבירים למשתנה element את הערך הראשון ברשימה (על פי הסדר ההפוך), מדפיסים אותו, ומעבירים לרשימה השניה, כך שברשימה השניה יהיה 4 פעמים את המספר הראשון של הרשימה הראשונה(3).
לאחר מכן מחפשים האם קיים הערך "4" והוצאה של הודעה מתאימה וכן למספר "3".
ועכשיו נעבור לירושה. יאייי.

```cpp
#include <iostream>
#include <string >
using namespace std;
class Document
{
private:
	char *name; // Document name.
public: Document(char* docName = nullptr);
		void setName(char*);
		char* getName() { return name; }
		void print();
};
Document::Document(char* docName)
{
	setName(docName);
}
void Document::setName(char* docName)
{
	if (docName)
	{
		int len = strlen(docName) + 1;
		name = new char[len];
		strcpy(name, docName);
	}
}
void Document::print()
{
	cout << "Name: " << name << endl;
}
class Book : public Document
{
private:
	long pageCount;
public:
	Book(char *name = nullptr, long pageNum
= 0);
	void setNumOfPages(long num);
	void print();
};
Book::Book(char *name, long pageNum)
	:Document(name)
{
	pageCount = pageNum;
}
void Book::setNumOfPages(long pageNum)
{
	pageCount = pageNum;
}
void Book::print()
{
	Document::print();
	cout << "Number Of Pages: ";
	cout << pageCount << endl;
}
```

```cpp
int main()
{
	Document d;
	Book b;
	d.setName("Doc");
	b.setName("My Book");
	b.setNumOfPages(543);
	d.print();
	b.print();
	b.Document::print();
	return 0;
}
/*Output:
Name: Doc
Name : My Book
Number Of Pages : 543
Name : My Book*/
```

# ירושה

```
class son: privacy father
{
public:
void func(type &name);
}
```

ירושה בc++ היא האפשרות ליצור מחלקה אחת שתהווה מחלקה ראשית המכילה משתנים ופונקציות, ומחלקת משנה אחת או יותר שתהיה חיצונית למחלקה הזאת (לא מחלקה פנימית כמו שראינו ברשימה בדוגמאות הקודמות), ו"תירש" את כל היכולות של המחלקה המקורית. באשר אנחנו מגדירים מחלקת בן, למעשה אנחנו צריכים לזכור שהיא בעצם מוגדרת גם כמחלקת אב, ולכן ההתייחסות אליו היא כפולה.
את הירושה מגדירים בסוג פרטיות שונה בכל פעם, ואת הפירוט יש בהמשך. כל סוג פרטיות מוריש לבן אפשרויות שונות ברמת גישה שונה לאופציות מהאב.

בתוכנית ניתן לראות מחלקה Document במחלקה יש שדה יחיד המכיל את שם המסמך כמחרוזת דינאמית. נתון זה הוא הנתון היחיד שאנחנו יודעים. יכול להיות שמדובר בספר המכיל את תוכן הספר עם מספרי עמודים, יכו להיות רק מידע כללי, אבל תמיד יש שם למסמך.
יש בנאי המקבל מצביע לשם המסמך ושולח את השם לפונקצית setName, ובהתאם יש גם פונקציית getName ואת ההדפסה של הנתון היחיד שאנו יודעים – שם המסמך.
המחלקה השניה נקראת Book והיא יורשת את המידע מהמחלקה document על ידי התחביר המיוחד: "`class Book : public Documen` " ועל ידי הגדרת ה"בן" של הbook כיורש מהאב document", האובייקט "Book" מקבל את כל המידע והאפשרויות הקיימות במחלקת האב.
ביסוס המחלקה על ידי מחלקה קיימת, גורם לכך שכל אובייקט הוא בעצם מקרה פרטי המשתייך למחלקת האב. בשדה של הספר יש גם את מספר העמודים בספר. כך שלמעשה ניתן להגדיר שני שדות של אתחול – אתחול הספר על פי שם המסמך, ואתחול של מספר העמודים השייך לאותו ספר.
המיוחד פה הוא שהבנאי מאתחל בעצמו את הקשור למחלקת האב. גם אם מדובר בכמות גדולה של פרמטרים , כולם מאותחלים על ידי האב והוא מקצה ובודק על פי כל הדרוש, כך שמה שנותר ליורש הוא לאתחל את הבסיס הקשור אליו.
פעולה זאת מתבצעת בצורה שאחרי זימון הקונסטרקור הרגיל מוסיפים בנקודתיים[5] את הגדרת השם בעזרת הקונסטרקטור של הdocument.
ניתן לראות בתוכנית הראשית שהאובייקט b של הBook מגדיר את שמו ישירות בsetName ולא "ניגש" למחלקת האב. אבל לא ניתן להגדיר את מחלקת הבן לפני שמגדירים את האב, ולכן יש להגדיר אובייקט למחלקת האב ורק אז להתחיל את המחלקה היורשת.
יש דבר מיוחד אותו צריך לזכור – יש זימון לקונסטרקטר למחלקת האב שמפעיל אותו בצורה פנימית.

[5] Book::Book(char *name, long pageNum)
:Document(name)

במידה ולא הגדרנו את השם המתיחס למחלקת האב והוא לא מאותחל, מה קורה אז? למעשה מוגדר שם המסך בתוך NULL וכך הגדרת השם קופצת ולא נכנסת לif ויכול להיות שהיה צריך להכניס else בידה ולא מגיע ערך ואז להגדיר את הNULL, אחרת התוכנית תקרוס.
אם לא נכתוב זימון מפורש לבנאי ש מחלקת האב, הוא יעודכן על פי ברירת המחדל של האב. אם אין בו בנאי בירית מחדל נקבל שגיאת קומפילציה.

## סוגי ירושה:

Class Child:
public
protected
private

יש משמעות לכל סוג הגדרה בכל ירושה. כאשר יורשים מחלקה יש משמעויות שונות כיצד תהיה הגישה בבן לכל אחת מהגישות –

| גישה באב (parent) | אופן הירושה | גישה בבן (child) |
|---|---|---|
| Public | public | Public |
| Protected | | protected |
| private | | אין גישה |
| Public | protected | protected |
| Protected | | Protected |
| private | | אין גישה |
| Public | private | Private |
| Protected | | Private |
| private | | אין גישה |

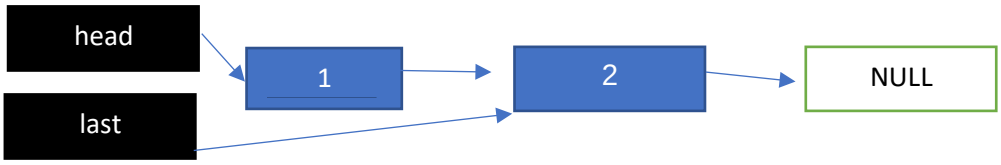
הגישה לאיברים אף פעם לא יכולה להיות מעבר ל מה שמוגדר באופן הירושה. אם מגדירים את הגישה ברמה גבוהה ניתן לגשת לPublic וכן לprotected שמעביר את מה שיש ליורשים ממנו.
אך אם מגדירים את הירושה בprivate ניתן רק לראות את הנתונים אך לא באמת לשנות בהם משהו אלא רק לצפות בהם.

```
// class DoubleEndedList
// a variation on Lists - can add elements
// to the end as well as to front
//-----------------------------------------
#include "List.h"
class DoubleEndedList : public List
{
public:
// constructors
	DoubleEndedList();

// override the following methods from class List
	void add(int value);
	void clear();
	void removeFirst();
// add a new element to the end of the List
	void addToEnd(int value);
protected:
// data area -- Link to end
	Link * last;
};
//-----------------------------------------
// class DoubleEndedList implementation
//-----------------------------------------
DoubleEndedList::DoubleEndedList() : List(),
last(nullptr)
{}
void DoubleEndedList::add(int val)
{
// add an element to the front of a double
// ended List only need to handle addition to
// empty List
	if (isEmpty()) {
		List::add(val);
		last = head;
	}
	else
		List::add(val);
}
void DoubleEndedList::clear()
{
// delete all values from collection
	List::clear();
// then set the pointer to the last element to zero
	last = nullptr;
}
void DoubleEndedList::removeFirst() {
// remove the first element
	List::removeFirst();
// if we remove last element
	if (isEmpty())
		last = nullptr;

}
void DoubleEndedList::addToEnd(int val)
{
// add a new element to end of a double ended List
	if (last != nullptr)
	{ last->next = new Link(val, nullptr);
	last = last->next;

// otherwise, just add to front
	 else
		add(val);
	}
```

```
	// USE:
#include <iostream>
#include "DoubleEndedList.h"
	using namespace std;

	int main()
	{
		int element;
		DoubleEndedList ls1, ls2;
		for (int i = 0; i < 5; i++)
		{
			ls1.add(i);
			cout << i << " ";
		}
		for (int i = 0; i < 5; i++)
		{
			element =
ls1.firstElement();
			if (i < 2)
				ls1.removeFirst();
			cout << element << " ";
			ls2.addToEnd(element);
		}
		cout << endl;
		cout << ((ls2.search(1)) ? "ls2
includes 1" : "ls2 doesn't Include 1") <<
endl;
		cout << ((ls2.search(2)) ? "ls2
includes 2" : "ls2 doesn't Include 2") <<
endl;
		ls2.removeFirst();
		cout << ((ls2.search(2)) ? "ls2
includes 2" : "ls2 doesn't Include 2") <<
endl;
		return 0;
	}
	/*
	 Output
	 0 1 2 3 4 4 3 2 2 2
	 ls2 doesn't Include 1
	 ls2 includes 2
	 ls2 includes 2
*/
```

# רשימה דו-צדדית (עם שני קצוות)

לפנינו רשימה העובדת כפול, ומסוגלת להכניס איברים לקצה הרשימה ולא רק לתחילתה.
הפונקציה מקבלת ירושה של הרשימה הרגילה, ומגדירה בנוסף את הlast המתייחס לNULL הקיים בו, על מנת שניתן יהיה להתייחס אליו. להלן הדגמה:




השינויים הדו כיוונים ברשימה נעשים על ידי הקישור הכפול הזה.
Add – משתמשים בפונקציה המקורית, באשר בשהרשימה ריקה ומדובר על הכנסה ראשונה, האיבר הראשון שנכנס, מוגדר מראש גם בתור התחלה וגם בתור סוף, וכל האיברים שיבואו אחרים פשוט ידחפו אותו קדימה בלי להגדיר קצה חדש, מאחר והוא כבר מוגדר.
Clear – מער למחיקה הרגילה של כל המערך, יש להגדיר את הזנב שלא יצביע על שום דבר ולכן last = nullptr.
removeFirst – מורידים את האיבר הראשון, בודקים אחר כך האם הוא היה האיבר היחיד ברשימה. במידה וכן, יש לוודא שמגדירים את הזנב שיצביע לNULL.
addToEnd – פונקציה חדשה המיוחדת רק לאופציה של רשימה עם שני קצוות - ראשית, בודקים שהרשימה לא ריקה (last != NULL) ואז מצרפים לשדה "הבא" של האחרון לינק חדש, המקבל ערך nullptr בתור next. דבר זה עושים באותו אופן שמוסיפים בצורה רגילה לראש הרשימה, מאחר ובעצם אין כל הבדל בין המצביע לראש למצביע לזנב.
יש לשים לב שבדיקת IsEmpty אין שום שינוי מאחר והבדיקה היא רק על הראש. אם שם אין כלום גם בקצה לא אמור להיות מוגדר כלום
יש כאן תופעה שנקראת "הסתרה" ויש לשים לב שיש הבדל בין זה לבין "דריסה" שיילמד בהמשך. ההסתרה זה בעצם השימוש הכפול שנעשה בפונקציה המקורית בצורה מוסתרת, ומשתמשים בשינויים ועדכונים שנעשים על גביהם.

```
#include <iostream>
#include <ctime>
using namespace std;
class Numbers
{
protected:
	int* vec;
	int size;
	virtual void swap(int i, int j);
	virtual int isSmaller(int i, int j);
	virtual void show(int i);
public: Numbers() { vec = nullptr; }
		Numbers(int);
		~Numbers();
		void print();
		void bubbleSort();
};
Numbers::Numbers(int munSize)
{
	size = munSize;
	vec = new int[size];
srand((unsigned)time(nullptr));
	for (int i = 0; i<size; i++)
		vec[i] = rand() % 100;
}
int Numbers::isSmaller(int i, int j)
{
	return (vec[i]<vec[j]);
}
Numbers::~Numbers() {
	if (vec) delete[] vec;
}
void Numbers::swap(int i, int j)
{
	int tmp = vec[i];
	vec[i] = vec[j];
	vec[j] = tmp;
}
void Numbers::bubbleSort()
{
	for (int last = size - 1; last > 0;
last--)
	{
		for (int i = 0; i < last; i++)
		{
			if (isSmaller(i + 1, i))
				swap(i, i + 1);
		}
	}
}
void Numbers::print()
{
	for (int i = 0; i<size; i++)
		show(i);
}
void Numbers::show(int i)
{
	cout << i << " : " << vec[i] << endl;
}
```

```
#include "Numbers.h"
class String : public Numbers
{
public:
	String(char*, char*, char*, char*);
private:
	char words[4][20];
	void swap(int i, int j) override;
	void show(int i) override;
	int isSmaller(int i, int j) override;
};
String::String(char* w0, char* w1, char* w2, char* w3)
{
	size = 4;
	strcpy(words[0], w0);
	strcpy(words[1], w1);
	strcpy(words[2], w2);
	strcpy(words[3], w3);
}
int String::isSmaller(int i, int j)
{
	return strcmp(words[i], words[j])<0;
}
void String::show(int i)
{
	cout << i << " : " << words[i] << endl;
}
void String::swap(int i, int j)
{
	char tmp[20];
	strcpy(tmp, words[i]);
	strcpy(words[i], words[j]);
	strcpy(words[j], tmp);
}
```

## USE:

```
#include "String.h"
int main()
{
	Numbers nums(5);
	cout << "Print Before Sort:\n";
	nums.print();
	nums.bubbleSort();
	cout << "Print After Sort:\n";
	nums.print();
	String words("Sara", "Rivka", "Rachel", "Leah");
	cout << "Print Before Sort:\n";
	words.print();
	words.bubbleSort();
	cout << "Print After Sort:\n";
	words.print();
	return 0;
}
```

# פולימורפיזם – ריבוי צורות

# מחלקות וירטואליות

```
virtual void NonAbstractMemberFunction (); // Virtual function.
```

**פולימורפיזם** – אפשרות ל"העמסת" פונקציות בצורה שיש פונקציה אחת שפועלת בצורה אחידה על כל הטיפוסים ללא הגדרה שונה לכל אחד ואחד מהם. במקרה שלפנינו יש מחלקה שיורשת את התכונות המתאימות לפונקציה ומשנה אותה בהתאם לערכים החדשים. התוכנית הראשית ששולחת את הטיפוסים השונים יודעת לשלוח לפונקציה הנכונה על פי סוג המשנה. דבר זה אפשרי בזכות
**פונקציות וירטואליות** – פונקציה אותה מגדירים במחלקת האב, כאשר בתחילת השורה מוסיפים את המילה virtual הנותנת לנו את האפשרות להשתמש בבסיס הפונקציה בכל המחלקות היורשות. בכל מחלקה יורשת המשתמשת בפונקציה יש לציין בסוף ההכרזה על כך שהיא override. אם לא תוגדר הפונקציה הוירטואלית במחלקה היורשת, היחס לפונקציה יעבור ישר לפונקציית האב.

בדוגמא זאת מוגדר לנו מערך דינמי של מספרי Int. יש 3 פונקציות וירטואליות בסיווג "Protected" שהשימוש שלהם יהיה רק במחלקה של הnumbers ובמחלקות שיירשו ממנה.
קיימים שני קונסטרקטורים , האחד ברירת מחדל שמגדיר את המערך כNULL. הקונסטרקטור השני המוגדר מחוץ למחלקה, מקבל את גדול המערך האמור להיות מוגדר, והקצאה של המערך הדינאמי בגודל מסוים. הגדרת המספרים היא בצורה אקראית בין 0-99.
התוכנית עושה מיון-בועות בעזרת שתי פונקציות פנימיות, בתחילה היא מחזירה האם האיבר הראשון קטן מהאיבר באינדקס השני(isSmaller), במידה וכן, האיברים מפרים את הסדר הדרוש ואנחנו מעבירים את האיברים להחלפה (swap)[6].
הפונקציות swap ו isSmaller משתמש במערך ובמספרים שלו ותלוי בInt ולכן לא מתייחס לשום ערך אחר.
פונקציית ההדפסה שולחת איבר איבר מהמערך לתוך פונקציה המוציאה פלט המתייחס גם למיקום במערך וגם לערך הקיים באינדקס.
בנוסף מוגדרת מחלקה numbers, המורישה לstring – מטריצה של מספרים בגודל קבוע של 19*4[7], מטרת הפונקציה היא למיין את ארבעת המחרוזות ומגדיר את גודל המחרוזת מתוך הירושה.
בסופו של דבר בתוכנית הראשית, מכניסים 4 מחרוזות ולאחר מכן ממינים אותם על פי סדר אלפביתי.
החידוש הוא שמיון הבועות הוא מוגדר כעובר על איברי int אך מסוגלת למיין גם את המחרוזות.
בהגדרה אנחנו מכריזים עליהם כvirtual ולאחר מכן משתמשים בהם בהסתרת השם "וירטואל" ומתייחסים אליהם על פי טיפוסי איברים שונים
הגדרת virtual בעצם מוסיפה שדה לכל אובייקט חדש בתור פוינטר בשם "vfprt" הפוינטר הזה מצביע למערך של פוינטרים נוסף המצביע לפונקציה. בטבלת הפונקציות הוירטואליות קיים מצביע לכל הפונקציות הוירטואליות, למשל בדוגמא זאת, האיבר הראשון יצביע לכיוון ה isSmaller. (כשם שמצביע מכל טיפוס, יכול להצביע רק על הטיפוס שלו, מצביע של פונקציה יכול להצביע רק לאותו טיפוס של פונקציה – אם המחלקה מקבלת שני מספרי intומחזירה Int, היא יכולה להצביע רק על פונקציה בפורמט הזה) בזמן שיוצרים אובייקט מחלקה מסוג string מגדירים גם מחלקה של האב,

[6] על אף שפה ה מוגדר כInt אין לזה ערך מיוחד ואפשר גם bool
[7] ההגדרה היא ל20 עמודות אך מאחר ומדובר בסטרינג ישלזכור את ה '\0'

ביחד עם הפוינטר. אך בשינוי הפונקציות כך שיישאר זהה לפונקציות המקוריות מלבד ה"דריסה" המתבקשת, בשינוי שם המחלקה של האב לשם המחלקה של הבן.

```
class SortAndPrint
{
protected:
	int size;
	virtual void swap(int i, int j) = 0;
	virtual void show(int i) = 0;
	virtual int isSmaller(int i, int j) =
0;
public:
	void setSize(int num);
	void print();
	void bubbleSort();
};
void SortAndPrint::setSize(int num)
{
	size = num;
}
void SortAndPrint::print()
{
	for (int i = 0; i<size; i++)
		show(i);
}
void SortAndPrint::bubbleSort()
{
	for (int last = size - 1; last > 0;
last--)
		for (int i = 0; i < last; i++)
			if (isSmaller(i + 1, i))
				swap(i, i + 1);
}
```

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
class Numbers : public SortAndPrint
{
private:
	int* vec;
	void swap(int i, int j) override final;
	int isSmaller(int i, int j)override final;
	void show(int i)override final;
public:
	Numbers(int);
	~Numbers() { if (vec) delete[] vec; }
};
Numbers::Numbers(int numSize)
{
	setSize(numSize);
	vec = new int[numSize];
srand((unsigned)time(nullptr));
	for (int i = 0; i<numSize; i++)
		vec[i] = rand() % 100;
}
int Numbers::isSmaller(int i, int j)
{
	return (vec[i]<vec[j]);
}
void Numbers::swap(int i, int j)
{
	int tmp = vec[i];
	vec[i] = vec[j];
	vec[j] = tmp;
}
void Numbers::show(int i)
{
	cout << i << " : " << vec[i] << endl;
}
```

```
#include <iostream>
#include <string.h>
using namespace std;
class String : public SortAndPrint
{
private:
	char words[4][20];
	void swap(int i, int j) override final;
	int isSmaller(int i, int j)override final;
	void show(int i)override final;
public:
	String(char*, char*, char*, char*);
};
String::String(char* w0, char* w1, char* w2, char*
w3)
{
	setSize(4);
	strcpy(words[0], w0);
	strcpy(words[1], w1);
	strcpy(words[2], w2);
	strcpy(words[3], w3);
}
int String::isSmaller(int i, int j)
{
	return strcmp(words[i], words[j])<0;
}
void String::swap(int i, int j)
{
	char tmp[20];
	strcpy(tmp, words[i]);
	strcpy(words[i], words[j]);
	strcpy(words[j], tmp);
}
void String::show(int i) {
	cout << i << " : " << words[i] << endl;
}
```

# מחלקה אבסטרקטית

```
class AbstractClass {
public:
 virtual void AbstractFunction() = 0;// Pure virtual function
};
```

אם נסתכל על המחלקות הוירטואליות בדוגמאות הקודמות, יש שגיאה חמורה ברמת עיצוב התוכנה. המערך של הנתונים עובר גם למחרוזות למרות שלמעשה, אין בו שום שימוש. יותר מזה, אם ננסה להגיע אליו נקבל NULL. מצב כזה נחשב פירצה בתוכנה וכל ניסיון לגישה אליה יקריס את התוכנה. בבדי לתקן את הליקוי הזה בעיצוב התוכנה עשו בדוגמא זו עיצוב מחדש למחלקה.
הפכו את המחלקות Numbers וstring כך שיהיו בנים למחלקה חדשה, הנקראת SortAndPrint (מיון והדפסה), כך שהמחלקות של הבנים יכילו רק את האיברים הרלוונטים לעצמם, ובמחלקת האב יוגדרו הפונקציות של מיון-הבועות.
הפונקציות הייעודיות מקבלות את האיברים i ו-j. וכל אחד מהם מכיל איברים שלא רלוונטיים לצד השני, ברמת הטיפוסים, ואין לנו אפשרות להגדיר את המערך בלי לדעת מה הוא אמור להכיל.
הפתרון – אנחנו יודעים כיצד אמור להיראות הפרוטוטיפ של הפונקציה, אך אנו לא יודעים איזה סוג טיפוסים לעשות, מה שצריך הוא שיהיה דרך להגדיר את דרך הפעולה של הפונקציה שיהיה מסוגל לעבוד עם כל סוג משתנה שיגיע אליו.
על מנת לבצע זאת, בסוף הגדרת הפונקציה מכניסים "=0" וזה אומר לקומפיילר לא לחפש את המימוש של הפונקציה אלא לחכות להמשך העבודה. דבר כזה נקרא "פונקציות וירטואליות טהורות". כל עוד נשארות פונקציות שלא מומשו, לא ניתן להגדיר אובייקטים של המחלקה.
מחלקה שכזאת נקראת "**מחלקה מופשטת**" (מחלקה שלא ניתן ליצור אובייקטים שלה.
איך מגדירים את המחלקות האלה? ברגע שאחת מהפונקציות הן וירטואליות טהורות **כל** המחלקה נקראת אבסטרקטית. הקונספציה של המחלקה המופשטת לא קשור רק לפונקציות וירטואליות טהורות, וגם בשפות האחרות ניתן להגדיר abstract.
למה יש צורך במחלקות מופשטות? לפעמים מחלקה מתארת קבוצת עצמים שלא קיימים באופן מוחשי, משל: חברה שיש בה עובדים ולכל אחד יש תפקיד שונה, ניתן להגדיר מחלקות לכל סוג עובד (ייצור, ניהול וכו') ואין אף עובד שנחשב רק "עובד" (בהשאלה למחלקת האב), כל אחד מוגדר לתת-קטגוריה.
(בשיעורי הבית – תרגיל 7 - יש קריטריונים למלגה לכל תואר בנפרד, כאשר מגדירים לכל סטודנט את התואר אותו הוא לומד – ראשון, שני, שלישי, ואין שום תלמיד שיכול להיות **לא מוגדר** לאחת מהאפשרויות
בתרגיל זה נבליל את ספריית vector, וניגש אליה בתוכנית הראשית בעזרת vector<int>.arr כאשר בתוך הסוגריים נגדיר את סוג המשתנים אליהם מתייחסים בווקטור, וכן ניתן לקבל פונקציות המוגדרות בספריה כגון push_back – הוספה לסוף הרשימה, size – החזרת גודל המערך)

נחזור לדוגמא:
המחלקה האבסטרקטית מבילה את הבסיס למיון הבועות, אך ללא התייחסות שונה לכל סוג משתנה. כל מחלקה חדשה שתיכתב על פי כל משתנה הקשור אליו, ייכתב איתן ביחד 3 פונקציות הדורסות את הפונקציה של מחלקת האב על פי המימוש הרלוונטי. מבחינת המימוש, בסוף כל הגדרת פונקציה בבן מוסיפים את ההגדרה "override", ניתן גם להוסיף "override final" כך שאם יהיה נבד רלונטי הוא לא יוכל לדרוס מחדש את הפונקציה.

```
/*****************************************************/
//pet.h
#if (!defined PET_H)
#define PET_H
#include <string>
#include <iostream>
using namespace std;
class Pet
}
public:
	virtual void print() { cout << "name " << name << endl; }
private:
	string name;
};
#endif
/*****************************************************/
//dog.h
#include "Pet.h"
class Dog : public Pet
}
public:
	void print() { cout << "name " << name << " breed " << breed << endl; }
	void price() { cout << "I am expensive!" << endl; }
private:
	string breed;
};
/*****************************************************/
//cat.h
#include "Pet.h"
class Cat : public Pet {
public:
	void talk() { cout << "Meow Meow" << endl; }
private:
	string gender;
};
/*****************************************************/
//main.cpp
#include <iostream>
using namespace std;
#include "Dog.h"
#include "Cat.h"
void main()
{
	Dog d;
	Cat c;
}
```

# הידור קבצים

אם יש קבצים שונים השייכים לאותה תוכנית, ומנסים להניס את הheader של המחלקה המכיל את אותו שם.
איך ניתן למנוע בעיה כזאת?
בגרסאות החדשות יש הגדרה של קדם-מהדר הנקראת "progma once#" המקבצת ביחד את כל הקבצים והוא עושה החלפות מאקרו, אך יש כאלה שלא מכירים בפקודה כזאת, במקרה כזה יש למנוע מהקומפיילר לקרוא פעמיים את אותו הדר.
כותבים "הידור-מותנה" (conditional compilation) בצורה כזו שאומרת לקדם-המהדר להתיחס לקטע מסוים ולדלג על קטע אחר ולהסתיר ותו מעיני הקומפיילר.
מזההdef if# – האם המזהה מוצהר
(מזהה defined!) if# – המזהה הזה מוגדר לפי שם הקובץ – רק שמקובל לקרוא לו באותיות גדולות ואי אפשר להשתמש בנקודה אלא בקו תחתון, אם הקומפיילר לא מכיר את השם של המזהה, סימן שהוא עדיין לא נתקן בקובץ הזה, ואז הוא מגדיר אותו.
במידה והתנאי לא מתקים, הקומיילר לא מסתכל בכלל על מה שכתוב אחרי התנאי עד הENDIF#
גם אם יש שם את השגיאות הכי גדולות, ואפילו טקסט לא קשור. אלא מדלג ישר למה שמופיע אחר כך.

המהדר בעצם נתקל 5 פעמים בקובץ האב pet, כאשר בהתחלה הוא קורא את הבסיס של התוכנית פעם אחת, פעמיים בהגדרת חיית המחמד, ועוד פעמיים בתוכנית עצמה של הגדרת האובייקטים של המחלקות השונות.

```
#include <iostream>
using namespace std;
class A
{
private:
	char* str1;
public:
	A(char* str);
	~A();
};
A::A(char* str)
{
	cout << "A constructor\n";
	str1 = new char[strlen(str) + 1];
	strcpy(str1, str);
}
A::~A()
{
	cout << "A destructor\n";
	if (str1)
		delete str1;;
}


class B :public A
{
private:
	char* str2;
public:
	B(char* str);
	~B();
};
B::B(char* str) :A(str)
{
	cout << "B constructor\n";
	str2 = new char[strlen(str) + 1];
	strcpy(str2, str);
}
B::~B()
{
	cout << "B destructor\n";
	if (str2)
		delete str2;
}
int main()
{
	A aa("test A");
	B bb("test B");
	return 0;
}
/*output:
A constructor
A constructor
B constructor
B destructor
A destructor
A destructor*/
```

```
#include <iostream>
using namespace std;
class A
{
private:
	char* str1;
public:
	A(char* str);
	~A();
};
A::A(char* str)
{
	cout << "A constructor\n";
	str1 = new char[strlen(str) + 1];
	strcpy(str1, str);
}
A::~A()
{
	cout << "A destructor\n";
	if (str1)
		delete str1;;
}


class B :public A
{
private:
	char* str2;
public:
	B(char* str);
	~B();
};
B::B(char* str) :A(str)
{
	cout << "B constructor\n";
	str2 = new char[strlen(str) + 1];
	strcpy(str2, str);
}
B::~B()
{
	cout << "B destructor\n";
	if (str2)
		delete str2;
}
int main()
{
	A* ab = new B("test ab");
	delete ab;
	return 0;
}
/*output:
A constructor
B constructor
A destructor
*/
```

```
#include <iostream>
using namespace std;
class A
{
private:
	char* str1;
public:
	A(char* str);
	virtual ~A();
};
A::A(char* str)
{
	cout << "A constructor\n";
	str1 = new char[strlen(str) + 1];
	strcpy(str1, str);
}
A::~A()
{
	cout << "A destructor\n";
	if (str1)
		delete str1;;
}



class B :public A
{
private:
	char* str2;
public:
	B(char* str);
	~B() override;
};
B::B(char* str) :A(str)
{
	cout << "B constructor\n";
	str2 = new char[strlen(str) + 1];
	strcpy(str2, str);
}
B::~B()
{
	cout << "B destructor\n";
	if (str2)
		delete str2;
}
int main()
{
	A* ab = new B("test ab");
	delete ab;
	return 0;
}
/*output:
A constructor
B constructor
B destructor
A destructor
*/
```

# Virtual destructors

```
class Interface {
public:
 virtual ~Interface(); // pure virtual destructor
};

Interface::~Interface(){} //virtual destructor definition
(should always be empty)
```

בדוגמאות הללו מחלקת-האב נקראת A והבן נקרא B.
מה יש במחלקה A – מופע בודד שנקרא str1 אליו שולחים מחרוזת וכן במחלקת הבן בעצם יש את המחרוזת הזו בצורה מוסתרת מעצם הירושה.
למעשה במחלקה B יופיע פעמיים המחרוזת כל אחד בתוך חלק אחר של ה"משפחה" אך לכל אחד תהיה הגדרה אחרת וזיכרון דינאמי אחר.
בדוגמא הראשונה יוגדרו המחלקות בסדר הבא:

1. מחלקה A בפני עצמה.
2. מחלקה A בתוך מחלקה B
3. מחלקה B

ולאחר מכן הדיסטרקטור יעבור בצורה הפוכה:

4. מחלקה B
5. מחלקה A בתוך B
6. מחלקה A

בדוגמא הזו אין שום בעיה של דליפת זיכרון ואפשר לעבור לשלב הבא.
הדוגמא השניה היא קצת שונה – יש לנו מצביע מטיפוס A, כאשר שאר הקוד בדיוק אותו דבר.
כאשר ההקצאה הדינאמית היא דרך B. מכיוון שאנחנו מקצים את האובייקט מאופן דינאמי, אנחנו משחררים אותו על ידי delete. איפה קונסטרקטורים יעבדו?
קודם כל נבנית הליבה, שהיא טיפוס A המוגדר בB – הווה אומר – מחלקה המכילה את B, כך שסדר העבודה יהיה:

1. מחלקה A בתוך B
2. מחלקה B
3. דיסטרקטור A (שימחק את הכל)

אך בעצם המיוחד פה הוא המצביע מתייחס רק לA הפנימי ולא מה שעוטף אותו (B) והדיסטרקטור יעבוד רק על הA אותו הוא מכיר מה שיגרום לדליפת זיכרון, מאחר ואין שום הפעלה לדיסטרקטור של B.
מה הפיתרון?
בדוגמא השלישית הגדירו את הדיסטרקטור של A כוירטואלי. כך שיתווסף הפוינטר הסמוי __vfptr[8], המכיל מערך פונקציות וירטואליות ובין השאר גם לדיסטרקטור. הפוייטר A עדיין רואה רק את עצמו, אך הפונקציה הוירטואלית התמלאה ברגע יצירת המחלקה היורשת, כך שברגע השחרור, הדיסטרקטור הוירטואלי ישחרר גם את השכבה החיצונית של B.

**כלל חשוב:**

[8] כאמור, אין דרך לראות את הפוייטר הזה מלבד בדיבאגר.

**במידה ומגדירים מחלקת-אב שתירש למחלקות בנים אחרות, יש להקפיד להגדיר את הדיסטרקטור כוירטואלי, כך שיובל לפעול על כל המחלקות היורשות ממנו.**

```
//-----------------------------------------
// class stack
// abstract class - simply defines protocol
for
// stack operations
//-----------------------------------------
class Stack
{
public:
	virtual void clear() = 0;
	virtual bool isEmpty() const = 0;
	virtual int pop() = 0;
	virtual void push(int value) = 0;
	virtual int top() = 0;
};

//#include "Vector.h" //homework targil 2 !
//-----------------------------------------
// class StackVector
// Stack implemented using Vector
// Vector will grow as necessary to avoid
overflow
//-----------------------------------------
class StackVector : public Stack
{
public:
	// constructor requires a starting size
	StackVector(unsigned int capacity);
	StackVector(const StackVector& s);
	// Stack operations
	void clear() override;
	bool isEmpty() const override;
	int pop() override;
	void push(int value) override;
	int top()override;
protected:
	// data fields
	Vector data;
};


//#include "List.h" //page 18
//-----------------------------------------
// class StackList
// Stack implemented using List operations
//-----------------------------------------
class StackList : public Stack
{
public:
	StackList();
	StackList(const StackList&);
	// Stack operations
	void clear()override;
	bool isEmpty() const override;
	int pop()override;
	void push(int value) override;
	int top()override;
protected:
	// data fields
	List data;
};
```

```
#include <iostream>
#include "StackVector.h"
#include "StackList.h"
using namespace std;
int main() {
	try {
		Stack* st;
		char base[7];
		cout << "Do you want a list base
or a vector base ? ";
		cin >> base;
		if (!strcmp(base, "vector"))
			st = new StackVector(20);
		else
			st = new StackList();
		for (int i = 0; i <= 20; i++)
			st->push(i);
		while (!st->isEmpty())
			cout << st->pop() << " ";
	}
	catch (const char* str) {
		cout << str;
	}
	return 0;
}
```

```
//-----------------------------------------
// class StackVector implementation
//-----------------------------------------
StackVector::StackVector(unsigned int
capacity)
	: data(capacity)
{
// create and initialize a Stack based on
Vectors
}
StackVector::StackVector(const StackVector& s)
	: data(s.data)
{}
void StackVector::clear()
{
// clear all elements from Stack, by setting
// index to bottom of Stack
	data.clear();
}
bool StackVector::isEmpty() const
{
	return data.getSize() == 0;
}
int StackVector::pop()
{
// return and remove the intopmost element in
the Stack
	if (isEmpty()) throw "Stack is empty";
	return data.delLast();
}
void StackVector::push(int val)
{
// push new value onto Stack
	data.insert(val);
}
int StackVector::top()
{
// return the intopmost element in the Stack
	if (isEmpty()) throw "Stack is empty";
	return data[data.getSize() - 1];
}
```

```
//-----------------------------------------
// class StackList implementation
//-----------------------------------------
StackList::StackList() :data()
{
// create and initialize a Stack based on
Lists
}
StackList::StackList(const StackList& lst)
	: data(lst.data)
{ /* copy constructor*/
}
void StackList::clear()
{
// clear all elements from Stack, by setting
/ delete all values from List
	data.clear();
}
bool StackList::isEmpty() const
{ // return true if Stack is empty
	return data.isEmpty();
}
int StackList::pop()
{
// return and remove the intopmost element in
the Stack
// get first element in List
	int result = data.firstElement();
	// remove element from List
	data.removeFirst();
	// return value
	return result;
}
void StackList::push(int val)
{
	// push new value onto Stack
	data.add(val);
}
int StackList::top()
{
	return data.firstElement();
}
```

# Stack

המחלקה "Stack" מבילה רק פונקציות וירטואליות טהורות, ללא שום פעולה בפועל במחלקה. מה שיש זה רק פרוטוטייפ ל5 פונקציות שהמחסנית צריכה לתמוך בהם: 1. ניקיון מחסנית. 2. בדיקת ריקנות 3. הוצאה 4. הכנסה. בדיקת האיבר הראשון.
מחלקה בזאת נקראת "ממשק" Interface אין לזה תחביר מסוים, אלא ההגדרה היא לכל מחלקה שמבילה **רק** פונקציות טהורות. (בשפות אחרות יש הגדרת "ממשק"). המחלקה הזאת מגדירה את הפעולות הבסיסיות של המחסנית, ללא שום הגדרה אלא רק "פרוטוקול" של פעולות לשימוש.
מה זה נותן לנו?
אפשר להגדיר מצביע למחלקה מופשטת, ומצביע מסוג מחלקת אב, יכול להצביע על כל המחלקות היורשות ממנו. במקרה של ממשקים, מחלקה שיורשת מהמחסנית, צריכה לדרוס את **כל** הפונקציות של הממשק, ולממש את כל מה שהמחלקה דורשת (מאחר והסברנו שכל פונקציה וירטואלית צריכה להיות ממומשת בכל מחלקה שיורשת ממנה, מחלקה שיורשת מהממשק חייבת לממש לפחות את כל הפונקציות האלה), וכך בעזרת מצביע נוכל לממש את כל המחסניות שנצטרך.
למשל בדוגמא: יש מצביע מסוג stack, שואלים את המשתמש האם הוא רוצה מחסנית על בסיס רשימה או מחסנית על בסיס וקטור (זיכרון רציף), אם הוקלד וקטור, מקצים אובייקט דינאמי של המחלקה stack כוקרטור, התחביר הוא כירושה רגילה, שדה הנתונים של המחסנית הוא משתנה מסוג וקטור.
במידה והוקלד כל דבר שהוא לא המילה vector עוברים אוטומטית לאפשרות של מימו על ידי רשימה, ישנה מחלקה שמתייחסת לרשימה שבוצעה בתרגילים הקודמים. הפונקציות צריכות להיות ממומשות על פי הגדרת הרשימה, המתבססת על מחלקת הרשימה.
ההבדל בין שני המימושים הוא שהרשימה היא בלתי-מוגבלת בכמות אליה ניתן להכניס, והווקטור צריך להיות ממומש מראש בגודל ספציפי ואסור לחרוג ממנו, בתוכנית הזאת מגדירים את הוקטור בגודל של 20 איברים).
מעבר לזה, יש לדאוג לכל החריגות השייכות לכל סוג מימוש למחסנית מלאה או ריקה בהתאם לפונקציה המתאימה.
הקונסטרקטור של הרשימה לא מקבל שום מידע, מאחר והוא בלתי מוגבל בזיכרון.
הפעולה של הוצאת האיבר הראשון, מחזירה את הInt של האיבר הראשון, וכן משתמשת בפנקיה של הרשימה של removeFirst, וכן הלאה.
בסופו של דבר, בתוכנית הזאת בכל אופן תקלוט רשימה רק של 20 מספרים, אם מדובר בהגדרת וקטור, או לולאה של הכנסת איברים למחלקת רשימה, ללמרות שאם לא ידוע הגודל הסופי של המחסנית עדיף להשתמש ברשימה שהיא איינסופית.

לסיכום: הוגדר ADT – abstract data type.

ואפשרויות מימוש שונות על פי בחירת המשתמש

```
'
//-----------------------------------------
// class Queue
// abstract class - simply defines protocol
for
// Queue operations
//-----------------------------------------
class Queue
{
public:
	// protocol for Queue operations
	virtual void clear() = 0;
	virtual int dequeue() = 0;
	virtual void enqueue(int value) = 0;
	virtual int front() = 0;
	virtual bool isEmpty() const = 0;
};
//-----------------------------------------
// class QueueVector
// Queue implemented using vector operations
//-----------------------------------------
class QueueVector : public Queue
{
public:
	// constructor requires a starting size
	QueueVector(int max);
	QueueVector(const QueueVector&);
	// implement Queue protocol
	void clear() override;
	int dequeue() override;
	void enqueue(int value) override;
	int front() override;
	bool isEmpty() const override;
private:
	int* data;
	int capacity;
	int nextSlot;
	int firstUse;
};
#include "DoubleEndedList.h"
//-----------------------------------------
// class QueueList
// Queue implemented using List operations
//-----------------------------------------
class QueueList : public Queue
{
public:
	// constructors
	QueueList();
	QueueList(const QueueList & v);
	// implement Queue protocol
	void clear() override;
	int dequeue()override;
	void enqueue(int value) override;
	int front()override;
	bool isEmpty() const override;
private:
	DoubleEndedList data;
};
```

```
#include <iostream>
#include <string>
#include "QueueVector.h"
#include "QueueList.h"
using namespace std;
int main() {
	Queue* Q;
	char base[7];
	cout << "Do you want a list or a vector
base Queue? ";
	cin >> base;
	if (!strcmp(base, "vector"))
		Q = new QueueVector(5);
	else Q = new QueueList();
	try {
		for (int i = 0; i<10; i++)
			Q->enqueue(i);
	}
	catch (const char* msg)
	{
		cout << msg;
	}
	cout << "first on Q is: " << Q->front()
<< endl;
	cout << "take out 2 elemets:" << endl;
	cout << Q->dequeue() << ' ' <Q-
>dequeue() << endl;
	cout << "first on Q is: " << Q->front()
<< endl;
	Q->enqueue(8);
	Q->enqueue(9);
	while (!Q->isEmpty())
		cout << Q->dequeue() << " ";
	return 0;
}
```

```
//== class QueueVector implementation ==
QueueVector::QueueVector(int size)
{
	capacity = size + 1;
	data = new int[capacity];
	clear();
}
void QueueVector::clear()
{
	nextSlot = 0;
	firstUse = 0;
}
int QueueVector::dequeue()
{
// can not dequeue from an empty queue
	if (isEmpty()) throw "Queue is
empty\n";
	int dataloc = firstUse;
	++firstUse %= capacity;
	return data[dataloc];
}
void QueueVector::enqueue(int val)
{
// make sure Queue has not overflowed
	if ((nextSlot + 1) % capacity ==
firstUse)
		throw "the Queue is full\n";
	data[nextSlot] = val;
	++nextSlot %= capacity;
}
int QueueVector::front()
{
// can not return a value from an empty Queue
	if (isEmpty()) throw "Queue is
empty\n";
	return data[firstUse];
}
bool QueueVector::isEmpty() const
{
// Queue is empty if next slot is
 // pointing to same location as first use
	return nextSlot == firstUse;
}
```

```
//== class QueueList implementation ==
QueueList::QueueList() :data()
{
	// no further initialization
}
void QueueList::clear()
{
	data.clear();
}
int QueueList::dequeue()
{
	int result = data.firstElement();
	data.removeFirst();
	return result;
}
void QueueList::enqueue(int value)
{
	data.addToEnd(value);
}
int QueueList::front()
{
	return data.firstElement();
}
bool QueueList::isEmpty() const
{
	return data.isEmpty();
}
```

# Queue

קוד דומה למחסנית, רק שכאן מדובר על "תור" FIFO המכיל מימושים דומים למחסנית, אך שונים על פי הבדלי מחסנית ותור של מבני נתונים רגילים.
הממשק מכיל את חמשת הפונקציות הקודמות, אך במקום דחיפה ושליפה, יש הכניסה לתור והיציאה מהתור.
גם פה נוכל להגדיר מצביע מסוג ממש שיממש את הפרוטוקול של התור, ובאופן דומה גם פה מדברים על וקטור ורשימה.
הוקטור באן לא ממומש על יש המחלקה המוגדרת של הויז'ואל, מאחר ויש פה לוגיקה קצת יותר מורכבת של ראש התור, סופו, וחזרה לתחילת המערך (הוקטור יותר יעיל למימוש של מחסנית).
מימוש התוק מגיע בצורה כזו שגודל הוקטור מוגדר +1 מהגודל המקסימאלי שמבקש המשתמש, על מנת שיהיה .
בנוסף מוגדרים שני שדות nextSlot ו-firstUse.
nextSlot – הזנב של הרשימה – המקום אליו ייכנס האיבר הבא, firstUse – מי שעתיד לצאת הבא בתור.
בתחילת ההגדרה, שני השדות מוגדרים לאותו איבר – האיבר הראשון. וכך גם נעשית בידקת הריקנות של התור, האם הם נמצאים באותו מקום.
באשר מכניסים איבר לרשימה, בודקים האם (nextSlot + 1) % capacity == firstUse), ובמידה ולא מבעים רצף פעולות: מגדילים את הnextSlot בו, ומחלקים במודולו לקפסטי, וכל עוד לא הגענו לקצה התור, הnextSlot יקבל את המספר הבא. כאשר ברגע שיגיע לקצה הגדרת המערך, הnextSlot יתאפס וכך בעצם יהיה שווה לufrstUse ויחזיר למשתמש שהתור כבר מלא.

התוכנית עצמה, מפעילה את הפעולות והפונקציות באופן שווה, למרות שעל פי בחיר המשתמש הפנים של התוכנה יעבור בצורה אחרת לגמרי, אך מבחינת המשתמש אין בזה הבדל.

## דוגמא 1.

```
#include <iostream>
using namespace std;
int max(int x, int y)
{
	if (x>y)
		return x;
	return y;
}
float max(float x, float y)
{
	if (x>y)
		return x;
	return y;
}
char max(char x, char y)
{
	if (x>y)
		return x;
	return y;
}
int main() {
	cout << max(3, 4) << endl;
	float a = 5.25, b = 3.5;
	cout << max(a, b) << endl;
	cout << max('a', 'z') << endl;
	return 0;
}
```

## דוגמא 2

```
#include <iostream>
using namespace std;
template <class T> T getMax(T x, T y)
{
	if (x>y)
		return x;
	return y;
}
int main() {
	cout << getMax<int>(3, 4) << endl;
	float a = 5.25, b = 3.5;
	cout << getMax<float>(a, b) << endl;
	cout << getMax<char>('a', 'z') << endl;
	return 0;
}
```

## דוגמא 3

```
#include <iostream>
#include <ctime > //to start rand by timing
using namespace std;
template <class T>
void Swap(T & x, T & y)
{
	T tmp = x; x = y; y = tmp;
}
template <class T>
void bubbleSort(T vec[], int size)
{
	for (int last = size - 1; last>0; last--)
		for (int i = 0; i < last; i++)
			if (vec[i + 1] < vec[i])
				Swap<T>(vec[i],
vec[i + 1]);
}
template <class T>
void print(T vec[], int size)
{
	for (int i = 0; i<size; i++)
		cout << vec[i] << ' ';
	cout << endl;
}
int main()
{
	int integers[10];
srand((unsigned)time(nullptr));
	for (int i = 0; i<10; i++)
		integers[i] = rand() % 100;
	bubbleSort<int>(integers, 10);
	print<int>(integers, 10);
	char characters[5];
	for (int i = 0; i<5; i++)
		characters[i] = (rand() % 26) + 65;
	bubbleSort<char>(characters, 5);
	print<char>(characters, 5);
	return 0;
}
```

# תכנות תבניתי – Templates

```
template<class T>
void funcName(T &a, T &b)
{
 function
}
```

במידה ויש תוכנית בה אנחנו צריכים לעשות בדיוק את אותם פעולות על משתנים שונים ללא שינוי של הפונקציה עצמה (ראינו בפונקציה הוירטואלית שמשנים בכ פעם את הונקציה בהתאם, אך אם אין בזה צורך) ניתן להגדיר תבנית של פונקציה המקבלת את הערכים ומפעילה עליהם את הפעולה ללא התחשבות בסוג המשתנה.
הדוגמא הראשונה מנמקת לנו את הצורך בתבניות. הפונקציה MAX מחזירה לנו את הגדול בין שני ערכים, אך במידה ונרצה מקסימום בין שני ערכים של float, או char נצטרך לכתוב עבור כל טיפוס הגדרה של פונקציה אחרת וזה מסרבל את כל הכתיבה.
בדוגמא השניה הכניסו את הפונקציה התבניתית בבסיס לפונקצית ההשוואה. ההכרזה על הפונקציה מוגדרת באופן הבא: ראשית הגדרה של התבנית template<class T>, ולאחר מכן שם הפונקציה, כאשר המשתנים המוכנסים לפונקציה נקראים בשמות שמתייחסים להגדרת הערכים שבסוגריים החדים.
כך שבכל פעם בתוכנית הראשית שנגדיר את הפונקציה הוא ייבנס מחדש על פי הערכים שנקלטים.
כל האמור, יוצא מנקודת הנחה שכל ההשוואות והפעולות המתבצעות, הן כאלה שיכולות לפעול על הערכים המוכנסים – למשל, אם מכניסים מחלקה שאין בה אפשרות להשוואה, תהיה בעיה בקומפילציה, ולכן יש לוודא שכל פעולה שמתבצעת על כל סוג טיפוס שיהיה צריך להכיל את כל הפעולות הממומשות בפונקציה התבניתית.

(בגרסאות ישנות יותר מגדירים הרבה פעמים במקום class את אותו דבר עם typename, אך מדובר על אותו דבר)

הפונקציות הבאות מבצעות מימושים שונים של תבניות בצורות של וקטור ועצים.

```
#include <iostream>
using namespace std;
const int DEF_CAPACITY = 100;
template <class T> class Vector
{
protected:
	T *data;
	int size; //size in use
	int capacity; //available capacity
public:
	//constructors
	Vector(int capacity = DEF_CAPACITY);
	Vector(const Vector<T>&);
	~Vector();
	//operations
	Vector<T>& operator = (const Vector<T>&);
	// view and modify
	T& operator [](int index);
	int getSize() const;
	int getCapacity() const;
	void insert(T value);
	void clear();
	T delLast();
};
//=======class Vector implementation=====
template <class T>
Vector<T>::Vector(int Capacity)
{
	capacity = Capacity;
	size = 0;
	data = new T[capacity];
	if (data == nullptr)
		throw "memory allocation problem";
}
template <class T>
Vector<T>::Vector(const Vector<T>& vec)
{
	capacity = vec.capacity;
	size = vec.size;
	data = new T[capacity];
	if (data == nullptr)
		throw "memory allocation problem";
	for (int index = 0; index < size;
index++)
		data[index] = vec.data[index];
}
template <class T>
Vector<T>::~Vector() {
	if (data != nullptr) {
		delete[] data;
		data = nullptr;
	}
}
template <class T> void Vector<T>::clear()
{
	size = 0;
}
//view and modify function
template <class T>
T& Vector<T>::operator [](int index)
{
	if (index < 0 || index >= size)
		throw "vector overflow";
	return data[index];
}
template <class T>
Vector<T> &Vector<T>::operator =
(const Vector<T>& vec)
{
	size = vec.size;
	capacity = vec.capacity;
	if (data)
		delete[] data;
	data = new T[capacity];
	if (data == nullptr)
		throw "memory allocation problem";
	for (int index = 0; index < size;
index++)
		data[index] = vec.data[index];
	return *this;
}
template <class T>
int Vector<T>::getSize() const
{
	return size;
}
template <class T>
int Vector<T>::getCapacity() const
{
	return capacity;
}
template <class T>
void Vector<T>::insert(T value)
{
	if (size >= capacity)
		throw "the vector is full";
	data[size] = value;
	size++;
}
template <class T>
T Vector<T>::delLast()
{
	if (size < 0)
		throw "the vector is empty";
	return data[--size];
}
```

```
//USE:
#include "Vector.h"
int main()
{
	Vector <int> nums;
	Vector <float> rels;
	nums.insert(58);
	nums.insert(42);
	rels.insert((float)58 / 100);
	rels.insert((float)42 / 100);
	for (int index = 0; index <
nums.getSize(); index++)
		cout << nums[index] << " ";
	cout << endl;
	for (int index = 0; index <
rels.getSize(); index++)
		cout << rels[index] << " ";
	return 0;
}
//Output:
// 58 42
// 0.58 0.42
```

# וקטור

מגדירים מחלקה תבניתית מסג וקטור, כאשר כל מימוש של הוקטור שונה בכל פעם בטיפוס בשהקומפיילר רואה תבנית, הוא מכין שטאנץ של הפונקציה, כאשר הוא משאיר מקום ריק בסוג הטיפוס, וכך בכל פעם שמגידירים את הפונקצה הוא משכפל את כל הפונקציות רק שהוא משנה בכל פעם את הטיפוס השונה בכל פעם לפי המתבקש

```
//---------------------------------
// class Tree (Binary Trees)
// process nodes in Pre/In/Post order
//---------------------------------
template <class T> class Tree
{
protected:
	//--------------------------------------
	// inner class Node
	// a single Node from a binary tree
	//--------------------------------------
	class Node
	{
	public:
		Node * left;
		Node * right;
		T value;
		Node(T val)
			: value(val),
left(nullptr), right(nullptr) {}
		Node(T val, Node * l, Node * r)
			: value(val), left(l),
right(r) {}
	};		//end of Node class

	Node * root;

public:
	Tree() { root = nullptr; }   //
initialize tree
	~Tree();
	bool isEmpty() const;
	void clear() { clear(root); root =
nullptr; }
	void preOrder() { preOrder(root); }
	void inOrder() { inOrder(root); }
	void postOrder() { postOrder(root); }

	virtual void process(T val) { cout << val
<< " "; }
	virtual void add(T val) = 0;
	virtual bool search(T val) = 0;
	virtual void remove(T val) = 0;

private:
	void clear(Node * current);
	void preOrder(Node * current);
	void inOrder(Node * current);
	void postOrder(Node * current);
};
```

```
#include "tree.h"
// class Tree implementation
//-----------------------------------------------
template <class T>
Tree<T>::~Tree() // deallocate tree
{
	if (root != nullptr)
		clear(root);
}
template <class T>
void Tree<T>::clear(Node * current)
{
	if (current)
	{ // Release memory associated with children
		if (current->left)
			clear(current->left);
		if (current->right)
			clear(current->right);
		delete current;
	}
}
template <class T>
bool Tree<T>::isEmpty() const
{
	return root == nullptr;
}
// preOrder processing of tree rooted at current
template <class T>
void Tree<T>::preOrder(Node * current)
{ // visit Node, left child, right child
	if (current)
	{ // process current Node
		process(current->value);
		// then visit children
		preOrder(current->left);
		preOrder(current->right);
	}
}
// inOrder processing of tree rooted at current
template <class T>
void Tree<T>::inOrder(Node * current)
{ // visit left child, Node, right child
	if (current)
	{
		inOrder(current->left);
		process(current->value);
		inOrder(current->right);
	}
}
// postOrder processing of tree rooted at current
template <class T>
void Tree<T>::postOrder(Node * current)
{ // visit left child, right child, node
	if (current)
	{
		postOrder(current->left);
		postOrder(current->right);
		process(current->value);
	}
}
```

# עצים

על בסיס המחלקות הוירטואליות, נבנה עץ בינארי שיהווה תבנית לשאר סוגי העצים.

תחת רמת Protected:

נגדיר מחלקה פנימית של node שתכיל בנוסף לערך הקיים בה שתי קישורים לבנים אפשריים ימינה ושמאלה. כאשר ברירת המחדל המינימלית היא לבנות רק את הערך בתוך הnode ואת הלינקים להכניס לnull. כולל אפשרות להגדיר גם את הלינקים בקונסטרקטור נפרד.

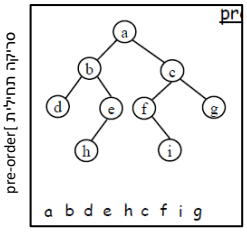
בנוסף נגדיר *node שיוגדר בתור root - שורש העץ, וכרגע לא נכניס לו ערכים.

תחת public:

עושים קונסטרקטור בסיסי לעץ המגדיר את שורש העץ בnullptr.

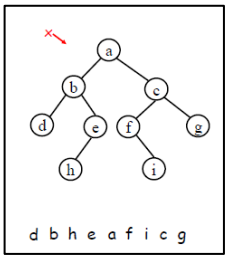
בנוסף מגדירים את הפונקציות הבסיסיות לעץ:

1. Is empty – בדיקת ערכים בעצם – בדיקה בוליאנית, האם הערך בשורש שווה NULL.
2. Clear – פונקציה שמוחקת את כל ערכי העץ – הפונקציה מקבלת את שורש העץ ועוברת בצורה רקורסיבית על כל העלים ומוחקת אותם אחד אחד.
3. Pre-order – בריקה תחילית – אחת מ3 פונקציות סריקה אפשריות לעץ על פי מה שנלמד במבנה נתונים. סריקה תחילית – הדפסת שורש – פניה שמאלה וכל עוד יש בנים משמאל הדפסה לשמאל, אחרת – לימין.

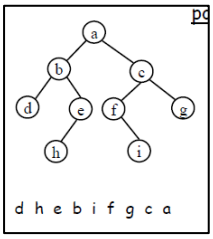


סריקה תחילית pre-order

4. In-order – סריקה תוכית – מתחילים מהאיבר השמאלי ביותר, עוברים לאיבר האב שלו, ואז יורדים לשאר הבנים של האב – כאשר יש עדיפות תמיד לבנים השמאליים
5. Post-order – סריקה סופית – מתחילים מהאיבר השמאלי ביותר, ומדפיסים אחריו את האיבר השמאלי ביותר אחריו, כך שהבן הימני יודפס לפני האב והכל נוטה להדפסה שמאלית יותר.
6. process – פונקציית עזר לסריקות – מדפסה את הערך עם רווח.
7. פונקציות וירטואליות ללא מימוש:
   a. Add – הוספת ערך לעץ. מאחר וכל סוג עץ (חיפוש,AVL וכו') מכיל סדר שונה של העלים והערכים שבהם יש להגדיר אותם רק במחלקת היורשות.
   b. Search – חיפוש ערך בעץ. שוב, פועל אחרת בכל עץ
   c. Remove – מחיקת ערך **בודד** מהעץ



סריקה תוכית – in-order



סריקה סופית – post-order

תחת רמת Private:

שלושת פונקציות הסריקה ופונקצית המחיקה נמצאים תחת רמת פרטיות גבוהה עם בסיס סריקה שונה – לא השורש המקורי של העץ אלא node שמקבל ערכים ומשם מתחיל לסרוק.

יש לשים לב שכל מימושי הפונקציות נעשים במימוש תבניתי – מאחר והתוכנית הנוכחית מתייחסת רק לאב של עץ ועליו נבנה את כל המחלקות היורשות (במקרה הזה עץ חיפוש בינארי) רק את המחלקות שיהיו אחידות בכולם אנחנו נגדיר בצורה תבניתית, ואת הפונקציות שיבוצעו שונה בכל סוג יורש נגדיר בנפרד בכל בן.

שיעורי הבית המייחסות למחלקה הכללית, מוסיפות פונקציות שאינן הכרחיות לכל עץ רגיל ולכן לא הוגדרו מלכתחילה.

```
#include "Tree.h"

template <class T>
class SearchTree : public Tree<T>
{
public:
	// protocol for search trees
	void add(T value);
	bool search(T value)
	{
		return search(root, value);
	}
	void remove(T value);

private:
	void add(Node * current, T val);
	bool search(Node* current, T val);
};
```

```
template <class T>
void SearchTree<T>::add(T val)
{
	// add value to binary search tree
	if (!root)
	{
		root = new Node(val);
		return;
	}
	add(root, val);
}

template <class T>
bool SearchTree<T>::
search(Node * current, T val)
{
	// see if argument value occurs in tree
	if (!current)
		return false; // not found
	if (current->value == val)
		return true;
	if (current->value < val)
	 return search(current->right, val);
	else
		return search(current->left, val);
}

template <class T>
void SearchTree<T>::add(Node* current, T val)
{
	if (current->value < = val)
		// add to right subtree
		if (!current->right)
		{
		 current->right = new Node(val);
			return;
		}
		else add(current->right, val);
	else
		// add to left subtree
		if (!current->left)
		{
			current->left = new
Node(val);
			return;
		}
		else add(current->left, val);
}

template <class T>
void SearchTree<T>::remove(T val)
{
	//HOME WORK!
}
```

# עץ חיפוש בינארי

בתוכנית זו, אנו מסתכלים על מימוש עץ שהוגדר קודם, בצורה של עץ חיפוש בינארי.
הגדרת "עץ חיפוש בינארי" – עץ שכל הילדים השמאליים לאותו קודקוד קטנים ממנו, וכל הימנים גדולים מהקודקוד. לצורך העניין, יכול להיות שגובה העץ הוא N על פי מספר איבריו, כאשר הוא מסודר בצורה כזו שכל האיברים גדולים מהשורש בצורה סדורה (ואז כל העץ ירד ימינה), או שכל האיברים קטנים מהשורש (ואז העץ ירד שמאלה).
ברמת המימוש של העץ – הוא יורש כמובן מהחלקה שנבנתה קודם, כאשר אפילו הקונסטרקטור מוגדר בתור תבנית לעצים שיבואו אחריו.
ברמת המימושים של הפונקציות שהוגדרו כוירטואליות –

1. Add – ראשית מוסיפים ערך לשורש, במידה והערך בשורש כבר קיים, בודקים האם האיבר החדש גדול או קטן מהערך הנמצא ויורדים לאט בין הערכים עד שמגיעים למיקום בו ניתן להכניס את האיבר החדש. (כמובנשמדובר על פונקציה של (log(n)
2. Search – הפונקציה עובדת בצורה רקורסיבית ובודקת החל מהשורש האם האיבר הנוכחי שוה לערך הדרוש, במידה ולא – בודקים את ההסרש האם מדובר בערך גדול או קטן מהנוכחי, ועל פי זה גולשים לעבר העלה הבא.
3. Remove – שיעורי הבית. יש לחפש אם האיבר קיים בעץ, ובמידה שכן, למחוק אותו בצורה שלא תהרוס את כל הרצף של העץ.

יש לשים לב בכל "מכולה" (מימוש של מבנה נתונים דינאמי), קיימות שלושת הפעולות האלה אותם צריך לממש בצורה שונה. גם בדוגמאות הקודמות בתור ובמחסנית היו לנו את הפנקציות האלו כוירטואליות והיה אפשרות לממש אותם בצורה שונה על פי סוג המימוש (וקטור/רשימה דינאמית), ובעצם בכל סוג מכולה חדש הדבר הראשון שאנחנו נדרשים זה לחפש א ת המימושים לבסיסים האלו.

```
#include <iostream>
#include "List.h"
using namespace std;

class IteratorOnList : public List
{
public:
	class iterator
	{
	private:
		Link* current;
		Link* headOfLst;
	public:

		iterator(Link* p, Link* q) :
			current(p), headOfLst(q) {}

		void operator++(int i)
		{
			Link*nextValue = headOfLst;
			while (nextValue &&
		nextValue->value <= current->value)
			nextValue = nextValue->next;
			if (!nextValue)
			{
				current = nullptr;
				return;
			}
			Link*p = nextValue->next;
			while (p)
			{
if (p->value>current->value && p->value <nextValue-
>value)
				nextValue = p;
				p = p->next;
			}
			current = nextValue;
		}

		bool operator!=(iterator rhs)
		{
			return current != rhs.current;
		}

		int operator*()
		{
			return current->value;
		}
	};	//end of class iterator

	iterator begin()
	{
		if (isEmpty()) return end();
		Link*min = head, *p = min->next;
		while (p)
		{
			if (p->value <min->value)
				min = p;
			p = p->next;
		}
		iterator it(min, this->head);
		return it;
	}

	iterator end()
	{
		iterator it(nullptr, nullptr);
		return it;
	}
};	//end of class IteratorOnList


int main()
{
	IteratorOnList lst;
	for (int i = 0; i<10; i++)
	{
		int val = rand() % 100;
		lst.add(val);
		cout << val << ' ';
	}
	cout << endl;
	IteratorOnList::iterator it = lst.begin();
	for (; it != lst.end(); it++)
		cout << *it << ' ';
	cout << endl;
	return 0;
}
```

# איטרטורים

```
for (list<int>::iterator it=lst.begin(); it != lst.end();it++)
```

איטרטורים בפועל זה משהו שאנחנו לא כותבים, אבל תמיד יש לנו איטרטורים מוכנים כמו בספריית סטרינג ווקטור. מה שנמצא פה זה הדגמה לאפשרות כתיבת איטרטור בצורה עצמאית.
מה זה איטרטור? הפירוש המיליולי – הוא למנות סדרה של חפצים או איברים. למשל, יש שורה של תלמידים, ניתן to iterate לעבור אחד אחד על התלמידים ולמנות אותם בצורה מסודרת (למשל בדיקת שמות).
האיטרטור נותן לנו לעבור על האוסף מבלי לדאוג לסדר הפנימי של האיברים. בכל סוג מכולה (container) אנחנו דואגים לעבור על כל המכולה בצורה מסוימת (למשל: סריקת עץ תוכית/סופית), אך לעומתם, האיטרטור עובר בכל פעם לאיבר הבא ללא התחשבות בסוג המכולה. אין צורך לכתוב את סוג המעבר בין האיברים, רשימה מקושרת, מערך ועץ יגיבו באותו אופן ויתנו לנו לראות את האיבר הבא ברשימה על פי דרישה.

בדוגמא לפנינו כוללים את התוכנית של הרשימה "List.h" ומגדירים list<int> lst באשר בעצם מגדירים פה רשימה של אינטים בשם lst. ממלאים את הרשימה בעזרת הפונקציה push_back הקיימת ברשימה, ומגדירים את כמות האופציות – lst.push_back(10), ולאחר מכן מכניסים את הכל בלולאה שנכתבת בצורה טיפה שונה ממה אנחנו רגילים: , for (list<int>::iterator it=lst.begin(); it != lst.end();it++)
הגדרת האיטרטור מגדיר לנו מחלקה פנימית, ולכן כפל הנקודותים (בדומה לגישה לתתי מחלקות), ופועלת בצורה שמקבילה לטמפלייט בדומה עם הסוגרים החדים. ולבסוף מגדירים את שם הטיפוס it מה שנותן ללולאה את מה שאנחנו מגדירים בדרך כלל int i. כך הגדרנו את נקודת ההתחלה של הלולאה, ומשתמשים בפקודו, הstl לשאר הלולאה.

**Begin()**

כל אוסף stl תומך בפקודה begin() המחזיר את האיבר הראשון במכולה.

**End()**

בסוף כל האיברים ברשימת stl קיים בסוף זקיף המשמש מעין 0\ הקיים במחרוזות, שמסמן את סוף הנתונים, כך שהאיטרטור יכול פשוט לרוץ על האיברים עד שהוא מגיע לזקיף ומשם הוא יכול לעצור.

**++**

ה++ מחביא בתוכו את האלגוריתם מעבר לאיבר העוקב ברשימה.
בתוך הלולאה ניתן לבטא פקודת פלט : cout << *it<< endl; כך שיהיה פוינטר מסוג Intעל פי הגדרת האיטרטור והוא יוכל לשלוח למסך את מה שהוגדר לאותו מיקום. וניתן גם להכפיל על ידי פקודה של *it *=2;
אם נעבור מהדוגמא הבסיסית הזו לפולימורפיזם, אם יש לי ליסט של סטודנטים מסוגים שונים, לא ניתן לשים אובייקטים מסוגים שונים באותה רשימה, וכולם צריכים להיות מאותו סוג – למשל * student, בצורה שהשתמשנו בתוכנית עם הוקטור:

```
list <Student * > lstStud;
lstStud.push_back(new BA(...));
for (list<Student * >::iterator it = lstStud.begin(); it != lstStud.end(); it++)
{
	cout <<*( *it )<< endl;

}
```

למעשה אופרטור הפלט חייב להיות במקרה זה וירטואלי, מאחר והגישה אליו במקרה הזה מגיע דרך הstudent, שאין לו שום ביטוי ממשי. ועל ידי השליחה בתוך שתי כוכביות הפולימורפיזם פועל.

פונקציות נוספות שניתן להשתמש בstl:
**Rbegin() –** מפעיל איטרטור המסוגל לחזור אחורה ברשימה ולבדוק איברים בכיוון אחר (במידה וסוג המכולה נותן את האפשרות)ץ
**Cbegin() –** מחזיר את האיבר עליו הוא מצביע בצורה של const כך שלא יקבל שינויים. בצורה זו ניתן גם לעשות את הלולאה שהגדרנו קודם עם auto ללא ציון סוג הטיפוס המדויק
**Crbegin() –** סריקת האיברים מהסוף להתחלה. במקרה כזה, ההתקדמות בלולאה תהיה it++ ולא בפורמט הרגיל.

ברשימות קיימים גם אפשרויות של push_back, push_front וגם פונקציות דומות לPop , למרות שבוקטור לא תהיה אופציה כזאת מאחר שמדובר על רמת עבודה גבוהה מאוד (במידה ויש כמות גדולה של איברים, צריך לעבור כל אחד בנפרד)

**Operator[] –** קפיצה למספר איבר שיוגדר בתוך הסוגריים המרובעות.
**List::insert** – הכנסת איברים בצורה שניתן לקבוע את המיקום בו צריך להכניס את האיברים, והגדרת טווח התחלתי וסופי.
**Remove_if()** – אפשרות למחיקה בעזרת קביעת תנאי. הכנה ללמבדה – ניתן למחוק את כל האיברים המתאימים לקריטריון מסוים המבוטא בlf, הקריטריונים הנקבעים מקבלים איברים שנמצאים ברשימה ברפרנס, ומחזירה בצורה בוליאנית האם האיברים מקיימים את התנאי. למשל - single_digit, is_odd וכדו'. וניתן לשלוח א שם הפונקציה הממיינת וזה מתפקד כמשתנה. בתכנות דבר כזה נקרא "lambda" ונלמד עליו יותר במפגש הבא.
**sizeOf() –** מחזיר גודל של מספר ספרות במערך. אם המערך בגודל 10, אך מוכילם בו רק 3 ספרות, הוא יחזיר 3. לפעמים משתמשים בפקודה ובסוגריים פשוט מגדירים int והוא מחזיר את כמות הבתים במשתנה (4).
הערות לשיעורי בית תרגיל 9:
-בדאי להגדיר את הפונקציה המחזירה את סוג הסטודנט כוירטואלית
- הוקטור או הרשימה צריכים להיות * student

### דוגמא 1

```
#include <iostream>
using namespace std;
int main()
{
	int x = [](int y) {return y*y; }(4);
	cout << x;
	return 0;
}
```

### דוגמא 2

```
#include <iostream>
using namespace std;
int main()
{
	int x;
	[&x](int y) {x = y*y; }(4);
	cout << x;
	return 0;
}
```

### דוגמא 3

```
#include <iostream>
using namespace std;
int main()
{
	int x = 2345;
	int z = [x]() mutable {
		int sum = 0;
		while (x)
		{
			x /= 10;
			sum++;
		}			return sum;
}();
		cout << "x=" << x << endl;
		cout << "z=" << z << endl;
		return 0;
}
```

### דוגמא 4

```
#include <iostream>
using namespace std;
bool isBigger(int i, int j) { return i>j; }
void bubbleSort(int* vec, int size,
bool(*cmp)(int, int))
{
	for (int last = size - 1; last>0; last--)
		for (int i = 0; i<last; i++)
		 if (cmp(vec[i], vec[i + 1])) {
				int temp = vec[i];
				vec[i] = vec[i + 1];
				vec[i + 1] = temp;
			}
}
```

```
int main() {
	int vec[10];
	for (int i = 0; i<10; i++) {
		vec[i] = rand() % 100;
		cout << vec[i] << ' ';
	}
	cout << endl;
	bubbleSort(vec, 10, isBigger);
	for (int i = 0; i<10; i++)
		cout << vec[i] << ' ';
	cout << endl;
	bubbleSort(vec, 10, [](int x, int y)
{return x<y; });
	for (int i = 0; i<10; i++)
		cout << vec[i] << ' ';
	cout << endl;
	return 0;
}
```

### מצביע לפונקציה

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

int addition(int a, int b)
{
	return (a + b);
}
int subtraction(int a, int b)
{
	return (a - b);
}
int(*minus)(int, int) = subtraction;
int operation(int x, int y,
int(*functocall)(int, int))
{
	int g;
	g = (*functocall)(x, y);
 // g = functocall(x,y);
	return (g);
}

int main()
{
	int m, n;
	m = operation(7, 5, addition);
	n = operation(20, m, minus);
	n = operation(20, m, subtraction);
	cout << n;
	return 0;
}
```

# Lambda

```
[](types) mutable[9] { function; }()
```

באשר יש זימון פונקציה ללא סוגריים (דוגמא 4), הזימון נקרא מצביע לפונקציה.
סימן היכר למצביע פונקציה, זה הכוכבית (*) עם שם הפונקציה בעת ההצהרה של הפונקציה. המצביע מתמחה רק לפונקציות עם חתימה ספציפית (bool,int וכדו) יש להתחשב בחתימה גם בפרמטרים המוכנסים לפונקציה ולא רק לסוג המשתנה[10].
כך ניתן לשלוח בפורמט דומה לפונקציה הקיימת כאשר שם המצביע לא חייב להיות השם המקורי של הפונקציה.
אפשרות נוספת לשליחה של הפונקציה מופיעה בהכרזה השניה לפונקציה bubbleSort. הכלילו את הפונקציה בצורה הרבה יותר "פשוטה". במידה והפונקציה שאנחנו רוצים לשלו מצביע עליה היא: 1. קצרה (שורה אחת בלבד), ובמיוחד אם 2. אנחנו רוצים להשתמש בה רק פעם אחת, ואין עוד חזרה על סדר הפעולות הנוכחי, ניתן להגדיר בקריאה מיוחדת שמגדירה את הפונקציה על המקום עם המימוש ללא שם. ביטוי כזה נקרא **lambda** (לעיתים נקרא גם "פונקציה אנונימית"). משתנה שתפקידו להתייחס לפונקציה. ללמבדה אין שם לפונקציה אלא נשלחת ריקה עם סוגריים מרובעות [] עם הפרמטרים הרלוונטיים ובסוגריים מסולסלים את הפעולה בשלמותה
למעשה, השימוש של המצביעים והלמבדה לפונקציות, הם שימוש בפונקציות בתור משתנים, אך איך הקומפיילר מבין שמדובר בפונקציה בוליאנית או אחרת? הרי אם שם הפונקציה לא כתוב, גם לא כתוב מה הוא מחזיר.
למעשה, הקומפיילר מסתכל על ההכרזה הקיימת בו כבר מקודם, והקומפיילר בודק את סוג הפונקציה.
נחזור לדוגמאות הראשונות- דוגמא 1. המשתנה X מוגדר בתור פונקציית למבדה המקבל ערך בודד ועושה עליו פעולה, ובסופו מוגדר בסוגריים עגולות המספר הנכנס לפונקציה. והכנסת הערך מוגדרת בזימון של הפונקציה המפעילה את הלמבדה.
על ידי הגדרת הערך המוכנס כInt (ולמעשה גם להצבת התוצאה בint) הקומפיילר מזהה פה את סוג הערך המוחזר. במידה ואין הגדרה ממשית, אז הפונקציה עובדת על הטיפוס הרחב יותר ( ,float>int double>int) בצורה דומה לזו שהוא מחליט מה להגדיר בשמגדרים טיפוס מסוג Auto.
הלמבדה לא חייב ברמה הטכנית להחזיר ערך כלשהו, אך ברוב המקרים משתמשים בפונקציה על מנת להחזיר ערך.
בדוגמא 2 ניתן לראות שהסוגריים המרובעות לא חייבות להיות ריקות. ניתן לסמן בהם משתני שמוברים לסקופ, שהוצהרו כבר מקודם, ואז הביטוי למבדה תופס את המשתנים האלה והוא יכול לעבוד על המשתנים האלה, ואפילו ניתן לשלוח את המשתנים by reference עם & וכך לשנות את הערך בעצמו בעזרת הלמבדה.
בצורה כזאת אנחנו מכניסים לתוך הסקופ של הלמבדה, משתנה שלא מוגדר בסוגריים של הטיפוס המוכנס לפונקציה וניתן לעבוד עליו בפנים. דוגמא זו היא מקבילה לדוגמא הראשונה רק שבמקום לקבל ערך מוחזר למשתנה מחוץ לפונקציה, אנחנו עושים את השינוי בתוך הפונקציה (למעשה מדובר פה על פונקציית void).

[9] אופציונלי (לא חובה)
[10] סדר ההכרזה הוא כזה: . טיפוס של הפונקציה (*מזהה=שם הפונקציה) (רשימת טיפוסים להכנסה)

דוגמא 3 ערך הX מאותחל מראש, והמבדה לוקחת את המשתנה ללא שום פרמטר נוסף (המשתנה לא עובר by referenc)
הגדרת **mutable** החריגה את הערכים, כך שגם אם הם יישלחו בתור const ניתן יהיה לעשות בהם שינוי בתוך הלמבדה – הם לא ישונו מחוץ לפונקציה עצמה, אך אם לא יוגדרו בmutable ויישלחו בתור קונסט, לא ניתן יהיה להשתמש בהם אפילו בתוך הלמבדה.
שני הפרמטרים הראשונים המתקבלים בפונקציה הם פרמטרים רגילים, והפרמטר השלישי הוא קריאה למצביע על פונקציה כך שנשלח בעצם בתוכנית הזאת, את המשתנים ואת הפונקציה המיוחדת אותה אנחנו רוצים לבצע. הפונקציה operation מנתבת לנו את הפונקציות אחת אחרי השנייה על פי מה שמתבקש בתוכנית.
בדוגמא 4 קיימת פונקציית מיון בועות המקבלת מערך ואת גודלו, ובדיקה המשווה את שני האיברים שנשלחו. (כאשר הפונקציה מחזירה 0 כששני האיברים שווים). ההשוואה היא בעצם מצביע לפונקציה (ולכן זה נשלח בצורה של מצביע לפני שם הפונקציה – (cmp*)) ,כאשר בתוכנית הראשית ניתן לשלוח ישירות את שם הפונקציה isBigger והפונקציה משווה כל שני איברים סמוכים.
למעשה, יש שני קריאות לפונקציה של המיון בועות, כך שבזימון הראשון הפונקציה מזמנת את isBigger, ובפעם השנייה משתמשים בביטוי למבדה שחוסך את הקפיצה לפונקציה, למרות שהוא עושה בדיוק את אותה הפעולה.

## הגדרות syntax לפונקציות למבדה:

**[]** אם לא שולחים אף משתנה בתוך הסוגריים המרובעות, אין שום שימוש במשתנים שהם חיצוניים ללמבדה.
אם רוצים להכניס את המשתנים, אפשר להכניס את שמות הערכים, אך הם יהיו const ולכן לא ניתן לשנות אותם ולערוך אלא אם יישלחו By reference (&) או Mutable.
אם יהיה התנגשות בין שמות משתנים, העדיפות הגבוהה היותר היא שימוש בשם המשתנה המקומי ולא החיצוני ללמבדה.
ניתן גם להכניס לסוגריים רק &, וכך הוא יתפוס את כל המשתנים הקיימים ויעביר אותם by reference.
הכנסה של '=' לסוגריים תופס את כל המשתנים, אך ללא אפשרות לשינוי (אלא אם מגדירים כmutable),
כמו כן, ניתן להוסיף ערך ספציפי שיעבור by reference – [=,&z]
**{}** – הפקודות הנעשות בלמבדה
**()** – הגדרת ברירת מחדל למשתנה מקומי המוגדר בין הסוגריים המרובעות למסולסלות, הסוגריים מגיעות גם בסוף ההגדרה של הלמבדה, אנחנו לא נזמן את הפונקציה והתוכנית תדלג מעל כל מה שכתוב בפנים (כאשר הלמבדה מוגדרת בתוך איטרטור, אין צורך בסוגריים עגולות בסוף הפונקציה מאחר שהזימון נעשה בכל פעם בעזרת האיטרטור.
Mutable – החרגת הערכים כך שיהיו ניתנים לשינוי. מופיע לפני הסוגריים המסולסלות

**דוגמא 1**

```
// min example
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
	cout << "min(1,2)==";
	cout << min(1, 2) << '\n';
	cout << "min(2,1)==";
	cout << std::min(2, 1) << '\n';
	cout << "min('a','z')==";
	cout << std::min('a', 'z') << '\n';
	cout << "min(3.14,2.72)=="
		cout << min(3.14, 2.72) << '\n';
	return 0;
}
```

**דוגמא 2**

```
// remove algorithm example
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
 int myints[] = { 10,20,30,30,20,10,10,20 };
	// bounds of range:
	int* pbegin = myints;
	int* pend = myints + 8;
	pend = remove(pbegin, pend, 20);
	cout << "range contains:";
	for (int* p = pbegin; p != pend; ++p)
		cout << ' ' << *p;
	return 0;
}
```

**דוגמא 3**

```
// remove_if & remove_copy example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
 int myints[] = { 9,20,35,30,20,12,10,20 };
	int* pbegin = myints;
	int* pend = myints + 8;
	pend = remove_if(pbegin, pend,
		[](int x) {return x % 3 == 0; });
	cout << "range contains:";
	for (int* p = pbegin; p != pend; ++p)
		cout << ' ' << *p;
	cout << endl;
	vector<int> myvector(pend - pbegin);
	remove_copy(myints, myints +
	 (pend - pbegin), myvector.begin(), 20);
	cout << "myvector contains:";
	for (auto it = myvector.begin();
		it != myvector.end(); ++it)
		cout << ' ' << *it;
	return 0;
}
// count/countif algorithm example
```

**דוגמא 4**

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool IsOdd(int i) { return ((i % 2) == 1); }
int main() {
	// counting elements in array:
	int myints[] = { 5,10,15,20,15,10,5,10 };
// 8 elements
 int mycount = count(myints, myints + 8, 10);
	cout << "10 appears " << mycount << "
times.\n";
	// counting elements in container:
	vector<int> myvector(myints, myints + 8);
	mycount = count(myvector.begin(),
myvector.end(), 20);
	cout << "20 appears " << mycount << "
times.\n";
	mycount = count_if(myvector.begin(),
myvector.end(), IsOdd);
	cout << "myvector contains " << mycount
<< " odd values.";
	return 0;
}
```

**דוגמא 5**

```
// all_of example
#include <iostream>
#include <algorithm>
#include <array>
using namespace std;
bool divBy3(int i) { return i % 3 == 0; }
int main() {
	array<int, 8> A = { 3,21,33,9,15,27,39,45
};
	if (all_of(A.begin(), A.end(), divBy3))
		cout << "all numbers divid by
3.\n";
	if (all_of(A.begin(), A.end(), [](int i)
{return i % 2; }))
		cout << "odd numbers only.\n";
	return 0;
}
```

## דוגמא 6

```
// any_of example
#include <iostream> // std::cout
#include <algorithm> // std::any_of
#include <array> // std::array
using namespace std;
int main() {
	array<int, 7> A = { 0,1,-1,3,-3,5,-5 };
	if (any_of(A.begin(), A.end(),
		[](int i) {return i<0; }))
		cout << "array includes negative
elements\n";
	return 0;
}
```

## דוגמא 7

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void display(vector<int> vec) {
	for_each(vec.begin(), vec.end(),
		[](int s) {cout << s << ' '; });
	cout << endl;
}
int main()
{
	vector<int> vec;
	for (int i = 0; i<10; i++)
		vec.push_back(rand() % 100);
	display(vec);
	sort(vec.begin(), vec.end());
	display(vec);
	sort(vec.begin(), vec.end(),
		[](int i, int j) {return i>j; });
	display(vec);
	return 0;
}
/*Output:
41 67 34 0 69 24 78 58 62 64
0 24 34 41 58 62 64 67 69 78
78 69 67 64 62 58 41 34 24 0
*/
```

# ספריית אלגוריתמים (STL)

יש ספריית אלגוריתמים אותם ניתן להכליל בתוכנית. נעבור כרגע על מספר פונקציות הקיימות בספריה.
דוגמא 1 מכילה מספר פונקציות של **min** ו-**max** , שעליהם אין מה להרחיב. החידוש מתחיל בדוגמא 2 עם הפונקציה, המיוחד בפונקציה היא , שהיא עובדת גם על רשימות וגם על ווקטורים ללא תלות בסוג הקונטיינר. הגדרת הפונקציה נעשית על ידי הכרזת התחום בו אנחנו מבצעים את המחיקה (במקרה הזה מההתחלה לסוף כמשתמע מהפונקציות) ולאחר מכן מכניסים את הערך אותו רוצים למחוק.
על פי הגדרת הפונקציה remove היא מקבלת כל טיפוס המקבל ++ ו* וכך על ידי הלולאה מקדמת את האיטרטור עד שהוא מגיע לקצה הטווח ולמחוק את הדרוש.
דוגמא 3 – **removeif** שני הפרמטרים הראשונים הם דומים לקודם, אך הביטוי השלישי הוא מצביע לפונקציה או ביטוי למבדה (בידה ומדובר על שימוש יחידאי בפונקציה). הפונקציה בעצם מכריזה על תנאי מסוים שכל ערך בקונטיינר המקיים את התנאי הבוליאני נמחק.
יוצרים ווקטור המקבל את המערך של int ומאתחלים את הגודל שלו על פי המערך. יש פה ייחוד בהגדרה על ידי אריתמטיקה על פוינטרים, בו ההפרש מוגדר בתור סוף המערך פחות ההתחלה. יש לזכור שברגע שלא מוגדר ערך לוקטור, כל האיברים שבפנים מוגדרים כ-0.
**Remove_copy** מחיקה של ערכים כפולים - מקבלת את ערכי תחילת הפונקציה (בתור מצביע שהוגדר ל Pbegin, ותנאי הסיום הוא ההתחלה בחיבור ההפרש לסוף המערך (כך שבעצ ם מגדיר את כולו) צורת ההעתקה עוברת בוקטור ומעתיקה את כולו מלבד מה שהוגדר לה.
דוגמא 4 – **count** – סופר הופעות של משתנה מסוים. **Count_if** סופר על פי קריטריון עם מצביע לפונקציה או ביטוי למבדה[11]. במקרה זה מוסיפים את האפשרות לספור את ה-**isOdd** והוא סופר את כל האי-זוגיים.
בדוגמא 5 יש אופציה הנקאת allOf המחזירה ערך true רק במידה וכל האיברים באוסף המוגדר בתחום מקיימים את התנאי – במקרה זה **divBy**, דוגמא 6 מתייחסת לשליחת אמת של **anyOf** - ברגע שיש איבר אחד המקיים את התנאי הפונקציה תחזיר אמת. ובדוגמא 7 **forEach** הוא פונקציה שימושית הנותנת לבצע שינוי בכל האיברים בתחום הקיים, ועליו ניתן לבצע שינויים או הדפסה של כל האיברים וכדו'. יש אפשרות גם לעשות **sort** המקבל תחום של מערך והוא ממיין אותם. כמובן שניתן לשלוח כל סוג קונטיינר המקבל ++ ו* (רשימות וכו') ולמיין את הכל.
המיון השני שהוגדר בפונקציה משתמש בביטוי למבדה שנותן לו את האפשרות למיין בסדר הפוך . (ברירת המחדל היא בסדר עולה, ובאן ניתן למיין גם בסדר יורד)

[11] התנאי ניקרא predicate – פונקציה שמתייחסת לאיברים שבתחום האוסף.

# פונקציות STL

## וקטורים

יש להכליל ספריית וקטור #include <vector>

| פונקציה | פעולה |
|---|---|
| `vector<T> v;` | יצירת וקטור ריק |
| `vector<T> v(n);` | יצירת וקטור בגודל מוגדר |
| `vector<T> v(n, value);` | יצירת וקטור מוגדר בגודל, בעל ערכים מאותחלים |
| `vector<T> v(begin,end);` | יצירת וקטור והעתקת הערכים בתטווח מסוים |
| **גישה לוקטור** | |
| `v.size()` | מחזיר את גודל הוקטור (הערכים שבבפים ולא הגודל המוגדר) |
| `v.empty()` | מחזיר אמת אם המערך ריק |
| `v.begin()` | מעביר את המצביע לערך ההתחלתי שלו |
| `v.end()` | מעביר את המצביע סוף הוקטור |
| `v.front()` | מחזיר את הערך הראשון |
| `v.back()` | מחזיר את הערך האחרון |
| `v.capacity()` | מחזיר את ערך האיברים המקסימלי האפשרי |
| **שינוי ערכים** | |
| `v.push_back(value)` | הכנסת ערך לסוף הוקטור |
| `v.insert(iterator, value)` | הכנסת ערך למקום על פי האיטרטור |
| `v.pop_back()` | מחיקה של הערך האחרון |
| `v.erase(iterator)` | מחיקת הערך המוגדר במיקום האיטרטור |
| `v.erase(begin, end)` | מחיקת טווח האיברים המוגדר |

## רשימה דו-כיוונית

יש להכליל את ספריית רשימה #include <deque>

| פונקציה | פעולה |
|---|---|
| `deque<T> d;` | יצירת רשימה ריק |
| `deque<T> d(n);` | יצירת רשימה בגודל n |
| `deque<T> d(n, value);` | יצירת רשימה בגודל N עם ערכים מאותחים |
| `deque<T> d(begin, end);` | יצירת רשימה והעתקת ערכים בטווח האיטרטור |
| **גישה לתור** | |
| `d.size()` | מחזיר את גודל הרשימה המובלת |
| `d.empty()` | מחזיר אמת אם הרשימה ריקה |
| `d.begin()` | מעביר את המצביע לערך ההתחלתי שלו |
| `d.end()` | מעביר את המצביע לסוף התור |
| `d.front()` | מחזיר את הערך הראשון |
| `d.back()` | מחזיר את הערך האחרון |
| `d.capacity()` | מחזיר את ערך האיברים המקסימלי האפשרי |
| **שינוי ערכים** | |
| `d.push_front(value)` | הכנסת ערך לתחילת הרשימה |
| `d.push_back(value)` | הכנסת ערך לסוף התור |
| `d.insert(iterator, value)` | הכנסת ערך למקום על פי האיטרטור |
| `d.pop_back()` | מחיקה של הערך האחרון |
| `d.erase(iterator)` | מחיקת הערך המוגדר במיקום האיטרטור |
| `d.erase(begin, end)` | מחיקת טווח האיברים המוגדר |

## רשימה

יש להכליל את הרשימה #include <list>

| פונקציה | פעולה |
|---|---|
| `list<T> l;` | יצירת רשימה |
| `list<T> l(begin, end);` | יצירת רשימה והעתקת ערכים בטווח האיטרטור |
| **גישה לרשימה** | |
| `l.size()` | מחזיר את מספר האיברים ברשימה |
| `l.empty()` | מחזיר אמת אם הרשימה ריקה |
| `l.begin()` | מעביר את המצביע לערך הראשון |
| `l.end()` | מעביר את המצביע לסוף הרשימה |
| `l.front()` | מחזיר את הערך הראשון |
| `l.back()` | מחזיר את הערך האחרון |
| **שינוי ערכים** | |
| `l.push_front(value)` | הכנסת ערך לתחילת הרשימה |
| `l.push_back(value)` | הכנסת ערך לסוף הרשימה |
| `l.insert(iterator, value)` | הכנסת ערך למקום על פי האיטרטור |
| `l.pop_front()` | מחיקת האיבר הראשון |

| | |
|---|---|
| `l.pop_back()` | מחיקה של הערך האחרון |
| `l.erase(iterator)` | מחיקת הערך המוגדר במיקום האיטרטור |
| `l.erase(begin, end)` | מחיקת טווח האיברים המוגדר |
| `l.remove(value)` | להוריד את כל המופעים בערך המוגדר |
| `l.remove_if(test)` | מחיקת כל האיברים המקיימים את התנאי |
| `l.reverse()` | הפיכת סדר הרשימה |
| `l.sort()` | מיון הרשימה |
| `l.sort(comparison)` | מיון השווה |
| `l.merge(l2)` | מיזוג רשימות ממוינות |

## מחסנית

יש להכליל את המחסנית #include <stck>

| פונקציה | פעולה |
|---|---|
| `stack<T> s;` | יצירת מחסנית ריקה |
| `stack<T, container<T> > s;` | יצירת מחסנית בהתבסס על מבנה נתונים קיים |
| **גישה לרשימה** | |
| `s.top()` | החזרת הערך העליון במחסנית |
| `s.size()` | מחזיר את מספר האיברים במחסנית |
| `s.empty()` | מחזיר אמת אם המחסנית ריקה |
| **שינוי ערכים** | |
| `s.push(value)` | הכנסת ערך למחסנית |
| `s.pop()` | הוצאת האיבר הראשון |

## תור

יש להכליל את ספריית התור #include <queue>

| פונקציה | פעולה |
|---|---|
| `queue<T> q;` | יצירת תור ריק |
| `queue<T, container<T> > q;` | יצירת תור בהתבסס על מבנה נתונים קיים |
| **גישה לרשימה** | |
| `q.front()` | החזרת הערך בקדמת התור |
| `q.back()` | החזרת הערך בסוף התור |
| `q.size()` | מחזיר את מספר האיברים בתור |
| `q.empty()` | מחזיר אמת אם התור ריק |
| **שינוי ערכים** | |
| `q.push(value)` | הכנסת ערך לתור |
| `q.pop()` | הוצאת האיבר הקדמי ביותר |

## תור קדימיות

יש להכיל את ספריית תור-קדימיות #include <queue>
זה אותה ספריה כמו תור רגיל.

| פונקציה | פעולה |
|---|---|
| `priority_queue <T, container<T>, comparison<T> > q;` | יצירת תור קדימיות והגדרת הבסיסים להשוואה |
| **גישה לרשימה** | |
| `q.top()` | החזרת הערך בעדיפות הגבוהה ביותר |
| `q.size()` | מחזיר את מספר האיברים בתור |
| `q.empty()` | מחזיר אמת אם התור ריק |
| **שינוי ערכים** | |
| `s.push(value)` | הכנסת ערך לתור |
| `s.pop()` | הוצאת האיבר בעדיפות הגבוהה ביותר |

## אלגוריתמים

הכללה תחת #include <algorithm>

| פונקציה | פעולה |
|---|---|
| all_of | מחזיר אמ אם **כל** הערכים מקיימים תנאי |
| any_of | מחזיר אמת אם אחד מהערכים מקיים את התנאי |
| none_of | מחזיר אמת אם **אף** איבר לא מקיים את התנאי |
| for_each | מבצע פעולה עבור **כל** איבר שמקיים תנאי |
| find | מחזיר מצביע לאיבר הראשון הנדרש |
| find_if_not | מחזיר מצביע לאיבר הראשון **המקיים תנאי;** |
| Count | סופר מופעים של ערך |
| Count_if | סופר מופעים המקיימים תנאי |
| Search | מחפש איברים בתחום שהוגדר ממערך אחד לאחר ומחזיר מצביע למיקום |
| **שינוי ערכים** | |
| Copy | מעתיק ערכים מהטווח שהוגדר למיקום |
| Copy_n | מעתיק מספר ערכים N |
| copy_if | מעתיק איברים המקיימים תנאי |
| copy_backward | מעתיק ערכים בסדר הפוך |
| swap | החלפת שני ערכים |
| transform | שינוי ערכים בטווח על פי הגדרה |
| replace | החלפת ערך מסוים בערך מוגדר |
| replace_if | החלפת ערכים המקיימים תנאי |
| Fill | הצבת ערכים (דריסה) בטווח מוגדר |
| remove | הסרת ערכים מוגדרים בטווח |
| remove_if | הסרת ערכים המקיימם תנאי |
| unique | משאיר מופע ראשון מכל האיברים הכפולים |
| reverse | הפיכת הסדר |
| rotate | שינוי הסדר על פי הגדרת טווח כ"אמצע" וערבוב החלקים |
| random_shuffle | ערבוב אקראי של המספרים |
| Partition | מחלק מערך ל2 על פי הגדרות |
| Merge | מיזוג מערכים ממוינים |
| min | מחזיר את הערך הנמוך ביותר |
| Max | מחזיר את הערך הגבוה ביותר |
| minmax | מחזיר את הערך הנמוך ואת הגבוה ביותר |
| lexicographical_compare | מיון לקסיקוגרפי מהקטן לגדול |

## דוגמא 1

```
#include <iostream >
#include <fstream>
using namespace std;
int main() {
	ifstream f1;  ofstream f2;
	char name[10];	float grade;
	f1.open("students.txt");
	if (!f1) {
		cout << "File could not be
opened.\n";
		return 0;
	}
	f2.open("grades.txt");
	if (!f2) {
		cout << "File could not be
opened.\n";
		return 0;
	}
	do {
		f1 >> name;
		cout << "enter " << name << "'s grade
";
		cin >> grade;
		f2 << name << '\t' << grade << endl;
	} while (!f1.eof());
	f1.close();   f2.close();
	return 0;
}
}
```

## דוגמא 2

```
#include <iostream>
#include <fstream>
#include<string>
using namespace std;
int main() {
	string fname, lname;
	int age;
	ifstream in("kelet.txt");
	if (!in) {
		cout << "could not open file.\n";
		return 0;
	}
	while (in >> fname >> lname >> age)
	{
		cout << fname << ' ' << lname <<
'\n';
		if (age >20)
			cout << "can work";
		else
			cout << "can't work";
	}
	return 0;
}
```

## דוגמא 3

```
#include <iostream >
#include <fstream
using namespace std;
struct workers {
	long id;
	char name[15];
	float hours;
	float salary;
};
int main() {
	ofstream f1;
	f1.open("workers.txt");
	workers worker;
	for (int i = 0; i<3; i++) {
		cout << "enter workers name ";
		cin >> worker.name;
		cout << "enter " << worker.name <<
"'s id ";
		cin >> worker.id;
		cout << "enter numbet of hours ";
		cin >> worker.hours;
		cout << "enter a salary per hour ";
		cin >> worker.salary;
		f1.write((char *)&worker,
sizeof(workers));
	}
	f1.close();
	ifstream f2("workers.txt");
	f2.read((char *)&worker, sizeof(workers));
	while (!f2.eof()) {
		cout << worker.name << endl;
		f2.read((char *)&worker,
sizeof(workers));
	}
	return 0;
}
```

## דוגמא 4

```
#include <iostream>


#include <fstream>
using namespace std;
struct workers {
	long id;
	char name[15];
	float hours;
	float salary;
};
int main()
{
	workers worker;
	ifstream f2("workers.txt");
	if (!f2) {
		cout << "could not open the file\n";
		return 0;
	}
	f2.read((char *)&worker, sizeof(workers));
	while (!f2.eof()) {
		float salary;
		salary = worker.hours*worker.salary;
		if (worker.hours>45)
			salary += (worker.hours - 45)
* 0.5 * worker.salary;
		cout << worker.name << '\t' <<
worker.id
			<< "\tsalary: " << salary <<
endl;
		f2.read((char *)&worker,
sizeof(workers));
	}
	return 0;
}
```

## דוגמא 5

```
#include <iostream >
#include <fstream>
using namespace std;
int main() {
	char line[150];
	int i = 1;
ifstream txt;
	txt.open("text.txt");
	txt.getline(line, 150, '\n');
	while (!txt.eof()) {
		cout << i << ":\t" << line << endl;
		txt.getline(line, 150);
		i++;
	}
	return 0;
}
int main() {
	int line = 0, words = 0, chars = 0;
	ifstream txt;
	txt.open("students.txt");
	char c = txt.get(), tav = ' ';
	while (!txt.eof()) {
		if (c != ' ' && c != '\t' && c !=
'\n') chars++;
		if (c == ' ' && c == '\t' && c ==
'\n' && tav != ' ' && tav != '\t' && tav != '\n')
words++;
		if (c == '\n' && tav != '\n') {
			line++;	words++;
		}
		tav = c;	c = txt.get();
	}
	cout << "# of characters: " << chars <<
endl;
	cout << "# of words: " << words << endl;
	cout << "# of lines: " << line << endl;
	return 0;
}
```

# קבצים

```
ofstream outFile ;
outFile.open("myfile.txt",ios::out|ios::app);
```

יש שתי שורות עבודה עם קבצים – קלט/פלט שהכרנו כcout cin שהם עוצרים בהגעה לרווח. ואפשר לעבור בצורה של קריאה/כתיבה ולעבור על רשומה שלמה של מבנה על מחלקה.
שולחים בהתחלה מצביע לערך הראשון מסוג char (המשמש בעצם כמצביע byte), ובפרמטר השני כמה בתים רוצים לקרוא או לכתוב.
המושג העיקרי בעניין עבודה עם קבצים הוא הזרם "stream" ולא רק בC++ אלא בכלל. בזרם רואים את נתוני הקובץ בתור מערך חד מימדי. מה שחשוב, שברגע שאנחנו קוראים נתונים המקום שממנו ייקראו הנתונים הבאים, מתקדם בצורה אוטומטית. כלומר, יש לנו קונספציה של "טייפ" – מתקדמים לאט לאט עם הסרט. במחשבים יש אפשרות לממש זיכרון קבוע לאורך זמן בצורה דומה לסלילי סרט.
מה שצריך להבין שמדובר על צורה חד כיוונית – ניתן לקרוא בשטף ולא לדלג ממקום למקום.
באופן דומה, אלא אם יוגדר אחרת, כאשר כותבים לקובץ, המצביע נמצא במקום בו סיימו לכתוב בפעם האחרונה, וכן בעת הקריאה.
מבחינת הכרזת פתיחת הקובץ, שדה הios הראשון מתייחס ל-in\out האם אנחנו רוצים לבצע קלט או פלט מהקובץ. השדה השני מגדיר את סוג הפתיחה של הקובץ בשביל לדעת באיזה אופן אנחנו עובדים עם הקובץ. ניתן גם להגדיר מספר העמסות של פתיחה כאשר מכניסים קו ישר '|' ביניהם.

המימושים השונים (ios)הם:

**app** – (קיצור של append – להוסיף) מעביר את המצביע של הכתיבה לסוף הקובץ בכל פעם מחדש.
**ate** – (קיצור של **at e**nd – בסוף) מעביר את המצביע באופן חד פעמי לסוף הפונקציה.
**binary** – כתיבה של כל הקובץ ביחידה אחת באופן בינארי ולא כטקסט.
**trunc** – (קיצור של **trunc**ate – קיצוץ) פתיחה של קובץ ריק. אם היה קים קובץ עם שם דומה, הוא נמחק. (מתאים לכתיבה ולא לקריאה)
**nocreate** - דרישה שהקובץ אותו רוצים לפתוח יהיה קיים, אם לא יהיה קיים תצא הודעת שגיאה. (מתאים לקריאה)
**noreplace** - דרישה שהקובץ לא יהיה קיים (מתאים לכתיבה)

**Ifstream/ofstream** – מחלקות המשויכות בספריית הקבצים ושימוש להצהיר על הקובץ בתור פלט קריאה או כתיבה של הקובץ. ההכרזה עליהם לא הופכת אותם עדיין למקושרים לקבץ מסוים, אלא הם מכריזים על אובייקט מסוים שאותו ניתן לקשר לקובץ. ברגע שמקשרים אותם לקובץ, ניתן להפעיל עליהם את הפונקציות הרלוונטיות על הקובץ. וכן, המשתנים המוגדרים (בודגמא f1,f2 1–) הם שמורים בRAM. הקבצים הנפתחים מחוברים למשתנים עד שמכריזים על סגירה שלהם.
בעת הקישור של הקובץ למשתנה מכניסים לו את המיקום המדויק של הקובץ. אם לא מגדירים לו מקום ספציפי, הוא לוקח את הקובץ מהתיקייה המקומית שלו. כמו כן, יש להקפיד על קו נטוי כפול במעבר הנתיב בין ספריות[12].

[12] לדוגמא: "c:\\myfiles\\fileName.txt"

לאחר הקישור של הקובץ בודקים שהקובץ בכלל קיים על ידי התנאי: "if(!f1)" הבודק שבכלל יש לנוו אפשרות לקלט/פלט. קובץ שרוצים לקרוא ממנו חייב להיות קיים בתיקייה. קובץ שרוצים לכתוב עליו ולא יהיה בתיקיית היעד, ייווצר על ידי התוכנית בשם שקראנו לו בתוכנית וגודלו הראשוני יהיה 0.
פעולת פתיחה יכולה להיכשל במקרים הבאים:

- נתיב לא קיים – חלק מהנתיב לא קיים או שאין הרשאות מתאימות לגישה בכל הנתיב.
- הרשאות קובץ – קובץ שמוגדר לקראה בלבד ואנחנו מנסים לפתוח אותו לכתיבה.
- אם מבקשים קובץ לקריאה כתיבה ביחד – והקובץ לא קיים, הוא לא ייצור כמו בכתיבה בלבד, אלא יוציא שגיאה.
- אם היה בו תוכן – ואנחנו ביקשנו קובץ לכתיבה – כל התוכן שהיה קיים בו נמחק, והקובץ מתאפס[13].

על מנת להימנע ממחיקה של הקובץ פותחים את הקובץ כ- fstream. אם פותחים fstream, יש לשלוח גם אופן גישה אחרי שם הקובץ, ומגדירים- ("fileNemr.dat", ios::in) - מוסיפים שדה סטטי המאפשר לנו קריאה, ואם רוצים גם כתיבה מוסיפים בהגדרה ("filename.dat", ios::in|ios::out).
על מנת לקרוא ולכתוב מהקבצים משתמשים בפקודות של >> << כמו שאנו רגילים בcin וcout כאשר צריך להגדיר שיקרא ישירות לתוך המשתנה אליו רוצים לכתוב. אין אפשרות להעתיק ישירות מקובץ אחד לשני, אלא יש להשתמש במשתנה ביניים. יש לשים לב, שאם מעבירים מחרוזות למשתנים, צריך להגדיר את המשתנה בגדול מתאים בשביל לקבל את גודל המחרוזת הנכנסת. בדוגמא 1 – השם מוגדר כמשתנה בגודל [10] ולכן לא ייכנס אליו שם גדול יותר, ואף יגרום בעיות אם יהיה אחד כזה.

## **פונקציות נוספות:**

**seekP** - מחפשת את המקום בו הסתיימה הכתיבה, ובהתאם הפונקציה seekG מחפשת את מקום הקריאה האחרון. אפשר גם להגיד לפונקציה לחפש ביחס לתחילת הקובץ או לסופו, וכך גם לברר את גודל הקובץ[14].

**getline** – קריאה של הקובץ בהצבת תנאים של גודל קובץ או עד לתו מסוים. למשל – f1,getline(name, 30 '\n') במקרה כזה הוא יקרא עד התו ה30 או עד שיגיע לתו שהוגדר - \n. וכן ניתן להגדיר כל תו שרוצים לעצור בו[15]. הגדרת סוף השורה מסייעת מאחר ואנחנו לא יודעים בדיוק איפה מוגדר סוף השורה, ולכן אנחנו לא יכולים לסרוק את השורה בצורה שהיא לא סדרתית. יש לזכור שהגדרה של מספר תוים, הוא בעצם תו אחד פחות, כאשר התו האחרון הוא '\n'.
**eof**() – בדיקה בולייאנית שלא הגענו לסוף הקובץ. ברגע שהגענו לסוף הקובץ, הפונקציה מוציאה true.
**write** (דוגמא 5)– העתקת זיכרון בצורה בינארית, קריאה ישירה מקובץ אחד לקובץ שני. הערכים העוברים מתייחסים לייצוג הבינארי של כל משתנה – למשל, משתנה int תופס ארבעה בתים בזיכרון, וכך הוא גם יעבור. לעומת קריאה טקסטואלית שמתייחסת לכל אות ומספר בצורה בה היא נקראת על ידי המשתמש. הפונקציה מקבלת 2 פרמטרים, טיפוס המוגדר בגודל בייט (*char), הפרמטר השני הוא גודל הזיכרון בבתים אותו יש להעתיק.
**read** – לוקחת שני פרמטרים באופן דומה לפונקציית הכתיבה, ומעתיקה את הקובץ מהזיכרון החיצוני לזכרון הפנימי. הקריאה הראשונה מתבצעת מחוץ ללולאה, ולאחריה ממשיכים לולאת while עד שייגמר

---

[14] על ידי השמת הסמן בסוף הקובץ ושליחת שאלה של מה גודל הקובץ עד לכאן.
[15] ישנם תוים המוגדרים כתוי-סוף-שורה (end of line characters) למשל בוורד יש שני תווים הנקראים (carriage return) CR וLF(line feed), וביוניקס ולינוקס משתמשים רק בLF לעומת מאק שמשתמש בCR. כאשר כולם בעצם מתפרשים ברמת המחשב בתור התו '\n', כך שבעצם בכל מערכת יש גודל שונה לשורות, אך במחינת הקבצים הבינאריים יש סוף מוגדר לכל שורה המוגדר ב\n

כל המידע מהקובץ. (על ידי הפונקציה eof). דבר זה נועד לבדוק לפני שמתחילים לעבור על כל הקובץ שבאמת יש תוכן בקובץ ולא מדובר בקובץ שהגיע ריק מלכתחילה.
get – התנאי המשולש המופיע בדוגמא 6, מתייחס לקריאת תווים שמשתמשים בהם לסימון של סוף מילה, ורק אז מעבר למילה הבאה – אנו יוצאים מנקודת הנחה שאכן יש הבדל אמיתי בין מילה למילה, והפונקציה גט סופרת את מספר המילים.

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

class ClientData
{
private:
	int accountNumber;
	char name[15];
	double balance;
public:
	ClientData(int accountNum = 0, string naMe =
"", double balaNce = 0.0);
	void setAccountNumber(int accountNum);
	int getAccountNumber() const;
	void setName(string tName);
	string getName() const;
	void setBalance(double balanceValue);
	double getBalance() const;
	friend ostream& operator<< (ostream&,
ClientData&);
}; // end class CientData


ClientData::ClientData(int accountNum, string Name,
double Balance)
{
	setAccountNumber(accountNum);
	setName(Name);
	setBalance(Balance);
}
void ClientData::setAccountNumber(int accountNum)
{
	accountNumber = accountNum;
}
int ClientData::getAccountNumber() const
{
	return accountNumber;
}
void ClientData::setName(string Name)
{
	strcpy(name, Name.c_str());
}
string ClientData::getName() const
{
	return name;
}
void ClientData::setBalance(double balanceValue)
{
	balance = balanceValue;
}
double ClientData::getBalance() const
{
	return balance;
}
```

```
ostream& operator<<(ostream& os, ClientData& client)
{
	os << client.accountNumber << '\t' <<
client.name << '\t' << client.balance << '\n';
	return os;
}



#include <iostream>
using namespace std;
#include <fstream>
//#include "clientData.h"
int main()
{
	try {
		ofstream createFie("credit.dat");
		if (!createFie)
			throw "File could not be
created.\n";
		ClientData client;
		for (int i = 0; i<100; i++)
			createFie.write((char*)&client,
sizeof(ClientData));
		createFie.close();
	}
	catch (char* msg)
	{
		cout << msg;
	}
}
```

```
#include <iostream>
using namespace std;
#include <fstream>
#include "clientData.h"

enum Choices {
	PRINT = 1,
	UPDATE, NEW,
	DELETE,
	END
};
int getAccount(const char * const prompt)
{
 int accountNumber;
 do {
	cout << prompt << " (1 - 100): ";
	cin >> accountNumber;
 } while (accountNumber < 1 || accountNumber > 100);
return accountNumber;
}
int enterChoice()
{
	cout << "\nEnter your choice" << endl
		<< "1 - store a formatted text file of
accounts\n"
		<< " called \"print.txt\" for printing"
<< endl
		<< "2 - update an account" << endl
		<< "3 - add a new account" << endl
		<< "4 - delete an account" << endl
		<< "5 - end program\n? ";
	int menuChoice;
	cin >> menuChoice;
	return menuChoice;
}
void printRecord(fstream &creditFl)
{
	ofstream outPrintFile("print.txt");
	if (!outPrintFile)
		throw "File could not be created.\n";
	outPrintFile << "Account" << '\t' << "Name" <<
'\t' << "Balance\n";
	creditFl.seekg(0);
	ClientData client;
	creditFl.read((char*)&client,
sizeof(ClientData));
	while (!creditFl.eof())
	{
		if (client.getAccountNumber() != 0)
			outPrintFile << client;
		creditFl.read((char*)&client,
sizeof(ClientData));
	}
}
void updateRecord(fstream &updateFl)
{
	int num = getAccount("Enter account to
update");
	updateFl.seekg((num - 1) *
sizeof(ClientData));
	ClientData client;
	updateFl.read((char*)&client,
sizeof(ClientData));
	if (client.getAccountNumber() != 0) {
		cout << client.getName() << '\t';
		cout << client.getBalance() << endl;
		cout << "\nEnter charge (+) or payment
(-): ";
		double transaction;
		cin >> transaction;
		double oldBalance = client.getBalance();
		client.setBalance(oldBalance +
transaction);
		updateFl.seekp((num - 1) *
sizeof(ClientData));
		updateFl.write((char*)&client,
sizeof(ClientData));
	}
	else cout << "Account #" << num << " not
exist.\n";
}

void deleteRecord(fstream &deleteFromFl)
{
	int num = getAccount("Enter account to
delete");
	deleteFromFl.seekg((num - 1) *
sizeof(ClientData));
	ClientData client;
	deleteFromFl.read((char*)&client,
sizeof(ClientData));
	if (client.getAccountNumber() != 0) {
		ClientData blankClient;
		deleteFromFl.seekp((num - 1) *
sizeof(ClientData));
		deleteFromFl.write((char*)&blankClient,
			sizeof(ClientData));
		cout << "Account #" << num << "
deleted.\n";
	}
	else cout << "Account #" << num << " is
empty.\n";
}
```

```
void newRecord(fstream &addToFl)
{
	int num = getAccount("Enter new account
number");
	addToFl.seekg((num - 1) * sizeof(ClientData));
	ClientData client;
	addToFl.read((char *)&client,
sizeof(ClientData));
	if (client.getAccountNumber() == 0) {
		char name[15];	double balance;
		cout << "Enter name and balance\n? ";
		cin >> name >> balance;
		ClientData client1(num, name, balance);
		addToFl.seekp((num - 1) *
sizeof(ClientData));
		addToFl.write((char*)&client1,
sizeof(ClientData));
	}
	else
		cout << "Account #" << num << " already
exists\n";
}
```

```
int main()
{
	try {
		fstream inOutCredit("credit.dat",
ios::in | ios::out);
		if (!inOutCredit) {
			throw " could not open file.\n";
		}
		int choice;
		while ((choice = enterChoice()) != END)
{
			switch (choice) {
			case PRINT: try {
				printRecord(inOutCredit);
			}
					Catch(char*
msg) {
					Cout
<< msg;
}
break;
	case UPDATE: updateRecord(inOutCredit);
				break;
	case NEW: newRecord(inOutCredit);
				break;
	case DELETE: deleteRecord(inOutCredit);
				break;
default: cout << "Incorrect choice" << endl;
}
inOutCredit.clear();//reset end-of-file indicator
		}
	}
	Catch(char* msg)
	{
		Cout << msg;
	}
	return 0;
}
```

# קריאה לא סדרתית מקובץ

בדוגמא לפנינו יש מחלקה המתייחסת לפרטי חשבון בנק של לקוח, המכיל בנאי ברירות מחדל, וכן גטרים וסטרים. כמוגן, מוגדרת פונקצית פלט os כאשר ההדפסה מוציאה את שם הלקוח ופרטי החשבון.
בוכנית הראשית אנחנו פותחים לכתיבה קובץ createFile כאשר בודקים שהקובץ אכן נפתח כמו שצריך.
הלולאה עוברת על כל הלקוחות ומדפיסה את כל הרשומות הריקות וסוגרים את הקובץ.
בהמשך העבודה עם הקובץ אנחנו לא שנה את גודל הקובץ מ100 הלקחות שהוגדרו לו מלכתחילה, וכל השינויים שנעשה יהיו רק על ה"חריצים" האלה, כאשר כל פעם שנחפש מקום לרשום עליו אנחנו נחפש את הרשומה בה מוגדר מס' החשבון ב-0, וכך נדע שניתן לכתוב עליו.
מספר החשבון בכל רשומה יקבע את מיקום הרשימה בקובץ (בין 100-1) כאשר מיקום הרשומה הראשונ הוא 0 והאחרונה הוא 99- כמו במערך – כך שלמעשה מספר חשבון פחות 1 יהווה את המיקום של החשבון בקובץ.
הENUM מגדיר לנו את האפשרויות השונות שנית לבצע בחשבון – 1. הדסה 2. עדכון/יצירת חדש 3. מחיקה 4. סיום הרף.
הפונקציה getAccount מקבל קלט של טקסט, ומדפיסה את הפעולה האמורה להתבצע בהמשך (זו פונקציה שמקבלת קלט שונהמכל פונקציה ששולחת אליה וחוסכת את הכתיבה הכפולה, או יותר, של כל כותרת פונקציה). הconst בתחילת הסוגריים חוסם את האפשרות לעשות שינוי בערכים עצמם, והקונסט השני חוסם את הפוינטר מלהמשיך ולהתקדם.
כמו כן, מגדירים במחלקה ios משתנה סטטי בשם in (המוגדר בעזרת ::) ו-Out ,

בפונקציית ההדפסה, שולחים להדפסה את הקובץ ממנו מדפיסים את פרטי החשבון, המתקבל במצב פתוח בפונקציה. פותחת קובץ פלט בשם print.txt ומדפיסה בכל שורה בנפרד, את הנתונים השונים ברווח של טאב ביניהם, כך שהנתונים יודפסו מתחת העמודות.

יש לשים לב, שהפונקציה לא סוגרת את הקובץ כמו שאנחנו רגילים מאחר והIOS אמור למחוק או בדיסטרקטור בצורה אוטומטית, אך יש להיזהר מלסמוךעל זה.
בפונקציה newReocrd קודם כל מתבצעחיפוש, של המקום אליו ניתן לכתוב, ומוודאים שוב, שהמקום בו אנחנו נמצאים שווה לס, משמע המקום הנוכחי ריק, ובמידה והמקם כבר קיים מוציאים הודעה מתאימה.
אם הכל בסדר יוצרים אובייקט חדש בעזרת הקונסטרקטור של כל המשתנים, ושומרים את כל המידע במקום הפנוי.

```
#include <iostream>
using namespace std;
void increase(void* data, int type)
{
	switch (type)
	{
	case sizeof(char) : (*((char*)data))++;
		break;
	case sizeof(short) : (*((short*)data))++;
		break;
	case sizeof(long) : (*((long*)data))++;
		break;
	}
}
int main()
{
	char a = 5;
	short b = 9;
	long c = 12;
	increase(&a, sizeof(a));
	increase(&b, sizeof(b));
	increase(&c, sizeof(c));
	cout << (int)a << ", " << b << ", " << c;
	return 0;
}
```

```
include <iostream>
using namespace std;
struct Mashke {
	int kod;
	int kamut;
	char* str;
};
int cmp1(void * elem1, void*elem2)
{
	return (*(int*)elem1 - *(int*)elem2);
}
int cmp2(void *elem1, void*elem2)
{
	Mashke *k1 = (Mashke*)elem1, *k2 = (Mashke*)
		elem2;
	return (k1->kamut - k2->kamut);
}
int cmp3(void *elem1, void* elem2)
{
	Mashke *k1 = (Mashke*)elem1, *k2 = (Mashke*)
		Elem2;
	return strcmp(k1->str, k2->str);
}

void * maxspecial(void * base, int n, int size, int
(*comparator)(void*, void*)) {
	void * makom = base;
	for (int i = 1; i <n; i++)
		if (comparator((char*)base +
(i*size), makom)> 0)
			makom = (char*)base +
(i*size);
	return makom;
}

int main()
{
	int v[] = { 1,3,7,2,4,0 };
	Mashke k[] = { { 1,20,"vodka" },
	{ 2,8,"water" },
	{ 3,100,"bira" },{ 4,20,"wine" } };
	int * gadolint =
	(int*)maxspecial(v, 6, sizeof(int), cmp1);
	cout << *gadolint << " hu hamispar hagadol"
<< endl;
	Mashke* bigMashke;
	bigMashke = (Mashke*)
		maxspecial(k, 4, sizeof(Mashke),
cmp2);
	cout << bigMashke->str << " hu haMashke
hameirabi
		(kamut)"<<endl;
		bigMashke = (Mashke*)
		maxspecial(k, 4, sizeof(Mashke),
cmp3);
	cout << bigMashke->str << " hu haMashke baal
hashem
		hachi gadol"<<endl;
		return 0;
}
```

# Void pointer

```
void * funcName(void * name, type(*comparator)(void*, void*))
```

**מצביע לפונקציה** – למעשה, כאשר אנחנו שולחים ארגומנטים לפונקציה, אנחנו משתמשים בסוג של מצביע, הלוקח את הארגומנטים ומעביר לפונקציה, מבצע את הפעולות ומחזיר את הדרוש.
האפשרות של מצביע לפונקציה, נותנת לנו כמה אפשרויות רחבות יותר לשליחה – מגדירים את הפונקציה כ * void ואז ניתן לשלוח אליו גם טיפוסים שונים עם המרה. למשל (* int) ואז כל ערך שמוגדר שנשלח לפונקציה כ*void צריך להיות מאותו סוג, וכך הפונקציה מבצעת את ההמרה.
ניתן לשלוח בתוכו גם מצביע לפונקציה/למבדה לפי הצורך, רק יש לשים לב להגדיר את סוג הפונקציה הנשלחת ולשלוח איתו את הערכים המתאימים.

בדוגמא לפנינו אנחנו מקבלים מצביע מכל סוג שהוא, ומשנים אותו בו. אך מידע זה אינו מספיק בשביל לגשת ולשנות את המידע של הטיפוס. וכך ניתן להגדיר לו כל פעולה לל סוג משתנה בהתאמה לשם הפויינטר עם כל הסוגריים המקיפות אותו שיעבור את כל המסכים עד למשתנה המקורי.
בהמשך ניתן לראות של void pointerניתן לשלוח כל משתנה מכל וג שהוא. למה עושים בהדפסה המרה לint? מאחר ואם לא תעשה המרה, יודפס הערך הascii של הספרה ולא המספר בעצמו.
בדוגמא השניה, יש מצביע לפונקציה, המקבל שני void pointer ובתוכו ממיר אותו למשתנה הרצוי ומבצעת עליו פעולות.
בפונקציה maxspecial שולחים לפונקציה המשווה מצביעים מסוג void כאשר אי אפשר לעשות עליהם שינוי אריתמטי.