

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

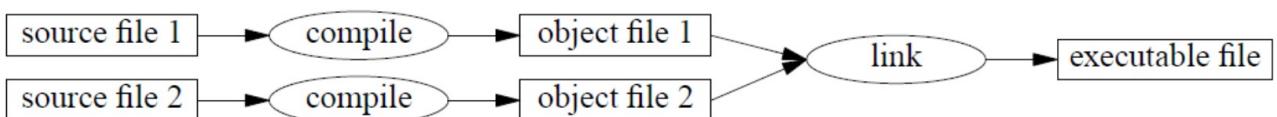
### נושא 1 - בניית תוכנית

נושאי השיעור:

- הידור וקישור
- קישור וטעינה
- ספריות סטטיות ודינמיות
- הצהרה והגדירה
- קבועי כוורת
- make

#### הידור וקישור compile and link

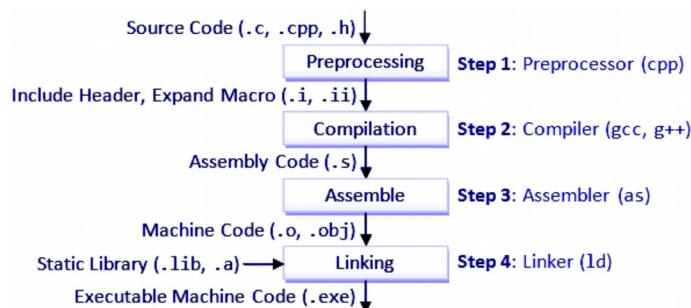
נוח לחלק תוכנית גדולה לקבצים קטנים, לעורר ולקמפל (להדר) כל קובץ בנפרד. כך, אם משנים את אחד הקבצים לא צריך לקמפל מחדש את כל התוכנית מחדש. הקבצים המכילים את התוכנית בשפת תוכנות נקראים קבועי מקור. הקומפילר יוצר עבור כל קובץ מקור, קובץ אובייקט. קבועי האובייקט הם קבועים בינארים והם מכילים את התוכנית בשפת מוכנה. לאחר יצירת קבועי האובייקט, יש צורך לחבר ביניהם. הלינקר (מקשר) לחבר קבועי אובייקט ויוצר קובץ ריצה. את קובץ הריצה ניתן לטעון לזרקן ולהריץ באמצעות מערכת הפעלה.



#### The gcc compiler driver

הפקודה gcc או g++ מפעילה את כל השלבים הדרושים לייצרת קובץ הריצה:  
 1. preprocessor: מחליף את הסמלים שהוגדרו על ידי #define, ומכניס קבועים שהוגדרו על ידי #include.

2. קומפילר (מהדר): מתרגם את התוכנית משפת C/C++ לשפת אסטטלי.
3. אסטטלאר: מתרגם את התוכנית משפת אסטטלי לקובץ אובייקט בשפת מוכנה.
4. לינקר (מקשר): לחבר את קבועי האובייקט ואת הספריות הדרישות ויוצר קובץ ריצה.



הערה: ניתן להשתמש בפקודה: gcc -fno-save-temps main.c sum.c וכך לשמור גם את כל קבועי הבינים.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

לדוגמה: בהינתן הקבצים הבאים:

<b>main.c</b>	<b>sum.c</b>
<pre>int sum(int *a, int n); int array[2] = {1, 2}; int main() {     int val = sum(array, 2);     return val; }</pre>	<pre>int sum(int *a, int n) {     int i, s = 0;     for (i = 0; i &lt; n; i++)         s += a[i];     return s; }</pre>

כרגע יראה הקובץ `i.main`, לאחר מעבר ה preprocessor על הקובץ (כמוכן שיוצר קובץ זהה עבור כל קובץ מקור):

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"

int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

לאחר מעבר הקומפילר על הקובץ `i.main` נקבל את הקובץ `main.s` הכתוב באסמבלי:

```
1 | .file   "main.c"
2 | .text
3 | .globl _array
4 | .data
5 | .align 4
6 | _array:
7 |     .long 1
8 |     .long 2
9 |     .def    __main;   .scl    2;   .type   32
10|     .text
11|     .globl _main
12|     .def    _main;   .scl    2;   .type   32; .e
13|     _main:
14|     LFB0:
15|         .cfi_startproc
16|         pushl %ebp
17|         .cfi_def_cfa_offset 8
18|         .cfi_offset 5, -8
19|         movl %esp, %ebp
20|         .cfi_def_cfa_register 5
21|         andl $-16, %esp
22|         subl $32, %esp
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### מה עושה המקשר ? linker ?

כדי לחבר את קבועי האובייקט לתוכנית ריצה, המקשר צריך לעשות שני דברים:

1. קישור בין סמלים symbol resolution

כל אחד קבועי האובייקט יכול להשתמש בסמלים ולהגדיר סמלים - שמות של משתנים או של פונקציות. ישנו קבועי האובייקט המשמשים בסמלים שהוגדרו בקבוצי אובייקט אחרים. המקשר יוצר את הקישור בין השימוש בסמל להגדרת הסמל על ידי עリכת הכתובות.

2. הזזה Relocation

הקומpileר והאSEMBLER יוצרים קבועי האובייקט כך שהכתובות של כל אחד מהם מתחילה מ-0. המקשר מזיז את הכתובות כך שלא יחפפו וקובע לכל סמל את כתובת הריצה. מעודן את ההתיחסות לסמל לפי כתובת הריצה שלו.

### סוגי סמלים

סמלים הם שמות של משתנים או פונקציות, ישנו מספר סוגי סמלים:

1. סמלים גלובליים: סמלים שהוגדרו בקובץ האובייקט ואפשר לגשת אליהם מקבוצי אובייקט אחרים.

לדוגמה: משתנים לא סטטיים שהוגדרו מחוץ לפונקציות וכן פונקציות שלא הוגדרו סטטיות.

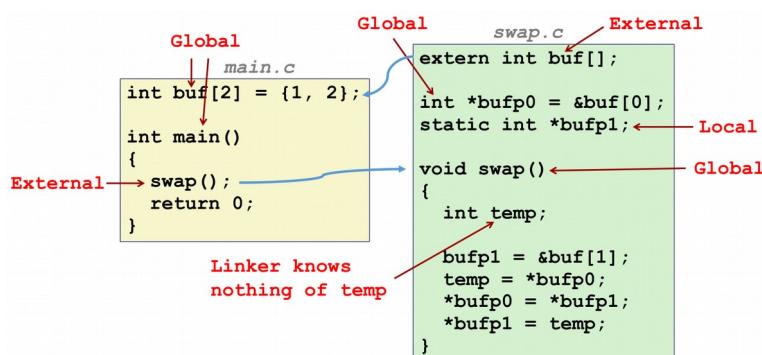
```
void swap() /* { ... } */
```

2. סמלים חיצוניים: סמלים שקובץ אובייקט משתמש בהם והם הוגדרו בקובץ אובייקט אחר.

3. סמלים מקומיים: סמלים שהוגדרו בקובץ האובייקט ואי אפשר לגשת אליהם מקבוצי אובייקט אחרים.

לדוגמה: משתנים סטטיים שהוגדרו מחוץ לפונקציות וכן פונקציות שהוגדרו סטטיות.

משתנים לא סטטיים שמוגדרים בתוך פונקציה מטופלים על ידי הקומPILEר ואינם נכללים ברשימת הסמלים של המקשר.

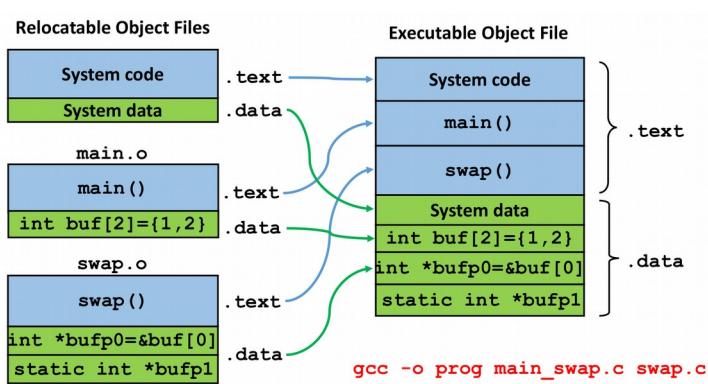


. מילה שמורה שאומרת לקומPILEר כי הסמל מוגדר בקובץ מקור אחר.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט



### הזות קוד ונתונים

בשני קבצי האובייקט שבתמונה יש חלק של קוד ונתונים. החלק של הטקסט מכיל את קוד הריצה שלהם. והחלק של ה נתונים מכיל את המשתנים הגלובליים שלהם. כאשר הקומpileר מcompiles את הקבצים הללו הוא יוצר קובץ שנקרא system code הLINKER לוקח את כל קבצי הקוד וקבצי הדאטא ומסדר אותם אחד ליד השני, וזה יוצר את קובץ ההרצה.

### סוגים של קבצי אובייקט

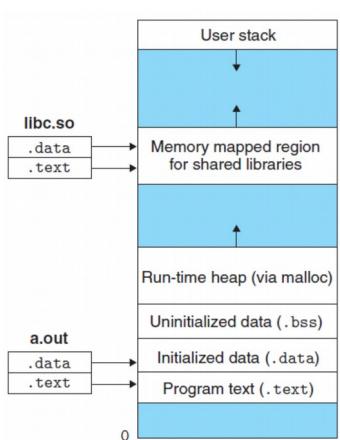
ישנם שלושה סוגים של קבצי אובייקט:

1. relocatable object file (.o file)  
מכיל קוד ונתונים באופן שמאפשר חיבור הקובץ עם קבצים אחרים.
2. Executable object file (a.out file)  
מכיל קוד ונתונים באופן שמאפשר טיענת הקובץ לזיכרון והרצה.
3. Shared object file (.so file)  
מכיל קוד ונתונים באופן שמאפשר חיבור דינامي של הקובץ לתוכנית הנטענת לזכרון או לתוכנית שכבר רצתה.

Executable and linkable format (ELF)  
פורמט בלינוקס לשימוש הסוגים של קבצי האובייקט.

### טעינה

במערכות הפעלה לינוקס, execve היא פונקציה שטוענת קובץ ריצה לזכרון. הפונקציה מעתיקה את הקוד ואת הנתונים לזכרון המחשב ו קופצת לביצוע את הפקודה הראשונה של התוכנית.  
 אז איך זה בעצם עובד ? ה shell מבקש מהמערכת להריץ את התוכנית כאשר המשתמש כותב בטרמינל:  
 ./a.out



איך מערכת הפעלה מריצה קובץ ? קובץ זה משוח שוכן על הדיסק, מערכת הפעלה תקח את הקובץ הזה מהdisk ותקרא אותו ותטען אותו לזכרון ה RAM. כאשר מערכת הפעלה טען את הקובץ היא תשים את חלקו התוכנית השוניים בחלקים שונים בזיכרון כמפורט בתרשימים. באזורי ה data מכיל נתונים שיש להם ערך תחומי ו bss מכיל נתונים שאין להם ערך תחומי. באזורי ה text יהיה את קוד ההרצה של התוכנית.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

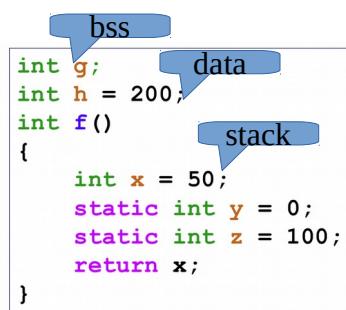
אם המערכת משתמשת בזיכרון דינמי זה יהיה במקום מיועד לו בעירמה.

so.lib.so ספריות דינמיות- מקום מיוחד בזיכרון.

בחלק העליון יש לנו את המחסנית. היא משתמש כדי לשמר את המשתנים המקומיים (המשתנים המוגדרים בתוך הפונקציות).

התמונה נראית מפת הזיכרון של התוכנית. כל תוכנית שרצה במחשב מקבל את כל מרחב הזיכרון הזה. כל תוכנית מקבלת את האשלה שיש לה את כל הזיכרון במחשב. וזה נקרא virtual memory memory. ולמעשה עברו כל תוכנית שרצה היא מיפה רק חלק קטן מהזיכרון הפיזי עבור החלק שכרגע התוכנית משתמשת בה.

## סמלים גלובליים וлокליים



משתנים סטטיים בתחום פונקציה הם משתנים המוגדרים בתחום פונקציה אבל הם נשמרים מקריאה לקריאה. אך אי אפשר לשמור משתנים אלו במחסנית ולכן יייחווים באזור ה data או bss.

## ספריות

פונקציות קלט/פלט, ניהול זיכרון, טיפול במחוזות, ופעולות מתמטיות הן פונקציות שכיחות. אך הן נמצאות בספריה ואין צורך שכל תוכנית ימציא אותן מחדש. אך לבנות את הספריה?

1. קובץ אחד גדול שמכיל את כל הפונקציות ויחבר לכל קובץ ריצה.
  - בזבוז זיכרון בדיסק ובזמן ריצה.
2. כל פונקציה בקובץ נפרד.
  - לא נוח לתוכנית.
3. חיבור קבצי אובייקט לקובץ אחד שיכיל אינדקס לאותם קבצי אובייקט, המקשר יחפש באינדקס ויחבר רק את קבצי האובייקט הדרושים.

## ספריות ידועות

הפקודה: ar -t libc.a מחרירה את כל קבצי האובייקט ב libc

```
libc.a (the C standard library)
  • 4.6 MB archive of 1496 object files.
  • I/O, memory allocation, signal handling, string handling, data and
    time, random numbers, integer math

libm.a (the C math library)
  • 2 MB archive of 444 object files.
  • floating point math (sin, cos, tan, log, exp, sqrt, ...)

% ar -t libc.a
fork.o
printf.o
fputc.o
fscanf.o
fseek.o
fstab.o
...
% ar -t libm.a
e_acos.o
e_asin.o
...
shaynaor@shayn:~$ locate libc.a
/usr/lib/x86_64-linux-gnu/libc.a
/usr/share/doc/libc/libc/README.libc
shaynaor@shayn:~$ ar -t /usr/lib/x86_64-linux-gnu/libc.a
lnt-first.o
lnt-second.o
lntd64.o
version.o
check_fds.o
lbc.o
elf-tint.o
dso_handle.o
errno.o
errno-loc.o
iconv_open.o
iconv.o
iconv_close.o
iconv_open.o
iconv.o
iconv_close.o
iconv_db.o
iconv_conf.o
iconv_builtin.o
```

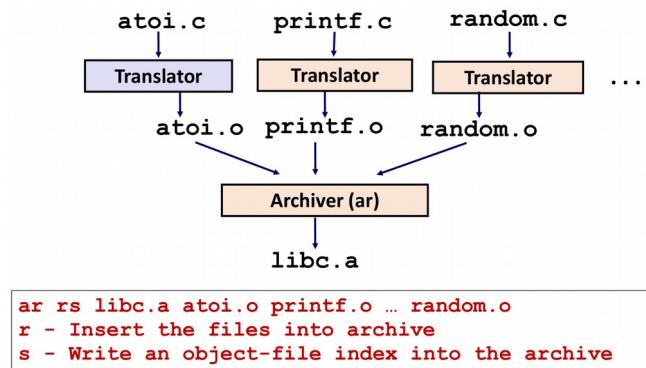
## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

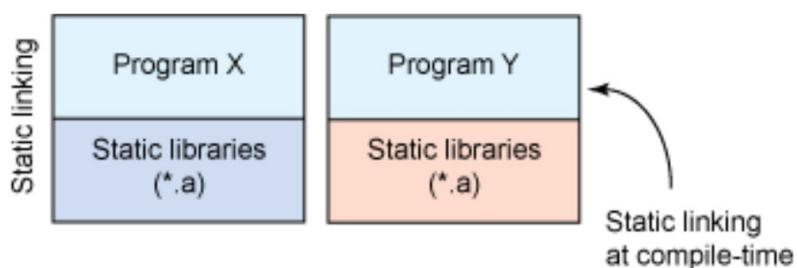
### יצירת ספרייה סטטית

ספריה היא אוסף של קבצי אובייקט שנחנו ממחברים אותם לטור קובץ אחד ומוסיפים אינדקס. קובץ שנוצר יש סיימת .a. כתוכנית רוצה להשתמש בו היא אמורה למקשר את שם הספריה והוא יודע לקחת רק את האובייקטים המופיעים שנחנו משתמשים בהם.



### ספריות סטטיות - חסרונות

- כל קובץ ריצה מכיל את פונקציות הספריה.
  - מגדיל את הקבצים בדיסק.
  - מגדיל את התוכניות הרצות בזיכרון.
- תיקון באג בספריה דורש תיקון של כל קבצי הריצה.



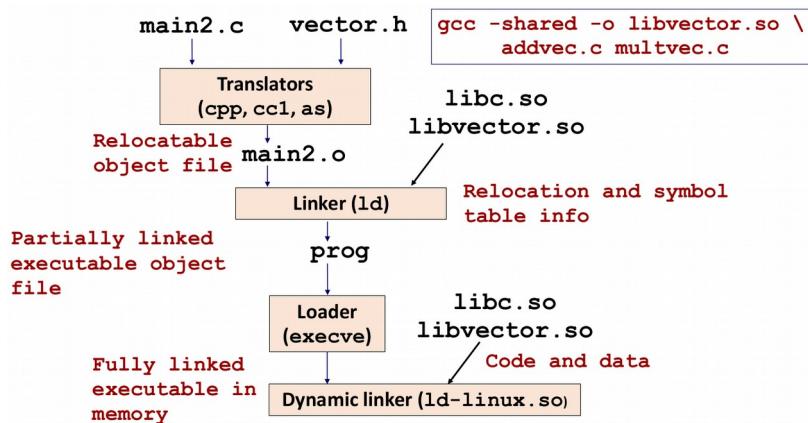
### ספריות דינמיות

- ניתן להשתמש בספריה דינמית על מנת לפתור בעיות אלה.
- נקראות גם ספריות משותפות Shared libraries.
    - משומם שקיים רק עותק אחד בדיסק שמשותף לכל התוכניות.
    - משומם שזמן ריצה רק עותק אחד נתען לזיכרון.
  - הספריות מוקשורות לקובץ הריצה בצורה דינמית (DLL, .so.).
    - או בזמן טיענת קובץ הריצה לזיכרון.
    - או לאחר שהתוכנית התחילה לזרז.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותוכנים באינטרנט

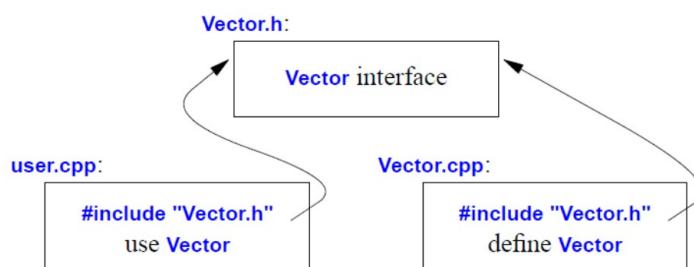


## הצירה והגדרה

- הצירה: מתארת לkompileר את התוכנות של משתנה או פונקציה המוגדרות במקום אחר.
  - אפשר להציג על אותו משתנה או פונקציה כמה פעמים.
- הגדרה: מזינה מקום עבור משתנה או פונקציה.
- יכולה להיות רק הגדרה אחת למשנה או לפונקציה.

## קבצי כותרת

- כל קובץ מקומפל בנפרד.
- כדי לקמפל קובץ, הקומpileר צריך הצירה שתתאים את המשתנים והפונקציות המוגדרות בקבצים אחרים וכן תיאור המחלקות שהקובץ משתמש בהם.
- בתוכנית שמורכבת מכמה קבצים, יש צורך בקובץ מרכזי שיכיל את ההצהרות עבור כל הקבצים.
- קובץ כותרת מכיל את ההצהרות, וכל הקבצים שכוללים את קובץ הכותרת (`#include`), משתמשים באותו הצירה.
- אם יש צורך לשנות הצירה, מספיק לשנות בקובץ הכותרת.
- גם הקובץ שגדיר את הפונקציות צריך לכלול את הקובץ (`#include`), כדי לוודא שיש התאמה בין ההצירה להגדרה.
- קובץ כותרת אינו יכול להכיל הגדרה של משתנה או פונקציה אם היא מוגדרת במקום נוסף משום שכך היא תוגדר כמה פעמים.
- קובץ כותרת יכול להכיל הצירה של משתנה, פונקציה או מחלוקת, זה דורש עבור הקומpileר בכל קובץ שימוש באובייקט.

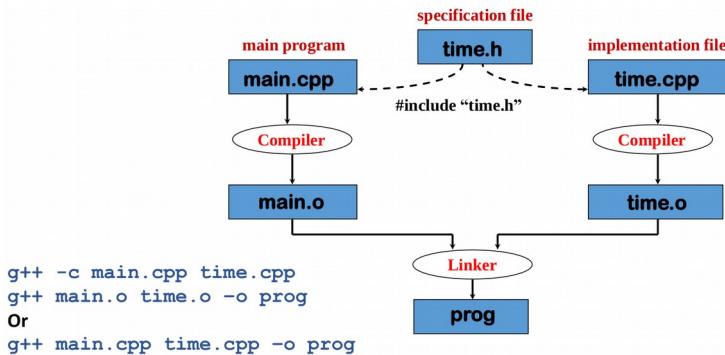


## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

## הידור וקישור עם קבץ כותרת



## הכלת קבצי הכותרת ומניעת הכלות כפולות

ניתן לכלול קובץ כותרת בשני אופנים:

```
#include <iostream>      // standard-library headers
// .h suffix not needed
#include "my_file.h"     // header supplied by program
```

קבצי כותרת יכולים לפעמים קבצי כותרת אחרים, זה יכול לגרום להגדרות כפולות.

כדי להגן מפני הגדרות כפולות:

```
#ifndef SALESITEM_H
#define SALESITEM_H
// Definition of Sales_item class and related functions
#endif
```

כאשר ה pre-processor עובר על הקובץ הוא מגיע לחילוק של ה if not define אם הוא מגיע לקטע זה בפעם הראשונה זה אומר ש `_SALESITEM` אכן לא מוגדר ולכן הוא מסHIR לשורה הבאה ומגדיר אותו ב define ומכו尼斯 את הקוד עד ה `.endif`. אחרת, `_SALESITEM` כבר מוגדר ולכן הוא מدلג על הקוד עד ה `.endif` ואז הוא לא מגדיר את אותו הקובץ פעמיים.  
 דרך נוספת היא:

```
#pragma once
```

## בנייה תוכנית באמצעות make

- make היא תוכנית לבניית תוכנה שモרכבת מכמה קבצים.
- Make מקומפלט רק את הקבצים שהשתנו מאז הקומpileציה הקודמת ואת הקבצים שתלויים בקבצים שהשתנו.
- כדי להשתמש ב make יש צורך להזכיר קובץ בשם Makefile, הקובץ מתאר את הפקודות שיש להפעיל כדי לעדכן קבצים ואת התלותות בין הקבצים.
- התוכנית make קוראת את הקובץ Makefile ומבצעת את הפקודות הדרשיות.
- אם קובץ תלוי בקובץ אחר וזמן העדכן של הקובץ האחר יותר מאוחר, התוכנית make תבצע את הפקודה שתעדכן את הקובץ.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### כללי הקובץ Makefile

- הקובץ Makefile מכיל כללים לפי הפורמט הבא:

```
target: prerequisites
[TAB]   command
```

לדוגמה:

```
myprog: myprog.cpp myprog.h
[TAB] g++ -o myprog myprog.cpp
clean:
[TAB] rm myprog
```

- תעדן את קובץ ה target **שתי** בקבצי ה **prerequisite**, אם אחד מקבציו ה prerequisite עודכן מאז שהקובץ target עודכן או אם הקובץ target לא קיים.
- הקובץ Makefile יכול להכיל כמה כללים הפקודה make תפעיל את הכלל הראשון.
- אם הכללים האחרים תלויים(target) ב target של הכללים אחרים, הם יופעלו קודם לכן.
- אם יש target שה הכללים המופעלים לא תלויים בו, הוא לא יופעל.
- כדי להפעילו צריך לתת אותו כפרמטר לפקודה make, כמו למשל `make clean`.

### דוגמה לקובץ Makefile פשוט

```
sum: main.o sum.o
      g++ -o sum main.o sum.o
main.o: main.cpp sum.h
      g++ -c main.cpp
sum.o: sum.cpp sum.h
      g++ -c sum.cpp
clean:
      rm sum main.o sum.o
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### משתנים של הקובץ Makefile

- בדוגמה הקודמת, היה צריך לפרט את קבצי האובייקט עבור כל כלל.
- ניתן לפשט את הקובץ על ידי הגדרת משתנים: `OBJECTS = main.o sum.o`
- לאחר הגדרת המשתנה, נוכל להתייחס לקבצי האובייקט ע"י `$(OBJECTS)`

```
OBJFILES = main.o sum.o
PROGRAM = sum
CC = g++
CFLAGS = -g -std=c++11 -Wall

$(PROGRAM) : $(OBJFILES)
    $(CC) $(CFLAGS) -o $(PROGRAM) $(OBJFILES)
    .
    .
clean:
    rm $(PROGRAM) $(OBJFILES)
```

## נושא 2 – מיכלים

### נושאים

- הוספות של C++ 11 +.
- String
- מיכלים
- איטרטורים

### ייצוג נתונים

נפח הזיכרון לכל סוג משתנה:

C Data Type	Typical 32-bit	Typical 64-bit
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	4
<code>long</code>	4	8
<code>float</code>	4	4
<code>double</code>	8	8
<code>pointer</code>	4	8

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### C++ 11 + תוספות של

- `auto` אומר לkompilelr להסיק את סוג המשתנה המאוחחל:

```
auto b = true;           // bool
auto ch = 'x';          // char
auto i = 123;           // int
auto d = 1.2;           // double
auto p = &i;             // pointer to int
auto& ri = x;          // reference on x (char)
const auto& cri = x;   // constant reference on x
auto z = sqrt(y);      // whatever sqrt(y) returns
auto item = val1 + val2; // result of val1 + val2
```

- `decltype()` אומר לkompilelr להסיק את סוג המשתנה מתוך ביטוי שאיןו משמש לאתחול. כלומר, דומה מאד ל`auto`, רק שהוא אומר לkompilelr להבין את סוג המשתנה מתוך ביטוי שנמצא במקום אחר.

```
decltype(f()) sum = x; // whatever f() returns
const int ci = 0;
decltype(ci) x = 0;    // const int
struct A { int i; double d; };
A* ap = new A();
decltype(ap->d) x;   // double
vector<int> ivec;
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // vector<int>::size_type
```

### List initialization •

```
:vector אתחול
vector<int> v{1,2,3,4,5,6,7,8,9};
vector<string> articles = {"a", "an", "the"};
vector<int> v1(10); // v1 has ten elements with value 0
vector<int> v2{10}; // v2 has one element with value 10
vector<int> v3(10, 1); // v3 has ten elements with value 1
vector<int> v4{10, 1}; // v4 has two elements, 10 and 1
ארבע אפשרויות לתחול משתנה:
int units_sold = 0;
int units_sold(0);
int units_sold{0};
int units_sold = {0};

int i1 = 7.2; // i1 becomes 7
int i2{7.2}; // error : narrowing conversion
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

range for •

תחביר:

```
for (declaration : expression)
    statement
expression represents a sequence
declaration variable to access the sequence
```

דוגמה:

```
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto i : v)           // i is a copy
    cout << i << " ";
cout << endl;
for (auto &i : v)          // i is a reference
    i *= i;                 // square the element
```

C++11.cpp

נשים לב להבדלים: ב for הראשון זה הוא העתק של האובייקט, ובשני הוא האובייקט עצמו.

Enum •

```
enum class Color { red, blue , green };
enum class Traffic_light { green, yellow, red };
Color col = Color::red;
Traffic_light light = Traffic_light::red;
Color x = red;                      // error, which red?
Color y = Traffic_light::red;        // error, not a Color
Color z = Color::red;                // OK
int i = Color::red;                 // error, not an int
Color c = 2;                        // error, 2 is not a Color
// enum with no class are not scoped within their enum
enum color { red, green, blue };     // no class
enum stoplight {red, yellow, green}; // error, redefines
int col = green; // convert to their integer value      enum.cpp
```

לפני השינויים שהחלו ב C++ 11 לא היה ניתן לחתה לשני enumים שונים אותם מתחננים. אך ב C++ 11 הוסיפו את האפשרות ל enum class, וכך ניתן לחתה לשני enumים שונים שמות זרים כי הגישה לכל אחד מהם היא שונה.

## סיכום קורס תכנות מתקדם

כתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### הספריה הסטנדרטית

לפני C-20 שנה החליטו להוסיף לשפת + C הבסיסית, ספריה הבנויה מעלייה ומשתמשת בעיקר בתבניות (template). הספריה נקראת Standard Template Library – STL, והוא כוון באה חלק בלתי נפרד מהשפה.

הרכיבים העיקריים של הספריה התקנית הם: מיכלים, איטרטורים, אלגוריתמים, פונקטורים, מתאים, זרים ומחוזות.

מיכלים, איטרטורים ואלגוריתמים קשורים זה לזה באופן הבא:

- כל מיכל מגדר איטרטורים המאפשרים לעבור על כל הפריטים במיכל.
- כל אלגוריתם מקבל כקלט איטרטורים המגדירים את התחום שבו האלגוריתם צריך לעבוד.

### מיכלים

הגדרה של הספריה התקנית קובעת שהמיכלים מכילים עותקים של עצמים – ולא קישורים לעצמים (בניגוד לג'אה). המשמעות:

- אפשר להכנס למיכל רק עצם שיש לו אופרטור השמה ובנאי מעתיק.
- בכל פעם שמנגנים עצם למיכל, נבנה עצם חדש; בכל פעם שמספרים מיכל, מתפרקם כל העצמים הנמצאים בו.

יש שני סוגים עיקריים של מיכלים:

- סדרתיים – אקטור, רשיימה... – שומרים פריטים לפי סדר ההכנסה שלהם
- אסוציאטיביים – קבוצה, מפה... – שומרים פריטים לפי הסדר הטבעי שלהם (המוגדר ע"ז).

### String

string הוא מחרוזת של תוים בגודל משתנה. בשפת C היה לנו מערך של תוים בתור סטרינג. החסרונו הוא שגודל המערך קבוע ולא ניתן להגדיל/להקטין אותו וכדומה. לכן ב C + + הוסיפו לספריה הסטנדרטית את המחלקה string, וכעת ניתן להגדיל/להקטין את האובייקט string בנוסף, לשרשר ופעולות נוספות.

```
הגדרה ואתחול של :string
string s1; // default initialization, empty string
string s2 = s1; // copy of s1
string s2(s1); // copy of s1
string s3 = "abc"; // copy of the string literal
string s3("abc"); // copy of the string literal
string s4(10, 'c'); // cccccccccc
:string פעולות על
s.size() // number of characters in s
s[n] // reference to char at position n
s1 + s2 // concatenation of s1 and s2
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### size\_type

בتوز המיכלים הגדרו משתנים, המתכנתים שכתבו את אותם מיכלים הגדרו את המשתנים הללו מטיפוסים מסוימים. הדרך הנכונה להגדיר משתנה שיחזק את אורך המיכל היא:

```
string::size_type len = line.size();  
// size() returns a string::size_type value  
  
The string class defines size_type so we can use the  
library in machine independent manner  
  
We use the scope operator to say that the name  
size_type is defined in the string class  
  
It is an unsigned type big enough to hold the size of  
any string
```

כאשר אנו משתמשים במיכלים אלו, וננו רצים לבצע מעבר על המיכל שהגדכנו, נכון יהיה להגדיר את המשתנה שעובר על המיכל מאותו טיפוס שבו המתכנת שהגדיר את המיכל הגדר. لكن נכון לעשות זאת כך: מעבר על תוכן מחוץ עם אינדקס:

```
string s("some string");  
// The subscript operator (the [ ] operator)  
// takes a string::size_type  
for (string::size_type index = 0;  
     index != s.size() && !isspace(s[index]);  
     ++index)  
    s[index] = toupper(s[index]);  
  
The output of this program is:  
SOME string  
  
מעבר על תוכן מחוץ עם range for  
string s("Hello World!!!!");  
for (auto &c : s)      // note: c is a reference  
    c = toupper(c);   // so the assignment changes the char  
cout << s << endl;  
  
The output of this program is:  
HELLO WORLD!!!
```

קליטת מחוץ מהמשתמש:

קראת מספר לא ידוע של מחוץ:

```
string word;  
// end-of-file or invalid input will put cin in error state  
// error state is converted to boolean false  
while (cin >> word)  
    cout << word << endl;                                         enum.cpp  
  
קריאת שורה שלמה:  
string line;  
// read up to and including the first newline  
// store what it read not including the newline  
while (getline(cin, line))  
    cout << line << endl;
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### Vector

וקטור הוא אוסף של אובייקטים מסוימים בגודל משתנה. משמש כמו מערכים, וקטור אחסון את אבריון באופן רציף, מה שאומר שניתן לגשת אל האלמנטים שלו גם באמצעות אריתמטיקת מוצבים, וביעילותות באותה מידה כמו במערכות. אך בניגוד למערכים, גודלם יכול להשתנות באופן דינמי, כאשר האחסון שלהם מטופל אוטומטית על ידי המיכל.

הגדרה ואתחלול של `vector` :

```
A vector is a class template
We have to specify which objects the vector will hold
vector<int> ivec;           // initially empty
vector<Sales_item> Sales_vec;
vector<vector<string>> file; // vector of vectors
                           :vector
v.size()      // number of elements in v
v[n]          // reference to element at position n
v.push_back(t) // add element to end of v
```

הוספה אלמנטים למיכל `vector` :

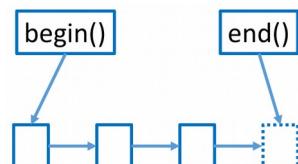
```
vector<int> ivec;           // empty vector
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // adds element with value ix
```

טוויות נפוצות בשימוש ב-`vector`:

```
vector<int> ivec;
cout << ivec[0];        // error: no elements
vector<int> ivec2(10);
cout << ivec2[10];       // error: elements 0 to 9
vector::size_type      // error
vector<int>::size_type // ok
```

### איטרטורים

איטרטור הוא אובייקט שמצויב על איבר של הסדרה. האיטרטור `begin` מצזיב על האיבר הראשון והאיטרטור `end` מצזיב על המקום לאחר האיבר האחרון.



**סיכום קורס תכניות מתקדם**

נקتب על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אריאל סגל הלוי ותכנים באינטרנט

**איטרטור צרי לספק את הפעולות הבאות:**

השוואה בין שני איטרטורים, האם מצביעים לאותו איבר:

התיחסות לערך של האיבר שהאיטרטור מצביע עליו:  
`val = *iter , *iter = val`  
`(*iter).member → iter->member`

## **קידום האיטרטור כך שיצביע לאיבר הבא:**

משמעות הדבר היא שבמידה ונתכנת מיכל כלשהו ונרצה להוסיף למיכל שלנו איטרטור שיבצע מעבר סידרתי על המיכל שלנו ניהו ח'יבים לספק את הפעולות הנ'ל.

לכל המיכלים המוגדרים בספריה הסטנדרטית יש איטרטורים, המוגדרים בספריה:

```
vector<int>::iterator it; // it can read and write
string::iterator it2;     // it2 can read and write
vector<int>::const_iterator it3; // can read but not write
string::const_iterator it4;      // can read but not write
// The type returned by begin and end depends on whether the
// object is const
auto it1 = v.begin();
// To ask for the const_iterator type:
auto it3 = v.cbegin(); // it3 has type const iterator
```

**דוגמיה לחיפוש ביןארו עם איטרטורים (אריתמטיקה של איטרטורים):**

```
vector<string> text = {"abc", "def", ...};  
auto beg = text.begin(), end = text.end();  
auto mid = text.begin() + (end - beg) / 2;  
while (mid != end && *mid != sought) {  
    if (sought < *mid)  
        end = mid;  
    else  
        beg = mid + 1;  
    mid = beg + (end - beg) / 2;  
} // mid will be equal to end or *mid will equal sought
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### סוגי המיכלים

- string
  - מיכל המכיל תוים בלבד.
  - גישה אקראית מהירה, הוספה ומחיקה מהירה בסוף המחרוזת.
- Vactor
  - מערך בגודל משתנה.
  - גישה אקראית מהירה, הוספה ומחיקה מהירה בסוף הווקטור.
- Deque
  - תור עם שני קציאות.
  - גישה אקראית מהירה, הוספה ומחיקה מהירה בתחילת ובסוף התור.
- List
  - רשימה מקושרת כפולה.
  - גישה סדרתית בלבד, הוספה ומחיקה מהירה בכל מקום בראשימה.
- Map
  - אוסף של זוגות מפתח-ערך.
  - חיפוש מהיר של ערך לפי מפתח.
- Set
  - אוסף של מפתחות.
  - חיפוש מהיר האם המפתח נמצא באוסף.

### Push\_front

למיכל list ולמיכל deque אפשר להוסיף בתחילת איבר גם לתחילת הרשימה עם push\_front:

```
list<int> ilist;
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

### insert

הכנסת איבר או איברים בכל מקום במיכל. הפעמטר הראשון הוא איטרטור שמצויב על המקום שלפניו יוכנסו האיברים:

```
list<string> slist;
// equivalent to calling slist.push_front("Hello!")
slist.insert(slist.begin(), "Hello ");
slist.insert(slist.end(), "world!"); // {"Hello", "world!"}
// insert the last two elements of v at the beginning of slist
slist.insert(slist.begin(), v.end() - 2, v.end());

vector<string> svec;
svec.insert(svec.begin(), "Hello!");
// No push_front on vector or string
// Can insert anywhere in a vector or string
// However, doing so can be an expensive operation
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### emplace

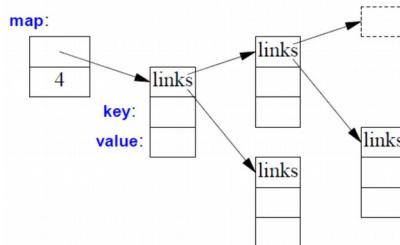
בניית אלמנט באמצעות בנאי והכנסתו למיכל.

```
// construct a Sales_data object at the end of c
// uses the three-argument Sales_data constructor
c.emplace_back("978-0590353403", 25, 15.99);
// equivalent to creating a temporary Sales_data object
// and passing it to push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
// in front of iter
c.emplace(iter, "999-999999999", 25, 15.99);
// The arguments to emplace must match a constructor
```

### map

חיפוש בראשימה הוא לא יעיל, פועל בזמן לינארי ( $N$ )ו.

מפה ממומשת באמצעות עצ חיפוש מארן, מכאן שהחיפוש במפה הוא יותר יעיל: ( $\log(N)$ )ו.



דוגמה לשימוש במפה:

```
map<string,int> phone_book {
    {"David Hume", 123456},
    {"Karl Popper", 234567},
    {"Bertrand Arthur William Russell", 345678}
};
int get_number(const string& s)
{
    return phone_book[s];
}
// When indexed by a key, a map returns the value
// If a key isn't found, it is entered into the map with a
// default value
// To avoid entering invalid numbers into our phone book,
// we could use find() instead of []:
phone_book.find(s)
```

### set

```
map<string, size_t> word_count;
set<string> exclude = {"the", "and", "or", "an", "a"};
string word;
while (cin >> word)
// count only words that are not in exclude
    if (exclude.find(word) == exclude.end())
        ++word_count[word];
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### נושא 3 – אלגוריתמים

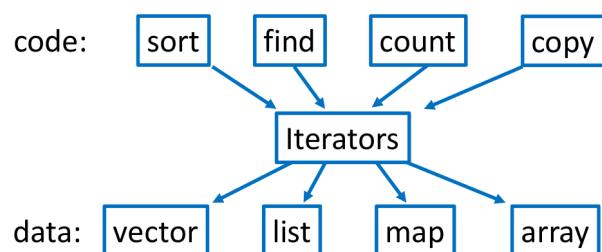
#### נושאים

- אלגוריתמים.
- פרדייקט.
- אובייקט פונקציה.
- ביטוי למבדה.

#### אלגוריתמים

בספריה התקנית של + C יש 105 אלגוריתמים, נכון לשנת 2017. אנחנו צריכים להכיר את כולם. למה? – כי האלגוריתמים פותרים בעיות מאד נפוצות בתיכנות. אם לא נכיר את כולם, אנחנו עלולים לנסתות למשם אותם בעצמנו תוך כדי פרויקט אחר. אז, אנחנו נראה המשם אותם בחיפזון, בלי בדיקות, בצורה לא יעילת ועם באגים. בנוסף, האלגוריתמים נכנסו לתקן של השפה, ולכן מתכוונים בכל העולם משתמשים בהם וمبינים אותם, ומצפים שגם מתכנתים חדשים יכירו אותם וישתמשו באותו שפה.

המיכלים מגדרים מספר מצומצם של פעולות על המיכל. ישנו פועלות נוספות שנרצה לעשות "חיפוש איבר, החלפת איבר, סידור מחדש של האיברים". כדי שלא נצטרך להגיד את כל הפעולות עבור כל המיכלים, הספריה הסטנדרטית (STL) הגדרה אלגוריתמים שיכולים לפעול על המיכלים השונים. המיכל מספק איטרטור, האלגוריתם קורא וכותב אל המיכל באמצעות האיטרטור.



#### Find

האלגוריתם עובר על המיכל בתחום של שני איטרטורים, מוצא ערך בטוח מסויים. מחזיר איטרטור לרכיב הראשון בטוחה [first, last) השווה לערך שchipשנו. אם לא נמצא אלמנט זהה, הפונקציה מחזירה את last).

```
// look through string elements in a list
string val = "a value";
auto result = find(1st.cbegin(), 1st.cend(), val);
מצביעים פועלים כמו איטרטורים, אך אפשר לחפש גם במערך:
int ia[] = {27, 210, 12, 47, 109};
int val = 83;
int* result = find(ia, &ia[5], val);
// int* result = find(begin(ia), end(ia), val);
אפשר לחפש בחלק מהמערך:
// search from ia[1] up to but not including ia[4]
auto result = find(ia + 1, ia + 4, val);
```

## סיכום קורס תכנות מתקדם

כתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

נשים לב כי ניתן להפעיל את האלגוריתמים על מערך רגיל, הספרייה הסטנדרטית מספקת למקרים אלוי באמצעות הפונקציות `begin`, `end` איטרטורים לתחילת ובסוף המערך.

### Accumulate

`accumulate` מסכם את האיברים שבמיכל. עבור על המיכל בתחום של שני איטרטורים, ומוסיף אותם לפתרון השלישי. הפעלה השלישי קובע איך פעולה חיבור תבוצע, כלומר אם הוא `int` יבוצע חיבור על אינטגרים וכדומה.

```
// sum the elements in vec starting the summation with 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
הפעלה + מוגדרת עבור string לנכון לחבר מחזוזות:
string sum = accumulate(v.cbegin(), v.cend(), string(""));
שגיאה, הפעלה + לא מוגדרת עבור *char*
// error: no + on const char*
string sum = accumulate(v.cbegin(), v.cend(), "");
```

באלגוריתמים כמו `find` ו-`accumulate` שרק קוראים את איברי המיכל, עדיף להשתמש ב- `cbegin` ו- `cend`.

### Equal

בודק האם שני מיכלים הם שווים ומהזיר אמת או שקר בהתאם. האלגוריתם מקבל שני איטרטורים עבור המיכל הראשון ואחד עבור המיכל השני (ישנה הנחה שהגודל זהה):

```
// roster2 should have at least as many elements as roster1
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
דוגמה:
list<string> roster1 = {"a", "an", "the"};
vector<const char*> roster2 = {"a", "an", "the"};
האלגוריתם מניח שהמיכל השני גדול לפחות כמו המיכל הראשון.
```

ניתן להשוות בין מיכלים שונים, ואפילו בין מיכלים שיש להם איברים שונים, במידה וניתן להשתמש באופרטור `=`.

### copy

מעתיק איברים ממכל למכל. האלגוריתם מקבל שני איטרטורים עבור המיכל הראשון ואחד עבור המיכל השני.

```
// use copy to copy one built-in array to another
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)]; // a2 has same size as a1
// ret points just past the last element copied into a2
auto ret = copy(begin(a1), end(a1), a2);
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### Sort, unique

sort ממיין את איברי המיכל לפי האופרטור < (באופן דיפולטיבי).

Unique מסדר את המיכל כך שבתחילה המיכל לא יופיע איברים כפויים. האלגוריתמים מקבלים שני איטרטורים:

```
void elimDups(vector<string> &words) {
    // sort words alphabetically so we can find the duplicates
    sort(words.begin(), words.end());
    // unique reorders the input so that each word appears once
    // in the front portion of the range
    // and returns an iterator one past the unique range
    auto end_unique = unique(words.begin(), words.end());
    // erase uses a vector operation to remove the non-unique
    words.erase(end_unique, words.end());
}
```

על מנת ש unique יעבד כמו שצריך, צריך למיין תחילה את המיכל.

Unique מחזיר איטרטור לאיבר שאחרי האיברים הייחודיים.

the quick red fox jumps over the slow red turtle  
`sort(words.begin(), words.end());`  
 fox jumps over quick red red slow the the turtle

auto end\_unique = unique(words.begin(), words.end());  
 fox jumps over quick red slow the turtle ??? ???  
 ↑  
 end\_unique  
(one past the last unique element)  
`words.erase(end_unique, words.end());`  
 fox jumps over quick red slow the turtle

### Replace, replace\_copy

replace מחליף ערך מסוים במיכל בערך אחר.

replace\_copy מחליף ערך מסוים בערך אחר ומעתיק למיכל אחר.

```
// replace any element with the value 0 with 42
replace(lst.begin(), lst.end(), 0, 42);

// leave the original sequence unchanged
// a third iterator is a destination to write the sequence
replace_copy(lst.cbegin(), lst.cend(), vec.begin(), 0, 42);
```

צריך לוודא שבמיכל האخر יש מספיק מקום.

## סיכום קורס תכנות מתקדם

כתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### Insert iterators

אם אין מספיק איברים במיכל שאלוי רוצים להעתיק או שהמיכל ריק, אפשר להשתמש ב insert iterators – איטרטורים המכניםים איברים למיכלים ומעתיקם לתוכם.  
push\_back: יוצר אובייקט שמתנהג כמו איטרטור ומשתמש ב.push\_back.  
push\_front: יוצר אובייקט שמתנהג כמו איטרטור ומשתמש ב.push\_front.  
.insert: יוצר איטרטור שמשתמש ב.insert.

נניח שהגדכנו **Insert Iterator** בשם `iter`, `t` הוא איבר שהוא רוצים להוסיף:

```
iter = t
// calls c.push_back(t), c.push_front(t)
// or c.insert(t,p) where p is the iterator given to insert

*iter, ++iter
// Each operator returns iter, they do nothing
```

### back\_inserter

מוסיף איברים למיכל ומעתיק לתוכם

```
vector<int> vec; // empty vector
auto back_it = back_inserter(vec);
// assigning through back_it adds elements to vec
*back_it = 42; // vec now has one element with value 42
```

בדוגמה הקודמת:

```
vector<int> vec; // empty vector
// use back_inserter to grow destination as needed
replace_copy(ilst.cbegin(), ilst.cend(),
            back_inserter(ivec), 0, 42);
```

### transform

הפעלת פעולה מסוימת על איברי מיכל. מימוש:

```
template<typename In, typename Out, typename Op>
Out transform(In first, In last, Out res, Op op)
{
    while (first!=last)
        *res++ = op(*first++);
    return res;
}
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותוכנים באינטרנט

### העברת פונקציה לאלגוריתמים

הרבה אלגוריתמים משווים בין איברים, ולצורך זה הם משתמשים באופרטורים `<`, `=`, `==`. לפעמים נדרש להגדיר בעצמו מה קטן ומה שווה. לאלגוריתמים יש גרסאות שמקבלות פונקציה שמחילה את האופרטורים `<`, `=`, `==`.  
 דוגמה, מיון לפי גודל מילה:

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

### אובייקט פונקציה – העממת אופרטור()

מחלוקת שמעמיסה את אופרטור הקריאה לפונקציה, מאפשרת להשתמש באובייקטים של אותה מחלוקת כאילו הם פונקציה.

דוגמה, אובייקט פונקציה שמחזיר את הערך המוחלט של מספר:

```
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
};

int i = -42;
absInt absObj;      // object has a function-call operator
int ui = absObj(i); // passes i to absObj.operator()

// Greater_than is a function object holding the value (42)
// to be compared against:
struct Greater_than {
    int val;
    Greater_than(int v) : val{v} { }
    bool operator()(const pair<string,int>& r)
        { return r.second>val; }
};
auto p = find_if(m.begin(),m.end(),Greater_than{42});
```

### העברת ביטוי למבדה לאלגוריתמים

כשמעבירים פונקציה לאלגוריתמים, צריך להגיד אותה בנפרד. מבדח זאת, הפונקציה לא יכולה לקבל יותר אחד או שני פרמטרים שהאלגוריתם מעביר לה.  
 ביטוי למבדה הוא פונקציה ללא שם שאפשר להגיד הקריאה לאלגוריתם. לביטוי למבדה יש את הצורה הבאה:

```
[capture list] (parameter list) { function body }
```

ביטוי למבדה ממומש באמצעות אובייקט פונקציה.

דוגמה, הדפסת איברי מיל עם `for_each` באמצעות ביטוי למבדה:

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

```
// print words, each one followed by a space
for_each(words.begin(), words.end(),
    [] (const string &s) {cout << s << " ";});
cout << endl;
```

ביטוי למבداה יכול להתמשב במשתנים חיצוניים בגין הפקציה שהיא מוגדר אם ורק אם הם מועברים ב list .capture

דוגמה, מציאת האיבר הראשון בגודל מערך מסוים עם :find\_if

```
// get an iterator to the first element
// whose size() is >= sz
int sz = 10;
auto wc = find_if(words.begin(), words.end(),
    [sz] (const string &a){ return a.size() >= sz; });
```

דוגמה, ספירת האיברים שקטנים מערך מסוים עם :count\_if

```
// count number of values less than x
int x = 50;
int c = count_if(vec.begin(), vec.end(),
    [x] (int a){ return a < x; });
```

Capture by reference מאפשר לביטוי למבדה לשנות ערכים הנמצאים מחוץ לביטוי.

דוגמה, חשב את סכום האיברים ושמור אותו במשתנה שהוגדר מחוץ למבדה:

```
int sum = 0;
for_each(vec.begin(), vec.end(),
    [&sum] (int x) { sum += x; });
```

דוגמה, הדפסת איברי המיל בamusot ביטוי למבדה:

```
// Print words to os separated by c:
void print_words(vector<string> &words,
    ostream &os = cout, char c = ' ')
{
    for_each(words.begin(), words.end(),
        [&os, c] (const string &s) { os << s << c; });
} // the only way to capture os is by reference
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### האפשרויות של capture list

[]	Empty capture list. The lambda may not use variables from the enclosing function. A lambda may use local variables only if it captures them.
[names]	<i>names</i> is a comma-separated list of names local to the enclosing function. By default, variables in the capture list are copied. A name preceded by & is captured by reference.
[&]	Implicit by reference capture list. Entities from the enclosing function used in the lambda body are used by reference.
[=]	Implicit by value capture list. Entities from the enclosing function used in the lambda body are copied into the lambda body.
[&, identifier_list]	<i>identifier_list</i> is a comma-separated list of zero or more variables from the enclosing function. These variables are captured by value; any implicitly captured variables are captured by reference. The names in <i>identifier_list</i> must not be preceded by an &.
[=, reference_list]	Variables included in the <i>reference_list</i> are captured by reference; any implicitly captured variables are captured by value. The names in <i>reference_list</i> may not include <code>this</code> and must be preceded by an &.

## סיכום אלגוריתם

`p=find(b,e,x)` p is the first p in [b:e) so that \*p==x  
`p=find_if(b,e,f)` p is the first p in [b:e) so that f(\*p)==true  
`n=count(b,e,x)` n is the number of elements \*q in [b:e) so that \*q==x  
`n=count_if(b,e,f)` n is the number of elements \*q in [b:e) so that f(\*q)  
`replace(b,e,v,v2)` Replace elements \*q in [b:e) so that \*q==v by v2  
`replace_if(b,e,f,v2)` Replace elements \*q in [b:e) so that f(\*q) by v2  
`p=copy(b,e,out)` Copy [b:e) to [out:p)  
`p=copy_if(b,e,out,f)` Copy elements \*q from [b:e) so that f(\*q) to [out:p)  
`p=unique(b,e)` Copy [b:e) to [b:p); discard adjacent duplicates  
`accumulate(b,e,init)` Sum elements of [b:e) to init

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### נושא 3 – קלט ופלט

#### קריאה וכתיבה של קבצים

```
// construct an input stream and open the given file
ifstream in(ifile);
// output stream that is not associated with any file
ofstream out;
out.open(ifile + ".copy"); // mode implicitly trunc
// close out stream so we can use it for another file
out.close();
out.open("precious", ofstream::app); // mode is app
out.close();
```

#### קליטה ועיבוד של מספר קבצים

```
int main(int argc, char *argv[]) {
    for (auto p = argv + 1; p != argv + argc; ++p) {
        ifstream input(*p); // create input and open the file
        if (input) { // if file open is ok,
            process(input); // "process" this file
        } else cerr << "couldn't open: " + string(*p);
    }
}
// Because input is local to the while,
// it is created and destroyed on each iteration
// When an fstream object goes out of scope,
// the file is automatically closed
```

```
argv[0] = "prog";
argv[1] = "file1";
argv[2] = "file2";
argv[3] = "file3";
argv[4] = 0;
```

#### Manipulators

```
// decimal octal or hexadecimal
cout << showbase; // hexadecimal: 0x, octal: leading 0
cout << "default: " << 20 << " " << 1024 << endl; // default: 20 1024
cout << "in octal: " << oct << 20 << " " << 1024 << endl; // in octal: 024 02000
cout << "in hex: " << hex << 20 << " " << 1024 << endl; // in hex: 0x14 0x400
cout << "in decimal: " << dec << 20 << " " << 1024 << endl; // in decimal: 20 1024
cout << noshowbase;

// Manipulators leave the format state changed
```

## סיכום קורס תכנות מתקדם

כתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו ותכנים באינטרנט

```
// precision, cout.precision reports the current precision
cout << "Precision: " << cout.precision() << ", Value: " << sqrt(2.0) << endl;
cout << setprecision(3) << "Precision: " << cout.precision() << ", Value: " << sqrt(2.0) <<
endl;
//Precision: 6, Value: 1.41421
//Precision: 3, Value: 1.41
```

```
// boolalpha
cout << true << " " << false << boolalpha << " " << true << " " << false << endl;
//1 0 true false
// setw: minimum width, left: left-justiy, right: right-justify
```

```
istringstream
```

```
struct PersonInfo {
    string name;
    vector<string> phones;
};
vector<PersonInfo> getData(istream &is) {
    string line, word;
    vector<PersonInfo> people;
    while (getline(is, line)) {
        istringstream record(line);
        PersonInfo info;
        record >> info.name;
        while (record >> word)
            info.phones.push_back(word);
        people.push_back(info);
    }
    return people;
}
```

input file

```
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

people	name phones
name phones	name phones
name phones	

```
ostringstream
```

```
ostream& process(ostream &os, vector<PersonInfo> people) {
    for (const auto &entry : people) {
        ostringstream formatted, badNums;
        for (const auto &nums : entry.phones) {
            if (!valid(nums)) {
                badNums << " " << nums;
            } else
                formatted << " " << format(nums);
        }
        if (badNums.str().empty())
            os << entry.name << " " << formatted.str() << endl;
        else
            cerr << "input error: " << entry.name
            << " invalid number(s) " << badNums.str() << endl;
    }
    return os;
}
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### istream\_iterator

```
istream_iterator<T> in(is);    in reads values of type T from input stream is.  
istream_iterator<T> end;        Off-the-end iterator for an istream_iterator that  
                                reads values of type T.  
  
vector<int> vec;  
istream_iterator<int> in_iter(cin); // read ints from cin  
istream_iterator<int> eof_iter;    // end iterator  
while (in_iter != eof_iter)       // while there's input  
    vec.push_back(*in_iter++);
```

we can rewrite this program as:

```
istream_iterator<int> in_iter(cin), eof;  
// construct vec from an iterator range  
vector<int> vec(in_iter, eof);
```

### שימוש ב Algorithms עם istream\_iterator

נניח ונרצה לכתוב תוכנית שקוראת מספרים מהמשתמש וסכוםת אותם, ניתן לעשות זאת כך:

```
istream_iterator<int> in(cin), eof;  
cout << accumulate(in, eof, 0) << endl;
```

הדפסת מספר הערכים הוזגים בקובץ "myfile"

```
istream f("myfile");  
istream_iterator<int> in(f), eof;  
cout << count_if(in, eof, [](int n) { return n % 2 == 0; });
```

### ostream\_iterator

```
ostream_iterator<T> out(os);    out writes values of type T to output stream os.  
ostream_iterator<T> out(os, d);  out writes values of type T followed by d to  
                                output stream os. d points to a null-terminated  
                                character array.
```

We can use an ostream\_iterator to write a sequence of values.

We may provide a string to print following each element.

```
ostream_iterator<int> out_iter(cout, " ");  
for (auto e : vec)  
    *out_iter++ = e;  
// out_iter = e; // equivalent
```

Or:

```
copy(vec.begin(), vec.end(), out_iter);
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

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

class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {   real = r;    imag = i; }
    friend ostream & operator << (ostream &out, const Complex &c);
    friend istream & operator >> (istream &in, Complex &c);
};

ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}

istream & operator >> (istream &in, Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imaginary Part ";
    in >> c.imag;
    return in;
}

int main()
{
    Complex c1;
    cin >> c1;
    cout << "The complex object is ";
    cout << c1;
    return 0;
}
```

Output:

```
Enter Real Part 10
Enter Imaginary Part 20
The complex object is 10+i20
```

## **סיכום קורס תכנות מתקדם**

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### **נושא 4 – בניית מחלקה**

#### **מחלקה**

- מחלקה היא הרחבה של struct בשפת C.
- המחלקה מאפשרת להגדיר משתני מחלקה (Data members).
- בנוסף, היא מאפשרת להגדיר גם פונקציות חברות (Member functions).
- מחלקה מאפשרת:
- הפעלת נתונים (Data Abstractions): התעלמות מפרטיו המימוש של העצם והתרכזות במאפיינים שלו.
- כיבוס (Encapsulation): הסתרת פרטי המימוש מהמשתמש.
- ניתן לקבוע הרשות גישה לחבריו המחלקה:
- חברי מחלקה המוגדרים private נגשים רק לפונקציות חברות במחלקה.
- חברי מחלקה המוגדרים כ public נגשים גם לשאר פונקציות התוכנית.

#### **בנייה מחלקה**

בשפת C++ יש עקרון חשוב:

- כל עצם חייב לעبور תהליך של **בניה** (construction) ברגע שהוא נוצר. הבניה מתבצעת ע"י בניאי.
- כל עצם חייב לעبور תהליך של **פרק** (destruction) ברגע שהוא מפסיק לתקיים. הפרוק מתבצע ע"י מפרק.

#### **בנייה – constructors**

בניאי הוא שיטה בשם **בניה** של המחלקה, וכן לה ערך-חזורה. כמו כל שיטה אחרת, ניתן למשהו בקובץ הכותרת או בקובץ המימוש. כמו כל פונקציה, גם בניאי אפשר להעמעיס. למשל, אפשר להגדיר כמה בניאים עם פרמטרים שונים, והקומפיאיר יקרה לבניאי הנכון לפי השימוש. איך קוראים לבניאי? תלו:

- אם זה בניאי עם פרמטרים, פשוט שמים את הפרמטרים אחרי שם העצם, למשל Point; Point(10,20);
- אם זה בניאי בלי פרמטרים, לא שמים שם דבר אחרי שם העצם, למשל Point(); (לא לשים סוגרים ריקים – זה יגרום לשגיאת קימפואל כי הקומפיאיר עלול לחשב שאתם מנסים להגדיר פונקציה).

#### **בנייה ללא פרמטרים**

יש כמה סוגי בניאים שיש להם תפקיד מיוחד – **C++**. אחד מהם הוא בניאי **ללא פרמטרים** (parameterless constructor). אם (ורק אם) לא מגדרים בניאי למחלקה – הקומפיאיר אוטומטית יוצר עבורה בניאי כזה. הוא נקרא default constructor. בהתאם לגישה של C++ "לא השתמשת – לא שילמת", בניאי ברירת-מחדר של מחלקה פשוטה לא עושה כלום – הוא לא מתחל את הדיבור. לכן, הערכים לא מוגדרים – יתכן שיהיו שם ערכים מוזרים שבמקרה היו בזיכרון באותו זמן.

## סיכום קורס תכנות מתקדם

כתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלוי ותכנים באינטרנט

### מפרק 1 – destructors

אנחנו מגיעים עכשווי לאחד ההבדלים העיקריים בין `+ C` ל JAVA. כזכור, ב- `+ C` ניהול הזיכרון הוא באחריות המתכנות. בפרט, אם אנחנו יוצרים משתנים חדשים על הערימה, אנחנו חייבים לוודא שהם משוחררים כשהאנחנו כבר לא צריכים אותם יותר. לשם כך, בכל מחלוקת שמקצת זיכרון (או מבצעת פעולות אחרות הדורשות משאבי מערכת), חייבים לשים מפרק – `destructor`.

מפרק הוא שיטה בליעור מוחזר, ששמה מתחילה בגל (טילדה) ואחריו שם המחלוקת. המתכנות אף-פעם לא צריך לקרוא למפרק באופן ידני. זהה האחוריות של הקומפיאר להכניס קריאה למפרק ברגע שהעצמם מפסיק להתקיים. מתי זה קורה?

- אם העצם נוצר על המחסנית – העצם מפסיק להתקיים כשהוא יוצא מה-`scope` (יצאים מהblkוק הנוכחי בסוגרים מסוימים המכיל את העצם).
- אם העצם נוצר על הערימה בעזרת `new` – העצם מפסיק להתקיים כמשמעותו עוזרת `.delete`.

מה קורה כשהעוצבים לשם מפרק? המשאים לא ישחררו, ותוצאה מכך תהיה דליפת זיכרון.

### האופרטורים `new`, `delete`

האופרטור `new` מבצע, כאמור מחדל, את הדברים הבאים:

- הקצת זיכרון עבור עצם חדש מהמחלוקת.
- קריאה לבניית המתאים של המחלוקת (בהתאם לפרמטרים שהועברו).

האופרטור `delete` מבצע, כאמור מחדל, את הדברים הבאים:

- קריאה למפרק של המחלוקת (יש רק אחד – אין פרמטרים).
- שיחזור הזיכרון שהוקצתה עבור העצם.

האופרטור `[ ] new` מבצע, כאמור מחדל, את הדברים הבאים:

- הקצת זיכרון עבור מערך של עצמים מהמחלוקת.
- קריאה לבניית ברירת-המחдел של כל אחד מהמעדים במערך.

האופרטור `[ ] delete` מבצע, כאמור מחדל, את הדברים הבאים:

- קריאה למפרק של כל אחד מהמעדים במערך.
- שיחזור הזיכרון שהוקצתה עבור המערך.

כשמאתחלים מערך ע"י `new [ ]`, חייבים לשחרר אותו ע"י `[ ] delete`. כאן הקומפיאר לא יכול אתכם משגיאה – אם תשתמשו ב-`delete` במקום `[ ]`, הקוד יתקפא, אבל הקומפיאר יקרא רק למפרק אחד (של האיבר הראשון במערך). תוצאה מכך תהיה לכם דליפת זיכרון, או אפילו שגיאת ריצה.

### הרכבה composition

בעצם ב- `+ C` יכולים להיות שדות בהם עצם עצמאי.

דוגמה קלאסית: מחלוקת Line שיש לה שני שדות מסוג Point גם ב JAVA זה אפשרי, אבל יש הבדל – ב- `+ C` העצים מסוג "נקודת" פשוט מונחים אחד ליד השמי בעצם מסוג "קו", בעוד שב JAVA העץ מסוג "קו" מכיל רק מצביעים לעצים מסוג "נקודת". לכן ב- `+ C` עשויים עצם מסוג "קו", עוד לפני שמאתחלים אותו, כבר יש בו "נקודת". בעוד שב JAVA הנקודות עדין לא קיימות – יש רק מצביע המאותחל ל-`Line`.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלוי ותוכנים באינטרנט

למה זה משנה ?

- בלי' מצביעים, התוכנה תופסת פחות זיכרון ודורשת פחות זמן כשניגשים למשתנים.
- כשל המבנה נמצא באותו מקום בזיכרון, ניהול הזיכרון מהיר יותר. בפרט, שימושם בזיכרון מטמון (cache), זה מאד יעיל שכן המבנה נטען בבת-אחת לאותו בלוק בזיכרון.

מה קורה כשהעצמים המוכלים יש בנאים ומפרקים משל עצמם ?

הקומפילר קורא להם לפי סדר פשוט והגיוני – כמו למשל בבנייה ופירוק של מכונית:

- בבנייה הולכים מהתן אל הגadol – קודם-כל יוצרים את כל הרכיבים (במקרה שלנו: שתי נקודות), ואז יוצרים את העצם הגדל (במקרה שלנו: קו).
- בפירוק הולכים מהגדל אל הקטן – קודם-כל מפרקים את העצם הגדל (קו) ואז מפרקים את כל הרכיבים (שתי נקודות).

### רשימת איתחול – initialization list

כשהקומפילר בונה את הרכיבים, הוא משתמש במבנה ברירת המחדל שלהם (default constructor)

- בנאי בלי פרמטרים – אם יש להם.

בנאי ברירת מחדל של מחלוקת פשוטה (בל' רכיבים) – לא עושה כלום.

בנאי ברירת מחדל של מחלוקת מורכבת – קורא לבניי ברירת המחדל של הרכיבים.

אם לרכיבים אין בנאי ברירת מחדל – אז כברית מחדל, תהיה שגיאת קומPILEZA.

מה עושים אם רוצים לקורא לבניי אחר במקום בנאי ברירת-הchandle ?

משתמשים ברשימה איתחול. יש לשים את הרשימה אחורי הכוורת של הבניי אבל לפני הסוגרים המסוללים של קוד הבניי.

שימוש ברשימה איתחול הוא מהיר יותר ובוטוח יותר מאשר מאיתחול השדות בתוך הבניי. מדוע ?

• מהיר יותר – כי הוא חוסף את האיתחול ע"י בנאי ברירת-הchandle.

• בטוח יותר – כי הוא מבטיח שבתוך הבניי, כל הרכיבים כבר בנויים עם הפרמטרים הנכונים.

רשימת איתחול לא משפיעה על סדר האיתחול של הרכיבים! סדר האיתחול של הרכיבים הוא תמיד לפי הסדר שבו הם מוגדרים במחלוקת.

### דוגמה למימוש של מחלוקת: Vector

- וקטור הוא אחד מהמכלים הנמצאים בספריה הסטנדרטית והשימושי ביותר.
- וקטור בדומה למערך המבנה בשפה מכיל סדרה של נתונים מסוים סוג, אך יש לו תוכנות נוספות, אפשר להגדילו, להעתיקו, לדעת את גודלו.

```
class Vector {  
    int sz;           // the size  
    double* elem; // a pointer to the elements  
public:  
    using size_type = unsigned long;  
    Vector(): sz{0}, elem{nullptr} {} //default constructor  
    Vector(int s) // constructor with element count  
        :sz{s}, elem{new double[s]} // initialize  
        { for (int i = 0; i<sz; ++i) elem[i] = 0.0; }  
    ~Vector() // destructor  
        { delete[] elem; }  
    int size() { return sz; }  
    Vector v1; // use default constructor, not Vector v1();  
    Vector v2(10); // create a vector with 10 elements
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

nullptr

- אנו רוצים להבטיח כי המצביע תמיד מצביע על אובייקט, בנוסף שהוא מצביע על מקום תקין בזיכרון.
- כאשר אין לנו אובייקט, נתחל את המצביע על ידי nullptr.
- פעם היה נהוג להשתמש ב 0 או NULL, אבל לשימוש זה יש חסרונות:

```
void func(int n);
void func(char *s);
func(NULL); // which function is called? (int)
```

במקרה זה הפונקציה הראשונה תיקרא, לטענו זה למעשה מצביע לכתובת 0 שכן הקומpileר יחשב שהבחירה הכי טובה היא הפונקציה הראשונה, בעוד שהמתכנת כנראה התכוון לשניה.

- שימוש בnullptr מונע למעשה בלבול בין integers an pointers בין :

```
func( nullptr ); // func(char *s) is called
```

```
double* pd = nullptr;
int x = nullptr; // error: nullptr is a pointer
```

### בנייה ברירת מחדל (= default)

- אם המחלקה שלנו לא מדירה במפורש בנסיבות כלשהם, הקומpileר יעשה זאת בשבילנו.
- בנייה ברירת המחדל מתחילה את שדות המחלקה.
- אובייקטים מורכבים (מערכות ופונקציות) מאותחלים עם ערך לא מוגדר.
- אנחנו יכולים לבקש מהקומpileר ליצור עבורנו את בנייה ברירת המחדל על ידי:

```
class Vector {
    Vector() = default;
```

### בנייה שמאכע המרה

בנייה מקבל ארגומנט אחד מגדר המרה מהסוג שלו למחלקה שלו, לדוגמה:

```
class complex {
    complex(double,double);
    complex(double); // defines double-to-complex
                     // conversion
    // . .
};

complex z = complex{1.2,3.4};
z = 5.6;           // OK, converts 5.6 to
                  // complex(5.6,0)
                  // and assigns to z
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### explicit

הבעה היא שההמורות הללו יכולות לגרום לתוצאות בלתי צפויות, ניתן למנוע זאת על ידי הגדרת בניין שהוגדר כ explicit (מופרש) מספק את האופציה לבנות אובייקט בדרך הרגילה בלבד ללא implicit (מרומז), כלומר לא ביצוע המרות מרומות.

```
class Vector {  
    explicit Vector(int);  
    Vector v(10); // OK, explicit  
    v = 40; // error, no int-to-vector conversion
```

### AIR לתחילת וקטור

1. Initialize to default and then assign:  

```
Vector v1(2); // error prone:  
v1[0] = 1.2; v1[1] = 2.4; v1[2] = 7.8; //
```
2. Use push\_back:  

```
Vector v2; // tedious  
v2.push_back(1.2); v2.push_back(2.4); v2.push_back(7.8);  
• push_back is useful for input:  
read(istream& is, Vector& v) {  
    for(double d; is >> d;) v.push_back(d)  
}
```
3. Best use {} delimited list of elements:  

```
Vector v3 = {1.2, 7.89, 12.34}; // C++11
```

נרצה לאפשר גם בחלוקת שאנו יצלנו שימוש בראשית אתחול, לשם כך נדרש להגדיר בניין מתאים:

```
class Vector {  
    int sz; // the size  
    double* elem; // a pointer to the elements  
public:  
    Vector(initializer_list<double> lst) // constructor  
        :sz(lst.size()), elem(new double[sz])  
        { copy( lst.begin(), lst.end(), elem); }  
        // copy using standard library algorithm  
    };  
Vector v1(3); // three elements  
vector v2{3}; // one element  
vector v3 = {3}; // one element
```

### בניין העתקה (ברירת מחדל)

דיברנו על סוגים מיוחדים של בניינים. אחד מהם הוא בניין ללא פרמטרים. עוד בניין מיוחד הוא בניין מעתק. בניין מעתק לחלוקת זה הוא בניין המקבל פרמטר אחד בלבד, שהסוג שלו הוא T.&const

הקומpileר קורא לבניין זהה אוטומטית בכל פעם שצורך להעתיק עצם מהסוג זה לעצם חדש, למשל, כשמעבירים פרמטרים לפונקציות.

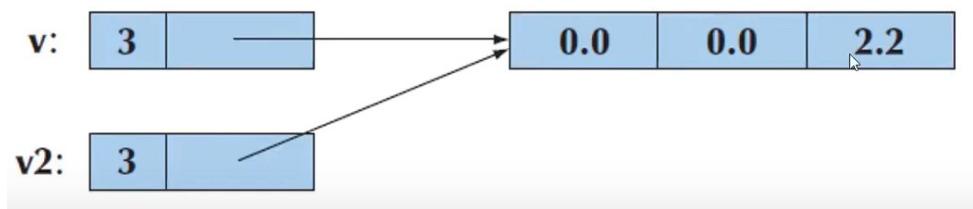
## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

אם לא מגדירים בנאי מעתיק, בירית המהידל היא לבצע העתקת סיביות (bitwise copy). זה בסדר כשמדוחבר במחולקות פשוטות (Point) אבל לא טוב כשמדוחבר במבנים מורכבים עם מצביעים. במקרה שבירית-המhedל לא מתאימה, אנחנו צריכים להגדיר בעצמנו בנאי מעתיק שיבצע "העתקה عمוקה".

שאלה: מדוע הפעמטר ח"ב להיות מסוג T &const ולא פשוט מסוג T? תשובה: כי כדי להבהיר פרמטר מסוג T, הקומפיילר צריך לקרוא לבניי המעתיק, אבל אנחנו מגדירים את הבניי הזה עכשו! אם אנחנו לא מגדירים בנאים הקומפיילר מגדיר עבורנו בנאי בירית מהידל. בנאי בירית המhedל מבצע העתקה רדודה (מבצע השמה פשוטה של השדות), יש לכך בעיות:  
 1. כל שינוי באחד מהוקטורים יוביל לשינוי גם בוקטור השני.  
 2. כאשר ייצאים מהסוכן נקרה, הם המצביעים לאותו מקום בזיכרון, מה שבוביל לשחרור פעמיים של אותו מקום בזיכרון, שהוא שגיאה חמורה.



### בנייה העתקה שמעתיק קרוא'

על מנת להימנע מהבעיות שהציגנו לעילנו להגדיר את בניי העתקה בעצמנו.

```
class Vector {  
    int sz;  
    double* elem;  
public:  
    Vector(const Vector& rhs) ; // define copy  
constructor  
    :sz{rhs.sz}, elem{new double[rhs.sz]};  
    { copy(rhs.elem, rhs.elem+sz, elem); }
```



כך ניצור שני מערכיים שונים, ונמנע מהבעיות שהציגנו קודם לכך.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### השנת העתקה

אופרטור השמה (=) מגדר איך להעתיק עצמים. חשוב במיוחד להגדיר אותו נכון כשהעצמים הם "עומקים" (כוללים הקצת זיכרון דינמית). בדרך כלל, במקרה זה נרצה להגדיר שלוש שיטות: בנאי מעתיק (copy constructor), מפרק (destructor), ואופרטור השמה (assignment operator).

אם השתמש בקוד הבא ללא הגדרת בנאי השמה:

**Vector v(3);**

מה שקרה זה: תחיליה אנו יוצרים וקטור באורך 3, ושמם

**v[2] = 2.2;**

במקומות האחרונים את הערך 2.2, לאחר מכן אנו יוצרים וקטור

**Vector v2(4);**

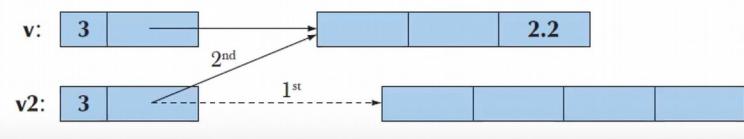
נוסף באורך 4, ומשתמשים במבנה השמה.

**v2 = v;**

מה שקרה זה שתתבצע השמה רדודה:

שני הווקטורים יצביעו לאותו מקום בזיכרון, יהיה לנו בעיה מכיוון

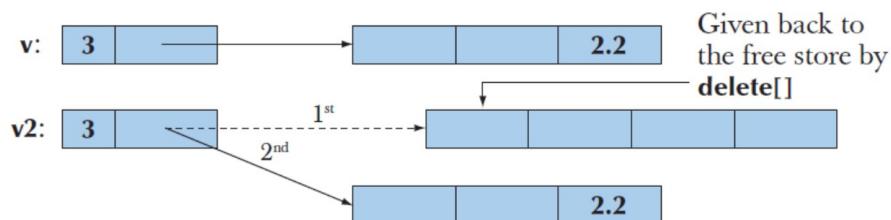
שנעבד את ההצעה לוקטור הראשון ולא יוכל לפרק אותו ביציאה מה SCOPE, כלומר, דליפת זיכרון, בנוסף כמו מוקודם שני הווקטורים יצביעו לאותו מקום בזיכרון ושינוי של אחד יוביל לשינוי השני.



פתרון, דרישת האופרטור =:

```
class Vector {  
    Vector& operator=(const Vector&) ; // copy assignment  
    ...  
    Vector& Vector::operator=(const Vector& rhs)  
    {  
        double* p = new double[rhs.sz]; // allocate space  
        copy(rhs.elem, rhs.elem+rhs.sz, p); // copy elements  
        delete[] elem; // deallocate old space  
        elem = p; // reset elem  
        sz = rhs.sz;  
        return *this; // return a self-reference  
    }  
}
```

- נזכיר עותק של האלמנטים מהווקטור המקורי.
- נשחרר את האלמנטים המקוריים מוקטור היעד.
- אפשריים להציב ערך האלמנטים החדש.



- העתקה רדודה - מעתק את את הפונטער כך שני הפונטרים מצביעים בעת על אותו אובייקט.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

- העתקה عمוקה – מעתק את מה שהפינטרא מציבע עליו כך שני הפינטראים יציבעו על אובייקטים שונים.

### בנאי מעתיק לעומת אופרטור השמה

העתקת עצמים מתבצעת בארכעה מקרים:

1. כשמגדירים עצם חדש מתוך עצם קיים (למשל Complex a=b או (a)(b)).
  2. כשמעברים פרמטר לפונקציה value.
  3. כשמוחזרים ערך מפונקציה value.
  4. בפועל השמה של עצם קיים לעצם קיים אחר (למשל Complex a; a=b).
- במקרים 1, 2, 3 הקומפילר קורא לבנאי מעתיק. במקרה 4 הקומפילר קורא לאופרטור השמה.

### מניעת העתקה (= delete)

אם אנחנו לא מגדירים לבנאים, אנחנו מקבל בחינם מהקומפילר את הבנאי המעתיק, ישנו מקרים בהם נרצה למנוע מאובייקט מסוים להיות מעתק. כמו למשל שימוש בתבנית העיצוב singleton. ניתן לעשות זאת באמצעות הוספה = delete על ה copy constructor באופן הבא:

```
struct NoCopy {  
    NoCopy() = default; // use the default constructor  
    NoCopy(const NoCopy&) = delete; // no copy  
    NoCopy &operator=(const NoCopy&) = delete; // no assignment  
    ~NoCopy() = default; // use the default destructor  
    // other members  
};
```

### רפנסים לעומת פוינטרים

בשפת סי, נדרשנו לקבל כתובות של משתנה (למשל כדי לשЛОח לפונקציה שתוכל גם לשנות אותו), הגדרנו פוינטר למשנה, למשל:

```
int num; int* pnum = &num;
```

כדי להשתמש בפואינטר, נדרש להקדים לו כוכבית, למשל:

```
(*pnum) = 5;
```

ואם המשתנה הוא עצם או struct, נדרש להשתמש בחץ, למשל:

```
Point location; Point* plocation = &location;
```

```
plocation->x = 5; // equivalent to (*plocation).x = 5;
```

בשפת C++ יש דרך נוספת לקבל כתובות של משתנה – להגדיר רפנס (reference):  
 int& rnum = num;

זה מאד דומה, רק שהפעם, אפשר לגשת למשנה בלי כוכבית, למשל:

```
rnum = 5;
```

בלי חץ, למשל:

```
Point& rlocation = location; rlocation.x = 5;
```

## סיכום קורס תכנות מתקדם

כתב על ידי: שי נאור <https://github.com/shaynaor>

mbased על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

חוץ מצורת הגישה, ישנו שני הבדלים עיקריים בין פוינטර לRefPtr:

- אפשר ליצור פוינטר בלי לאתחל אותו (או לאתחל ל-`ptr=null`), אבל כשיוצריםRefPtr – חייבים לאתחל אותו מיד למשתנה. זה אמרור לצמצם את הסיכוי לשגיאות – חשוב לנוRefPtr, אנחנו יכולים להיות בטוחים שיש שם עצם ממש ולא null.
- אפשר לשנות פוינטר לאחר יצירתו, למשל להוסיף לו 1 (ואז הוא יצביע למקום אחר בזיכרון). אבל אי אפשר לשנותRefPtr לאחר יצירתו. גם זה אמרור לצמצם את הסיכוי לשגיאות –RefPtr תמיד מצביע לאותו מקום בזיכרון ולא יכול "לנדוד" למיקומות לא רצויים.

### **lvalue and rvalue**

- lvalue יכול להופיע מצד שמאל של אופרטור השמה.
  - זה אובייקט שניתן לשנות אותו.
- rvalue יכול להופיע מצד ימין של אופרטור השמה.
  - זהו ביטוי זמני שלא ניתן לשינוי.

לדוגמה:

```
y = x + 2; // y is an lvalue, x + 2 is an rvalue
z = 7;      // z is an lvalue, 7 is an rvalue
s = f(x);  // f(x) is an rvalue
x + 2 = y; // Error
7 = z;     // Error
f(x) = s; // Error
```

### **rvalue references**

זה לא חוקי להגיד השמה של rvalueRefPtr:

```
int i = 42;
int& r = i;        // OK
int& r = x + 3;  // Error
```

הקריאה הבאה לפונקציה היא לא חוקית:

```
int f(int& n) { return 10 * n; }
x = f(x + 2);
```

ב C++ הוסיף את האפשרות לrvalueRefPtr:

```
int&& r = x + 3; // OK: note the two ampersands
int&& rr = i;   // Error: cannot reference an lvalue
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### העמסת פונקציות עם && ו- &

```
void ref(int& n) {
    cout << "reference: " << n << endl;
}
void ref(int&& n) {
    cout << "rvalue reference: " << n << endl;
}

int main() {
    int x = 10;
    ref(x);           // lvalue
    ref(x + 10);     // rvalue
    ref(30);          // rvalue
    ref(std::move(x)); // lvalue cast to rvalue
```

### בנייה חזזה

```
Vector::Vector(Vector&& rhs)
:sz{rhs.sz}, elem{rhs.elem} // move rhs.elem to elem
{
    rhs.sz = 0;              // make rhs empty
    rhs.elem = nullptr;
}

vector fill(istream& is) {
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;
}
vector vec = fill(cin);
• Copying res out of fill() and into vec could be expensive,
the move constructor is implicitly used to implement the return
```

### השנת חזזה

```
Vector& Vector::operator=(Vector&& rhs)
{
    delete[] elem;      // deallocate old space
    elem = rhs.elem;   // move rhs.elem to elem
    sz = rhs.sz;
    rhs.elem = nullptr; // make rhs the empty vector
    rhs.sz = 0;
    return *this;       // return a self-reference
}
• If the caller passes an rvalue, the compiler generates code that
invokes the move constructor or move assignment operator
• We thus avoid making a copy of the temporary
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### פעולות הנדרשות במחלקה שמשתמשת במשאים

- מחלקה צריכה מפרק כאשר היא משתמשת במשאים:
- כאשר המחלקה מקצה משאים על הערמה. ככלומר משתמש בnew במפרק נדרש לשחרר משאים אלו ע"י delete.
- משאים שהמחלקה משתמשת בהם כמו: קבועים, טרידים וכו. צריכים להיסגר כראוי במפרק.
- אם למחלקה יש מפרק, סביר להניח שמודדים בה הונקציות הבאות:

```
X() ;                                // default constructor
X(Sometype);                          // ordinary constructor
X(const X&);                         // copy constructor
X(X&&);                            // move constructor
X& operator=(const X&); // copy assignment
X& operator=(X&&);    // move assignment
~X();                                // destructor
```

### פונקציות ומחלקות חברות – friend

לפעמים יש פונקציה שהיא קשורה באופן הדוק למחלקה, אבל אי אפשר להגיד אותה בתוך המחלקה. לדוגמה, אופרטורי קלט ופלט (<> <>).

יש פונקציה שאפשר להגיד בתוך המחלקה אבל יותר יותר להגיד בחוץ. לדוגמה, אופרטור חיבור (+) אפשר להגיד בתוך המחלקה:

```
Complex Complex::operator+ (Complex other)
```

אבל יותר טבעי להגיד אותו מחוץ למחלקה:

```
Complex operator+ (Complex a, Complex b)
```

בדרך כלל, כשפונקציות מוגדרות מחוץ למחלקה, אין להן גישה לשדות הפרטאים של המחלקה. אבל בשפת C++ אפשר לצאת מכל זה ע"י הצהרה שהפונקציה המדוברת היא חברה (friend) של המחלקה. למשל, כדי להגיד את אופרטור החיבור מחוץ למחלקה ולאפשר לו להשתמש בשדות של המחלקה, צריך לכתוב בתוך המחלקה (בקובץ h):

```
friend Complex operator+ (Complex a, Complex b);
```

את המימוש אפשר לכתוב כרגע בקובץ cpp; המימוש יוכל לגשת לשדות הפרטאים של המחלקה. באותו אופן, אפשר להגיד מחלקה חברה: מחלקה אחת יכולה להגיד מחלקה אחרת כ"חברה" ובכך לאפשר לה לגשת לשדות הפרטאים שלה. זה שימושי כשהיש שתי מחלקות הקשורות באופן הדוק זו לזו, למשל: מחלקה של וקטור, ומחלקה של איטרטור על הווקטור זהה.

### העמתת אופרטורים

העמתה (overloading) היא מצב של כמה פונקציות עם אותו שם וארגומנטים מסוימים. בשפת C++, גם אופרטור הוא פונקציה, ולכן אפשר להעמת אופרטורים: להגיד אופרטורים המבצעים פעולה שונות לפי סוג הארגומנטים המועברים אליהם. לדוגמה:

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### העמתת האופרטור []

```
double operator[] (int i) {  
    return elem[i];  
}
```

ע"י ההדרה בדרכו זאת של האופרטור [] אנו מחזירים ערך, כלומר, ניתן לקרוא ערכים מהקטור אך לא ניתן לשנותו.

```
Vector v(10);  
double x = v[2]; // fine  
v[3] = x; // error, v[3] is not an lvalue
```

ע"מ שנוכל לשנות גם את הקטור עליו להחזיר רפרנס למיקום המבוקש בוקטור:

```
double& operator[ ](int n)  
{  
return elem[n];  
}
```

אבל יש לנו בעיה נוספת, לא יוכל לטפל ב `const vector` מכיוון שרק `const member functions` יכולות לפעול על אובייקטים שהם `const`.

הפתרון הוא להגיד פעמיים את האופרטור, באופן הבא:

```
double& operator[] (int n); // for non-const  
const double& operator[] (int n) const; // for const
```

### כללי העמתת האופרטור +

נגדיר אופרטור זה מחוץ למחלקה על מנת לאפשר המרת לאופרנד הימני והשמאלי.

```
Vector operator+(const Vector& a, const Vector& b)  
{  
    if (a.size()!= b.size())  
        throw Vector_size_mismatch{};  
    Vector res(a.size());  
    for (int i = 0 ; i != a.size() ; ++i)  
        res[i] = a[i] + b[i];  
    return res;  
}
```

כעת נוכל לבצע את הפעולה הבאה:

```
Vector r;  
r = x + y;
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### העמסת אופרטור הפלט << לוקטור

```
ostream& operator<<(ostream& os, const Vector& vec)
{
    os << '{';
    int n = vec.size();
    if (n > 0) {
        // Is the vector non-empty?
        os << vec[0]; // Send first element
        for (int i = 1; i < n; i++)
            os << ',' << vec[i];
    }
    os << '}';
    return os;
}
cout << vec1 << endl;
```

ואז נוכל לבצע את הפקודה הבאה:

### איטרטורים

```
class Vector {
    int sz;           // size
    double* elem;   // pointer to the elements
public:
    typedef double* iterator;
    typedef const double* const_iterator;

    iterator begin() { return elem; }
    const_iterator cbegin() const { return elem; }
    iterator end() { return elem+sz; }
    const_iterator cend() const { return elem + sz; }
    . . .
};
```

### tabniot

למה בכלל צריך tabniot ?

במקרים רבים אנחנו צריכים לכתוב פונקציה כללית המתאימה לטיפוסים שונים, אבל מתבצעת בצורה שונה לכל טיפוס.

בשפת סי יכולנו לעשות דבר זהה בעזרת מצביע כללי (void\*), אבל זה בעיתי מכמה סיבות:  
(א) אין בדיקה שהטיפוסים אכן תואמים – אפשר למשל לנסות להחליף מספר שלם עם מחזורות והקומפונילר לא ישים לב.

(ב) הקריאה פחות נוחה – צריך להעביר לפונקציה את גודל הסוג שרצים להחליף.

(ג) הביצוע פחותiesel – צריך להעביר בית בית.  
 בשפת C++ יש דרך נוחה ובטוחה יותר להגדיר פונקציה כללית: template – tabniot.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלו' ותוכנים באינטרנט

air זה עובד ?

כש��דים תבנית, הקומפילר זוכר את ההגדירה אבל עדין לא מייצר שום קוד. תבנית היא לא קוד – היא רק מרשם ליצור קוד.

הקוד נוצר רק כמשמעותו להפעיל את התבנית

- בכל פעם שהקומפילר נתקל בקריאה לפונקציה, ולא מוצא פונקציה עם הפרמטרים המתאימים הוא מחפש התבנית שאפשר להשתמש בה כדי ליצור פונקציה מתאימה. התהילה זהה של ייצור פונקציה מסויימת מתוך תבנית נקרא instantiation (מלשון ייצור).

איפה מגדים תבניות ?

הקומפילר משתמש בתבניות ליצור קוד תוך כדי קומפלציה, ולכן כל תבנית חיבת להיות מוגדרתcola (כולל המימוש) במקום שהקומפילר יכול לראות כאשר הוא בונה את התוכנית הראשית –

כלומר בקובץ `h`. אם נגדיר בקובץ `h` כותרת של התבנית בלבד מימוש, כמו שעשינו לפונקציות רגילים – הקומפילר לא יוכל להשתמש בתבנית זו.

air הקומפילר בוחר לאיזו פונקציה לקרוא ?

כשיש כמה פונקציות עם אותו שם ואוטו מספר פרמטרים, הקומפילר בוחר ביניהם באופן הבא:  
קודם-כל, הוא בוחר את כל הפונקציות עם רמת ההתאמה הגבוהה ביותר (הכי פחות המרות סוגים).

בתוך הקבוצה עם רמת ההתאמה הגבוהה ביותר, הוא בוחר את הפונקציות שהן לא תבניות, ורק אם אין כאלה – הוא מפעיל את התבניות.

הדבר מאפשר לכתוב מבנים כלליים, ויחד איתה, פונקציה ספציפית יותר הופעלת באופן שונה על סוגים שונים. הקומפילר יבחר את הפונקציה הספציפית אם היא מתאימה; אחרת – הוא יבחר את הפונקציה הכללית יותר.

באוטו אוף אפשר להגיד מבנים של מחלקות. אנחנו לא רוצים שהקיטור שלו יהיה מוגדר רק עבור `doubles` אלא עבור כל טיפוס שנרצה, נוכל לעשות זאת באמצעות מבנים:

```
template<typename T>
class Vector {
    T* elem; // elem points to an array of type T
    int sz;
public:
    explicit Vector(int s);
    T& operator[](int i);
    const T& operator[](int i) const;
};
template<typename T>
Vector<T>::Vector(int s) { . . . elem = new T[s]; . . . }
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### חריגות exceptions

- ישנה התלבבותה היכן לוודא חריגות:
- מצד אחד, כותב המחלקה מכיר את המחלקה היטב, ולכן הגיוני שהוא יתמודד עם החריגות.
- מצד שני, מי משתמש במחלקה יידע היטב, באילו מהפעולות יכולות להיות חריגות, ואופן הטיפול שהוא רוצה לאותן חריגות, ולכן הגיוני גם שהוא זה שיתמודד עם אותן החריגות.
- לכן נהוג שמי שכותב את המחלקה ידוע על חריגה, ומישתמש במחלקה טיפול בה כראצנו.

לדוגמה:

- דיווח על חריגה:

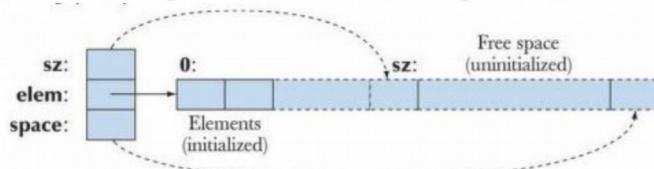
```
double& Vector::operator[](int i)
{
    if (i < 0 || i >= size())
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

- טיפול בחריגה באמצעות try & catch

```
try { // exceptions are handled below
    // v[v.size()] = 7; // returns an undefined value
}
catch (out_of_range) { // oops: out_of_range error
    // ... handle range error ...
}
```

### הוספת push\_back() לוקטור שלנו

```
class Vector {
    int sz;      // size of vector
    int space;   // space allocated
    double* elem; // a pointer to the elements
}
• The default constructor creates an empty vector:
Vector():sz{0}, space{0}, elem{nullptr} {}
```



## **סיכום קורס תכנות מתקדם**

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

```
void Vector::reserve(int newalloc) {  
    double* p new double[newalloc];  
    for (int i=0; i<sz; ++i) p[i] = elem[i];  
    delete[] elem;  
    elem = p;  
    space = newalloc;  
}
```

```
void Vector::push_back(double val) {  
    if (space == 0) reserve(8);  
    else if (sz == space) reserve(2*space);  
    elem[sz] = val;  
    ++sz;  
}
```

## **נושא 5 – הורשה ורבי צורתיות**

### **הורשה**

ירושה בשפת C++ עובדת, בגדול, באופן דומה ל-Java. שני הבדלים חשובים הם:

- אין ממשקים.
- יש ירושה מרובה – מחלוקת אחת יכולה לרשף כמה מחלוקות.  
למה בכלל משתמשים בירושה?
  - כדי שתוכנית שלנו תשקוף את המיציאות: ירושה מבטאת קשר של "a-is" -- "הוא סוג של". למשל, אם המחלוקת של "מנהל" ירושת את המחלוקת של "עובד", זה מעביר את המסר של "מנהל הוא סוג של עובד".
  - כדי להציג פולימורפיזם בזמן ריצה – ע"י החלפת שיטות (overriding). בנגדוד לפולימורפיזם בזמן קומPILEציה שימושיים ע"י העמסת שיטות (overloading).
  - הירשה נועדה להגדיר מחלוקות שיש להן מכנה משותף.
    - המחלוקת המורישה נקראת מחלוקת הבסיס.
    - המחלוקת היורשת נקראת המחלוקת הנגזרת.
  - הירשות מימוש
    - מחלוקת הבסיס מספקת למחלוקת היורשת פונקציות ונתונים מוכנים.
  - הירשות ממשק
    - אפשרות שימוש בחלוקת היורשות השונות באמצעות הממשק של מחלוקת הבסיס המשותפת.
    - את המחלוקות היורשות נקצת בזיכרון הדינמי באמצעות new.
    - ניתן "יחס" למחלוקת הבסיס.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### אופן הגדרת ההוורשה

```
class Base {  
    int x;  
};  
class Derived : public Base {  
    int y;  
};  
int main()  
{  
    Base base;  
    Derived derived;  
    cout << sizeof(base) << '\n'; //---> 4  
    cout << sizeof(derived) << '\n'; // ---> 8
```

נשים לב שכתבנו public Base בהוורשה, המשמעות היא שאנחנו יורשים את התכונות מ Base כפי שהם. קלומר מה שהיא private נשאר כך וכו'.

```
Base base2(base); // copy constructor  
Derived derived2(derived);  
Base base3(derived); // copy constructor  
Derived derived3(base); // Error  
cout << sizeof(base3) << '\n'; // -->4  
  
base2 = derived; // assigment operator  
derived2 = base; // Error  
cout << sizeof(base2) << '\n'; // → 4  
}
```

### מחלקה יורשת Upcasting and Downcasting

```
Base* basep = &derived;           // OK, upcasting  
Base& baser = derived;          // OK, upcasting  
  
Derived* derivedp = &base;         // Error, downcasting  
Derived& derivedr = base;        // Error, downcasting  
  
Derived* derivedp2 = basep;       // Error  
  
cout << "basep*: " << sizeof(*basep) << '\n';
```

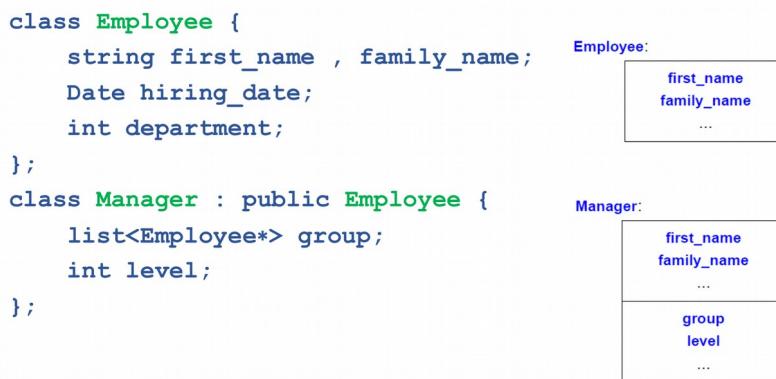
- המרה של מצביע של המחלקה הירשת למחלקת הבסיס נקראת casting .
- דבר זה מותר תמיד ללא צורך ב casting מפורש .

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

- המחלקה היורשת מכילה בתוכה את מחלקת הבסיס, אך כל דבר שאנו עושים על מחלקת הבסיס אנו יכולים לעשות על המחלקה היורשת.
- המרה של מצביע של מחלקת הבסיס למצביע של המחלקה היורשת נקרא Downcast.
- הקומפיאילר לא מהפסר המרה כזאת ללא casting במפורשות.
- הסיבה לכך היא שהמחלקה היורשת יכולה להוסיף נתונים חדשים שלא היו במחלקת הבסיס.



## שימוש בפונקציה של מחלקת הבסיס

```
class Employee {
public:
    void print() const;
// ...
};

class Manager:public Employee // ...
{
public:
    void print() const;
// ...
};
```

```
void Manager::print() const
{
    // print Employee info
    Employee::print();
    // print Manager info
    cout << level;
}
```

## הצורך בפונקציה וירטואלית

פונטור של מחלקת הבסיס יכול להצביע על אובייקט של המחלקה היורשת:

```
void f(Manager m1, Employee e1, Employee e2)
{
    list<Employee*> elist {&m1, &e1, &e2};
    print_list(elist);
}
```

## סיכום קורס תכנות מתקדם

כתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלוי ותכנים באינטרנט

הבעיה היא שהפונקטר של מחלוקת הבסיס יכול להפעיל רק את הפונקציות המוגדרות בחלוקת הבסיס. לעומת אם המחלוקת היורשת דורשת פונקציה מסוימת של מחלוקת הבסיס לא יוכל להפעיל אותה בדרך ה"רגילה":

```
void print_list( const list<Employee*>& elist ) {  
    for( auto x : elist )  
        x -> print();  
}
```

במקרה זה הפונקציה `print()` שתוועל היא הפונקציה שמוגדרת בחלוקת הבסיס.

### פונקציה וירטואלית

ברירת-הchodל ב- + C++, היא, שהקומפיילר בוחר איזו שיטה להריץ, לפי סוג המשתנה בזמן הקומPILEציה. למשל, אם הגדרנו עצם מסוג Programmer, אז שמו עליי רפרנס או מצביע מסוג Person, אז הקומפיילר יבחר את השיטה של Person ולא את השיטה המחליפה. זה מנוגד לג'אבה – בג'אבה השיטה נבחרת לפי סוג העצם בזמן ריצה. אם רוצים לקבל ב- + C את אותה התנהגות של ג'אבה, צריך להגדיר את השיטות הרלוונטיות כווירטואליות – בעזרת שימוש במילת המפתח `virtual`. כמשמעותם שיטה כווירטואלית בחלוקת-הבסיס, הקומפיילר יבחר את השיטה בכל המחלוקות היורשות ממנה לפי סוג העצם בזמן ריצה. כל שיטה המוגדרת כווירטואלית בחלוקת-הbasis, היא וירטואלית באופן אוטומטי גם בכל המחלוקות היורשות ממנה. עם זאת, ממקובל לטען גם את השיטות בחלוקת היורש, העצם החדש הוא מסוג הבסיס, ולכן גם אם השיטה היא וירטואלית, הקומפיילר יקרה לשיטה של מחלוקת הבסיס. כדי ליהנות מה יתרונות של שיטות וירטואליות, צריך להשתמש ברפרנס או בפינטער.

### קריאה לשיטות וירטואליות משיטות אחרות

אם שיטה f מוגדרת כווירטואלית בחלוקת הבסיס, ושיטה g בחלוקת הבסיס קוראת לה – אז הגירסה של שיטה f שתיקרא בפועל תלואה בסוג העצם בזמן ריצה. ש לזה שני יצאי-דופן:

- אם השיטה g היא בניין של מחלוקת-הbasis.
- אם השיטה g היא מפרק של מחלוקת-הbasis.

בשני המקרים האלה, שיטה f שתיקרא בפועל היא זו של מחלוקת הבסיס. מדוע? בגלל סדר הבניה והפירוק:

- כשובנים את מחלוקת הבסיס – המחלוקת היורשת עדין לא בנויה ולכן מסוכן לקרוא לשיטות שלה – אולי יש שדות חשובים שעדיין לא מאוחzählים.
- כשמפרקים את מחלוקת הבסיס – המחלוקת היורשת כבר מפרקת, ולכן שוב מסוכן לקרוא לשיטות שלה – אולי יש שדות חשובים שכבר נמחקו.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותוכנים באינטרנט

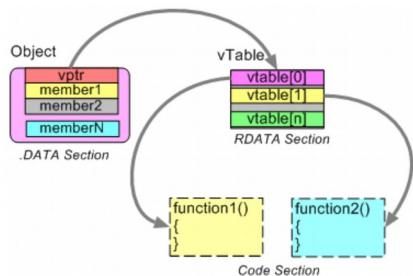
### מימוש שיטות וירטואליות – מה קורה מאחרי הקלעים?

כשכותבים שיטה וירטואלית – איך המחשב יודע לאיזו גירסה לקרוא?

- עבור כל מחלוקת עם שיטות וירטואליות, מוגדרת טבלת שיטות וירטואליות. הטבלה הזאת היא למעשה מערך של מצביעים לפונקציות. עבור כל פונקציה וירטואלית, יש בטבלה הזאת מצביע לימוש שלה בפועל.
- בכל עצם מהמחלקה, יש מצביע לטבלת השיטות הווירטואליות.
- כאשרנו כותבים קוד שקורא לשיטה וירטואלית, הקומפайлר למעשה כותב קוד שקורא את המצביע לטבלת השיטות הווירטואליות מהעצם שנמצא בזיכרון, הולך לכניסה המתאימה בטבלה (לפי שם הפונקציה שקבענו לה), ומפעיל את המימוש המתאים.

**שימוש לב:** בשפת + C אפשר לבחור איזה שיטות יהיה וירטואליות ואיזה לא. הבחירה תלולה באופן שבו אנחנו רוצים להשתמש בשיטות. שיטה לא וירטואלית היא מהירה יותר וגם חסכונית יותר בזכרון. אבל שיטה וירטואלית מאפשרת פולימורפים.

לעומת זאת, בשפות אחרות כגון Smalltalk, Python, Java, כל השיטות וירטואליות. לכן, בכל עצם יש מצביע לטבלת הווירטואלית, וכלקריאה לפונקציה עוברת דרך הטבלה הווירטואלית. זה מבוצע גם מקום וגם זמן. בשפת + C הפילוסופיה היא "לא השתמשת – לא שילמת".



### מחלקה אבסטרקטית

מחלוקת עם פונקציה וירטואלית נקראת מחלוקת אבסטרקטית. איך מסמנים פונקציה כווירטואלית? נסיף = 0 בסוף ההגדירה שלה. לא ניתן ליצור אובייקטים ממחלוקת אבסטרקטית. על מחלוקת היורשת מחלוקת אבסטרקטית למש את כל הפונקציות של המחלוקת האבסטרקטית.

```
class Shape {  
public:  
    virtual Point center() const = 0; // pure virtual  
    virtual void move(Point to) = 0;  
    virtual void draw() const = 0;  
    virtual void rotate(int angle) = 0;  
    virtual ~Shape() {}  
}
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### override

- פונקציה במחלקה הירושת override (דורשת) פונקציה וירטואלית במחלקת הבסיס אם לפונקציה זו יש לבדוק את אותה חתימה.
- על מנת לסייע שפונקציה מסוימת דורשת את הפונקציה במחלקת האם ניתן להוסיף את המילה השמורה override בסופה של אותה הפונקציה.

```
class Smiley : public Circle {  
// ...  
    void move(Point to) override;  
    void draw() const override;  
    void rotate(int) override;  
}
```

### RTTI – Run Time Type Identification

אם יש מצביע למחלקה הבסיס ורוצים לצבע פעולות הקשורות למחלקה הירושת RTTI מאפשר לקבוע בזמן ריצה לאיזה אובייקט להצביע בפועל. dynamic\_cast: ממיר מצביע למחלקה הבסיס, למצביע למחלקה הירושת. אם המצביע למחלקה הבסיס לא מצביע בפועל למחלקה הירושת, הערך המוחזר הוא nullptr. בתוכנית שכתובה היטב אין צורך בRTTI.

```
class Base { virtual void f() {} };  
class Derived : public Base { };  
int main()  
{  
    Base *pBase1 = new Base;  
    Base *pBase2 = new Derived;  
    Derived *pD;  
    pD = dynamic_cast<Derived*>(pBase1); // pD = nullptr  
  
    cout << (pD ? "OK" : "nullptr") << "\n";  
    pD = dynamic_cast<Derived*>(pBase2); // OK  
    cout << (pD ? "OK" : "nullptr") << "\n";  
}
```

## **סיכום קורס תכנות מתקדם**

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

ניתן להשתמש ב `dynamic_cast` לשם קבלת מידע:

```
Shape* ps {read_shape(cin);}
// is a Smiley pointed to by ps ?
if (Smiley* p = dynamic_cast<Smiley*>(ps)) {
    // returns pointer to Smiley
    // yes, a Smiley
}
else {
    // returns nullptr
    // not a Smiley, try something else
}
```

## **נושא 6 – מחרוזות וביטויים רגולריים**

### **מחרוזות**

- בשפת C מחרוזת היא מערך של תוים שמסתאים בתו שערכו הוא 0.
- בשפת C אין מחלפונקציות חברות ואין העמת אופרטורים, וכך לubarד מחרוזת משתמש בפונקציות שמקבלות מחרוזות כפרמטר: `strcpy()`, `strcat()`.
- בשפת C `+` + נשתמש במחילקה `string` שמאפשרת לubarד מחרוזות בצורה נוחה:

```
s1 = s2      // Assign s2 to s1
s1 += x      // Add a character or a string at end
s1 + s2      // Concatenation
s1 == s2     // Comparison
s1 < s2      // Lexicographical comparison
s.size()     // Number of characters
s.c_str()    // C-style version of string
```

### **IMPLEMENTATION\_OF\_STRING**

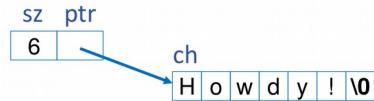
- נשתמש במערך של תוים שמסתאים ב-0 לייצוג המחרוזת בתוך המחליקה, זה מפשט העתקה מחרוזות שמיוצגת בסגנון C.
- כדי ליעל, מחרוזות קצרות ישמרו בתול האובייקט, בחרוזות ארוכות ישמרו בזיכרון הדינמי.
- מחרוזת קצרה בתוך האובייקט תישמר במערך `ch`.
- המשתנה `max_short` מכיל את הגודל המרבי (15) לשמרת המחרוזת בתוך האובייקט.
- בשני המקרים, המשתנה `ptr` יצביע על התו הראשון במחרוזת.
- כפי שעשינו בעבר עם וקטור, גם כאן נקצתה זיכרון נוספת כדי ליעל הוספה של תוים.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

```
class String {  
private:  
    static const int short_max = 15;  
    int sz; // number of characters  
    char* ptr;  
    int space; // unused space on free store  
    char ch[short_max+ 1]; // leave space for 0 (16)  
  
public:  
    String(); //default constructor: x{""}  
    String(const char* p); // C-style: x{"Euler"}  
    String(const String&); // copy constructor  
    String& operator= (const String&); // copy assignment  
    String(String&& x); // move constructor  
    String& operator= (String&& x); // move assignment  
    ~String() { if (sz > short_max) delete[] ptr; }  
    const char* c_str() { return ptr; } // C-style access  
    int size() const { return sz; } // number of elements  
};
```



### ימוש הבנאים

בנאי דיפולטיבי, מגדר מחרוזת ריקה:

```
String::String() : sz{0}, ptr{ch}  
{  
    ch[0] = 0; // terminating 0  
}
```

בנאי המקבל מחרוזת בסגנון C:

```
String::String(const char* p) :  
    sz(strlen(p)),  
    ptr((sz<= short_max) ? ch : new char[sz+ 1]),  
    space{0}  
{  
    strcpy(ptr,p); // copy characters into ptr from p  
}
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### ביטויים רגולריים

- ביטוי רגולרי הוא מחרוזת המתארת תבנית של טקסט.
- ביטוי רגולרי משמש לחיפוש והחלפה של טקסט景德י לבודוק תכונות של טקסט.
- הפונקציות הבאות ב C + + מאפשרות שימוש בביטויים רגולריים:
  - regex\_match: פונקציה המנסה להתאים את הביטוי הרגולרי לכל המחרוזת.
  - regex\_search: פונקציה המנסה להתאים את הביטוי הרגולרי לחלק מהמחרוזת.
  - regex\_replace: פונקציה המחליפה מופעים של הביטוי הרגולרי בטקסט אחר.
  - sregex\_iterator: יוצר איטרטור שמאפשר מעבר על כל התוצאות בטקסט.

### ( )regex\_max

```
string input;
regex pat("abc"); // print Match iff input = "abc"
regex pat("[abc]"); // print Match iff input = 'a' or 'b' or 'c'
regex pat("\ \d"); // String literals: "\n" the compiler delete the first \ so we stay with "\d"
regex pat(R"(\d)"); // Raw string:R symbol tells the compiler to leave the string as it is.
while (true){
    cout << "Enter text:" << endl;
    if(!(cin >> input)) break;
    if(regex_match(input, pat)) cout << "Match" << endl;
    else cout << "No Match" << endl;
}
```

- הסימן של סוגרים מרובעים נקרא character class.
- מחרשת התאמה של תו בודד מבין התווים הרשומים בין סוגרים.
- מקף '-': יוצר טווח כאשר הוא ממוקם בין שני תוים בתוך סוגרים מרובעים.
- מתאים לכל התווים (גדולים וקטנים) ומספר: [9-0-zA-Za-z]
- כובע '^': שולץ את התווים הבאים לאחריו בסוגרים מרובעים.
- לא מספרים [9-0^]

### תווים שמורים ב regex

- ישנו שישו סימונים הcool'ם :backslash'ם

- ספירה בודדת: \d
- כל תו פרט למספר: \D
- כל אות, מספר או קו תחתון: \w
- רווח, טאב, ירידת שורה: \s
- \ כלתו שהוא לא: \S

Examples:

\d-\d matches 1-2, 3-4

\w\w-\d\d matches Ab-12, 12-34 – digits are in \w

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

- ישנו 12 תווים פיסוק שמורים בביטויים רגולריים, נקראים meta-characters:
  - כל ביטוי רגולרי שלא כולל את אחד מהתווים האלה מתאר פשוט את הביטוי עצמו:  
                \$() .^\* [? .\]
- אם נרצה להשתמש באחד התווים הללו ללא כל משמעות מיוחדת נשים לפניהם backslash :
  - ככלומר, הביטוי : ?.\.+\* \ מתאים לטעט ?.+\*
- נשים לב شبמידה ולא להשתמש במחוזות raw אז נctruck להוסיף שני slash מאחר והקומפיילר מוחק אחד.
  - נשים לב כי בתרוים בשימוש אין את הסוגרים המרובעים והמסולסים הסוגרים, וזאת מאחר שהם נחוצים למינוחדים רק כאשר מגיעה לפניהם הסוגר הפותח המתאים.
- נשים לב בנוסף, שיש תווים שיש להם משמעות שונה בין אם הם בתוך סוגרים מרובעים [] או לא:
  - נקודת '.' :תו מיוחד מחוץ לסוגרים מרובעים, אבל לא בתוכם.
  - דASH '-' :תו מיוחד בתוך סוגרים מרובעים (בין שני תווים), אבל לא מחוץ.
  - כובע '^' :משמעות אחרת מחוץ לסוגרים המרובעים, וכאשר הוא מופיע בתחילת הסוגרים.
- נקודת '.' :מתאים לכל תו פרט לירידת שורה.
  - [\Z] מתאים לכל תו כולל ירידת שורה. באופן דומה עבור w,d.

## סימן חוזרות ב regex

- כוכבית '\*' : לאחרתו מסוים או ביטוי משמעויה אף או יותר חוזרת.
- פלוס '+' : לאחרתו מסוים או ביטוי משמעויה אחד או יותר חוזרת.
- אם תרצה מספר מדויק של חוזרות מסוימת: {n} , עבור n חוזרות.
- עבור טווח חוזרות מסוים מסומן: {m,n} , בין n ל m חוזרות.
- סימן שאלה '?' : לאחרתו מסוים או ביטוי משמעויה אף או חוזרת אחת.

- Examples:

[A-Za-z\_][A-Za-z\_0-9]\* an identifier in a programming language

0[xX][A-Fa-f0-9]+ C-style hexadecimal number

10{100} a googol (10<sup>100</sup>)

[A-Fa-f0-9]{1,8}h? 1-8 digit hexadecimal number with an optional h suffix

colou?r matches both colour and color

A\*B+ C? matches AAABBB, BC, B does not match AAA, AABBC

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

### greedy and lazy matches

- To match an HTML tag:  
This is a `<EM>first</EM>` test  
you may attempt to use  
`<.+>`  
but this matches `<EM>first</EM>`, not `<EM>`
- The reason is that `*`, `+` and `{n,m}` are **greedy**, they repeat the preceding token as many times as possible (longest match)
- `.*[0-9]+` applied to "Copyright 2003", `[0-9]+` matches only "3"
- You can make them **lazy** instead of **greedy** by putting a `?` after them  
`<.+?>`  
or use the pattern: `<[^>]+>`
- `(ab)+` matches all of `ababab`, however `(ab)+?` matches only the first `ab`

### התאמת בתחילת ובסוף שורה

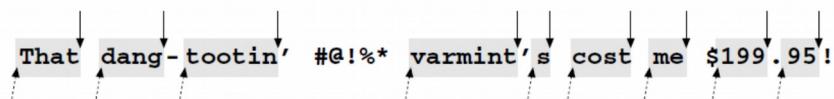
- התווים המיוחדים `^`, `$` נקראים עוגנים.
- הם לא מבצעים התאמת לשוםתו, המטרה שלהם במקומות מסוימים, היא לעגן את הביטוי הרגולרי.
- כובע '`^`' מבצע התאמת אם היא מתרחשת בתחילת המחרוזת.
- דולר '`$`' מבצע התאמת אם היא מתרחשת בסוף המחרוזת.

Examples:

- `^cat` a line that begins with cat, matches catxxx
- `cat$` a line that ends with cat, matches xxzxcat
- `^cat$` a line that consists of only cat, matches cat
- `^$` matches an empty line

### Match whole words

- Create a regex that matches the word `cat` in "A cat and a mouse", but not in `category` or `bobcat`  
Place the word `cat` between two word boundaries `\bcat\b`
- The regular expression token `\b` is called a word boundary, it matches at the start or the end of a word
- The first `\b` requires the `c` to occur at the very start of the string, or after a non-word character.
- The second `\b` requires the `t` to occur at the very end of the string, or before a non-word character



## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### Match one of several alternatives

- The **vertical bar or pipe symbol |**, splits the regular expression into multiple alternatives
- **Mary|Jane|Sue** matches Mary, or Jane, or Sue with each match attempt
- The regular expression finds the **leftmost match**:  
When you apply **Mary|Jane|Sue** to  
**Jane, Mary and Sue went to Mary's house**  
the match Jane is found first  
The match that begins earliest (leftmost) wins
- Each alternative is checked in a left-to-right order:  
**Jane|Janet** matches **Jane in Her name is Janet**

### Group parts of the match

- Improve the regular expression for matching Mary, Jane, or Sue by forcing the match to be a whole word
- Use grouping to achieve this with one pair of word boundaries for the whole regex, instead of one pair for each alternative
- **\b(Mary|Jane|Sue)\b** has three alternatives: Mary, Jane, and Sue, all three between two word boundaries
  - This regex **does not match** anything in **Her name is Janet**
- The alternation operator, has the **lowest** precedence of all regex operators
  - If you try **\bMary|Jane|Sue\b**, the three alternatives are **\bMary**, **Jane**, and **Sue\b**
  - This regex matches **Jane in Her name is Janet**

### Group Parts of the Match

- Examples
- **Nov(ember)?** matches November and Nov (**greedy**)
- **Feb(ruary)? 23(rd)?** matches many alternatives
- **\b(one|two|three)\b** Find a line containing certain words:
- **(\s|:|,)\*(\d+)** spaces, colons, commas followed by a number
- **(?\s|:|,)\*(\d\*)** parentheses that **don't** define a subpattern
- **<HR( +SIZE \*= \*[0-9]+)? \*>** <HR SIZE=30>
- **\\$[0-9]+(\.[0-9][0-9])?** Dollar amount with optional cents
- We have seen two uses for grouping parentheses:
  - to limit the scope of alternation
  - to group multiple characters into larger units to which you can apply quantifiers like question mark and star

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

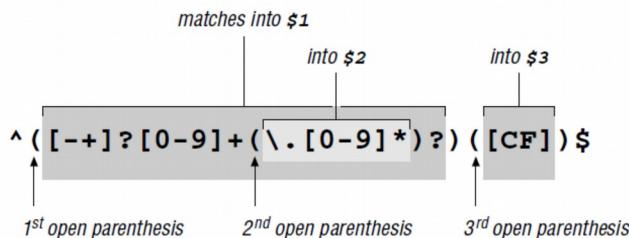
### Capture parts of the match

- Create a regular expression that matches any date in **yyyy-mm-dd** format, and separately **captures** the **year**, **month**, and **day**
- A pair of **parentheses** isn't just a group, it's a **capturing group**
- Captures become useful when they cover only part of the regular expression, as in **\b(\d\d\d\d)-(\d\d)-(\d\d)\b**
- The regex **\b\d\d\d\d-\d\d-\d\d\b** does exactly the same, but does not capture
- Captures are numbered by counting opening parentheses from left to right
- There are three ways you can use the captured text:
  - match the captured text again within the same regex match
  - insert the captured text into the replacement text
  - The program can use the parts of the regex match

### Capture parts of the match

- Example, convert temperatures:

**^([-]?[0-9]+(\.[0-9]+)?)([CF])\$**



- or group but do not capture:

**^([-]?[0-9]+(?:\.[0-9]+)?)([CF])\$**

- Now, the text that the parentheses surrounding [CF] match, goes to \$2

### Match previously matched text again

- Create a regular expression that matches "magical" dates
- A date is magical if the year minus the century, the month, and the day of the month are all the same numbers
- For example, 2010-10-10 is a magical date:

we first have to capture the previous text, then we match the same text using a **back-reference**

**\b(\d\d(\d\d)-\1\1\b**

The **(\d\d)** matches 10, and is stored in capturing group 1

The **back-reference \1** matches the **10** of the **month** and **day**

- Match a pair of opening and closing HTML tags:  
**<([A-Z][A-Z0-9]\*[^>]\*>.\*?</1>**

- Checking for Doubled Words (the the):

**\b(\w+)\s+\1\b**

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

```
string input;           regex_search(), smatch
regex pattern(R"(\d+)");
smatch result;
while (true) {
    cout<<"Enter:"<<endl;
    if(!(cin >> input)) break;
    if(regex_search(input, result, pattern)) {
        cout<<"Match prefix: "<<result.prefix()<<endl;
        cout<<"Match string: "<<result[0]<<endl;
        cout<<"Match suffix: "<<result.suffix()<<endl;
    }
    else cout<<"No Match"<<endl;
}


|            |      |     |             |            |
|------------|------|-----|-------------|------------|
|            | m[0] |     |             |            |
| m.prefix() | m[1] | ... | m[m.size()] | m.suffix() |


```

```
void use() {           regex_search(), smatch
    ifstream in("file.txt"); if (!in) cerr << "no file\n";
    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"}; //postal code
    int lineno = 0;
    smatch matches; // matched strings go here
    for (string line; getline(in,line);) {
        ++lineno;
        if (regex_search(line , matches, pat)) {
            cout << lineno << ":" << matches[0] << '\n';
            if (1 < matches.size() && matches[1].matched)
                cout << "\t: " << matches[1] << '\n'; // sub-match
        }
    } } // TX77845 and DC 20500-0001 match
```

## regex\_replace()

Replace all matchings of pattern:

```
string input {"x 1 y2 22 zaq 34567"};
regex pat {"(\w+)\s+(\d+)"}; // word space number
string format "{$1,$2}\n";
cout << regex_replace(input,pat,format);
```

- The output is:

```
{x,1}
{y2,22}
{zaq,34567}
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

## sregex\_iterator

Replace all matchings of pattern:

```
regex reg("([A-Za-z]+) \\\1");
string target = "the the cow jumped over over the fence";
sregex_iterator reg_begin =
    sregex_iterator(target.begin(), target.end(), reg);
sregex_iterator reg_end = sregex_iterator();
for (sregex_iterator it = reg_begin; it != reg_end; ++it) {
    cout << "Substring: " << it->str() << ", ";
    cout << "Position: " << it->position() << endl;
}
cout << "Found: " << distance(reg_begin, reg_end) << endl;
```

The output is:

```
Substring: the the, Position: 0
Substring: over over, Position: 19
Found: 2
```

## Special Characters

### Regular Expression Special Characters

.	Any single character (a “wildcard”)	\	Next character has a special meaning
[	Begin character class	*	Zero or more (suffix operation)
]	End character class	+	One or more (suffix operation)
{	Begin count	?	Optional (zero or one) (suffix operation)
}	End count		Alternative (or)
(	Begin grouping	^	Start of line; negation
)	End grouping	\$	End of line

### Repetition

{n}	Exactly n times
{n,}	n or more times
{n,m}	At least n and at most m times
*	Zero or more, that is, {0,}
+	One or more, that is, {1,}
?	Optional (zero or one), that is {0,1}

### Character Class

\d	A decimal digit
\s	A space (space, tab, etc.)
\w	A letter (a-z) or digit (0-9) or underscore (_)
\D	Not \d
\S	Not \s
\W	Not \w

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

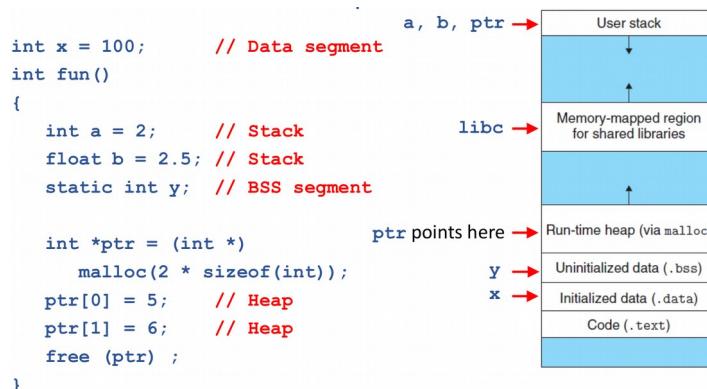
מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותוכנים באינטרנט

### נושא 7 – ניהול זיכרון

#### חלק 1: ניהול זיכרון

- תכנית מקזה מקום בזיכרון עבור כל אובייקט (נתון) של התוכנית.
- אובייקטים גלובליים (אובייקטים שמוגדרים מחוץ לפונקציות), וסטטיים (אובייקטיה שמוגדרים כ static) נמצאים בזיכרון מרגע שהתוכנית מתחילה לזרז ועד שהיא מסתימה.
- אובייקטים מקומיים (שמוגדרים בתוך פונקציה או מחלוקת) נמצאים בזיכרון למשך זמן הריצה של הפונקציה או הבלוק בו הם מוגדרים.
- אובייקטים דינמיים נמצאים בזיכרון החל מההקצתה שלהם תוך כדי ריצת התוכנית ועד לשחרור.
- תוכנית משתמשת בהקצתה דינמית:
  - אם בזמן כתיבת התוכנית לא יכול מספר האובייקטים.
  - אם בזמן כתיבת התוכנית לא ידוע סוג האובייקט.
  - כדי לאפשר שתוּף של האובייקט.

#### מפת הזיכרון של תוכנית



#### malloc and free

• הפקצת זיכרון דינמית מתבצעת על הערימה.

• הפקaza מנוהלת בדרך כלל עד ידי פונקציית הספרייה malloc:

```
void *malloc(size_t size)
// size — number of bytes
```

Examples:

```
char* str;
str = (char *) malloc(13); // explicit casting needed in C++
strcpy(str, "Hello World!");
ptr = (int*) malloc(100 * sizeof(int));
for (i = 0; i < 100; ++ i)
ptr[i] = i;
```

• כדי למנוע דליפת זיכרון, התוכנית צריכה לשחרר את הזיכרון שהוקצה ע"י free().

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותוכנים באינטרנט

### New and delete

- שימוש ב `w +` `malloc()` and `free()` in C++ rather than `new` and `delete`.
- ב C++ אנו משתמשים ב `new` ו `delete`.
- `New` הוא אופרטור שגורם לLOCATION מקום וגורם `malloc` את הבנייה המתאימים.
- `Delete` הוא אופרטור שגורם `free` וגורם `malloc` את הזיכרון.

לדוגמה:

```
MyClass *fp = new MyClass(1,2);
```

תחילה `malloc(sizeof(MyClass))` נקרא ולאחר מכן נקרא הבנייה של `MyClass` עם הפרמטרים (1,2).

### new and delete for arrays

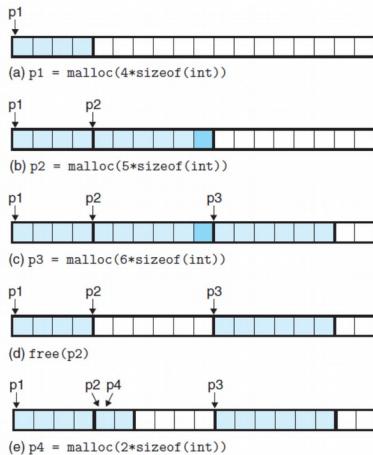
```
MyType* fp = new MyType[100];
```

- במקרה זה הבנייה נקרא עבור כל אובייקט בזיכרון.
- בבני דיפולטיבי חיב להיוות מוגדר במקרה זה, מאחר שבבנייה ללא פרמטרים חיב לקרוא עבור כל אובייקט בזיכרון.
- על מנת לפרק את כל האובייקטים שבזיכרון משתמש ב `delete []`。

```
delete [] fp;
```

#### Dynamic Memory Allocator

- Example:
- The heap consists of (4 bytes) words (each box)
- Allocations are aligned on (8 bytes) double words
- Notice that after the call to `free`, the pointer **p2** still points to the freed block



### Allocator Goals

- מקסום תפקוד.
- הגדלת מספר הבקשות שהושלמו בכל זמן ליחידה.
- צמצם של פיצולים.
- פיצול פנימי מתרחש כאשר הבלוק שהוקצה גדול יותר מהגודל המבוקש.
- אילוצי alignment (עיגול לכפולת של 8).
- גודל מינימלי על בלוקים שהוקצו.
- פיצול חיוני מתרחש כאשר יש מספיק זיכרון פנוי בצד לספק את בקשת ההקצאה, אך אף בלוק פנוי לא גדול מספיק כדי לטפל בבקשתה.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

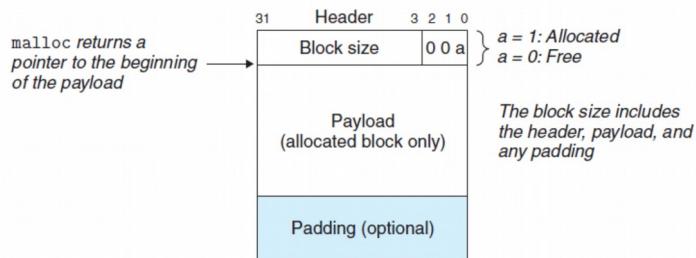
מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### Allocator implementation

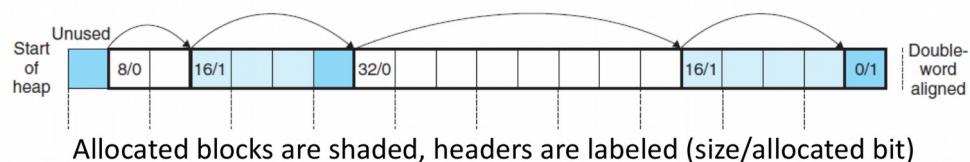
- אופן ארגון שחרור הבלוקים. איך אנחנו עוקבים אחר בלוקים משוחררים?
- איפה למקם בלוק חדש?
- פיצול לאחר מקום בלוק חדש?
- מה לעשות עם בלוק שזה עתה שוחרר? Coalescing

#### Implicit Free Lists

- Header of block contains size and whether allocated or free
- Since blocks are double-word aligned, we need only 29 bits for the block size, freeing the remaining 3 bits for other information



- The free blocks are linked **implicitly** by the size fields in the headers
- We are using the allocated bit to indicate whether the block is allocated
- The allocator can traverse the entire set of **free** blocks by traversing **all** of the blocks in the heap



### Placing Allocated Blocks

- כאשר תכנית מבקשת בלוק בגודל  $k$  בתים, המקצתה מ Chapman ברשימת הבלוקים הפנויים בלוק מתאים.
- ישנו מספר פרדיוגמות חיפוש:
  - First fit מתחילה לסרוק מתחילה הרשימה, מקצתה את הבלוק הראשון שמתאים.
  - Next fit מתחילה לחפש במקום שבו בוצעה ההקצתה הקודמת.
  - בוחר את הבלוק החופשי בגודל הקטן ביותר שמתאים. Best fit

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

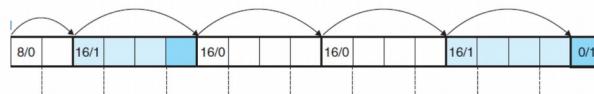
מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט

### Splitting and Coalescing Free Blocks

- ברגע שהמקרה הקצה בлок חופשי, עליו להחליט האם להשתמש בכל הבלוק החופשי או לפרק את הבלוק לשני חלקים.
- כאשר המקרה משחרר בלוק שהוא מוקצה קודם לכן, עשוי להיות בלוקים שכנים שהם גם משוחזרים.
  - המקרה יכול למזג coalesce את הבלוקים השכנים.

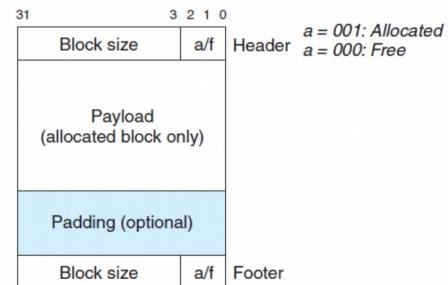
### Coalescing

- Coalescing the next free block is straightforward and efficient
  - The header of the current block points to the header of the next block, which can be checked to determine if the next block is free
  - If so, its size is added to the size of the current header and the blocks are coalesced in constant time
- But for coalescing the previous block, the only option would be to search the entire list
  - With implicit free list, this means that each call to free would require time linear in the size of the heap



### Coalescing with Boundary Tags

- Boundary tags, allow for constant-time coalescing of the previous block
- The idea, is to add a footer (the boundary tag) at the end of each block, where the footer is a replica of the header
- The footer is one word away from the start of the current block
- So the allocator can determine the starting location and status of the previous block by inspecting its footer
- If we were to store the allocated/free bit of the previous block in one of the excess low order bits of the current block, then allocated blocks would not need footers



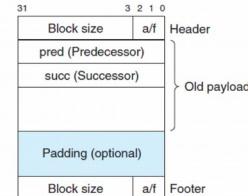
## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

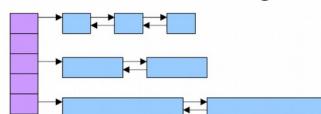
### Explicit Free Lists

- With implicit free list, block allocation time is **linear** in the total number of heap blocks
- A better approach is to organize the **free** blocks into an explicit doubly linked list
- The pointers that implement the list can be stored within the bodies of the **free blocks**
- Reduces the first fit allocation time to **linear** in the **number of free blocks**
- Can be maintained in last-in first-out (LIFO) order by inserting **newly freed** blocks at the beginning of the list
  - freeing a block can be performed in **constant time**
  - If boundary tags are used, then **coalescing** can also be performed in **constant time**



### Multiple Free Lists

- A single linked list of free blocks requires time linear in the number of free blocks to allocate a block
- Can maintain **multiple free lists**, where each list holds blocks that are the same size
- To free a block, the allocator inserts the block at the front of the list
- Allocating** and **freeing** blocks are both fast **constant-time** operations
- No splitting**, and **no coalescing** means there is little per-block overhead
- The **size** of an allocated block can be inferred from its **address**, allocated blocks require **no headers**
- However may cause **internal** and **external** fragmentation



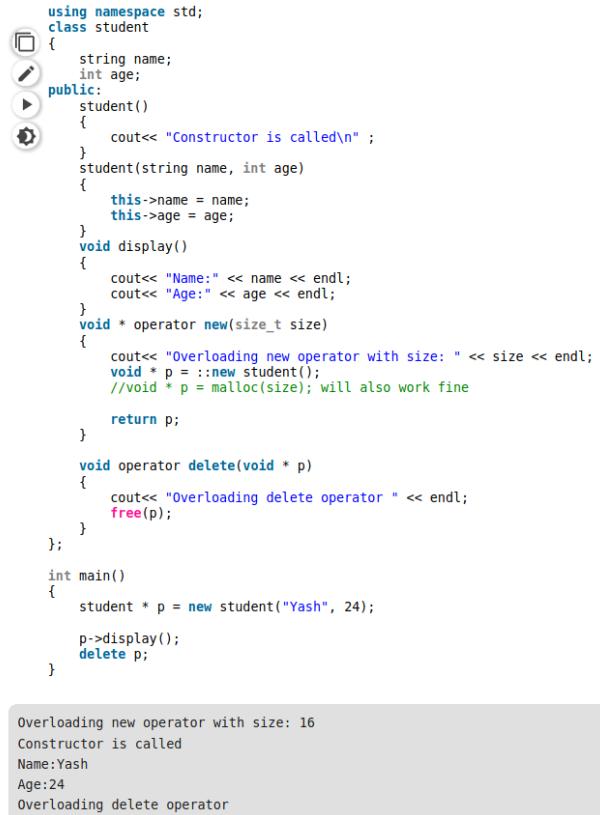
## העמסת האופרטורים new and delete

- ניתן להעמס את ה global allocator או לשימוש ב allocator אחר עבור המחקלה שלנו.
- הקומפיילר יקרא לאופרטור new שהעמסנו, ולאחר מכן יקרא לבניית המתאים.
- האופרטור new מקבל כקלט size\_t ומחזיר void \* פיניטר לאובייקט בגודל זהה. מחזיר 0 או זורק חריגה במידה ולא נמצא מקום להקצתה.
- הקומפיילר יקרא ל delete המועמס לאחר שהמפרק נקרא.
- delete מקבל כקלט void \* ומחזיר void.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותכנים באינטרנט



```
using namespace std;
class student
{
    string name;
    int age;
public:
    student()
    {
        cout<< "Constructor is called\n" ;
    }
    student(string name, int age)
    {
        this->name = name;
        this->age = age;
    }
    void display()
    {
        cout<< "Name:" << name << endl;
        cout<< "Age:" << age << endl;
    }
    void * operator new(size_t size)
    {
        cout<< "Overloading new operator with size: " << size << endl;
        void * p = ::new student();
        //void * p = malloc(size); will also work fine
        return p;
    }
    void operator delete(void * p)
    {
        cout<< "Overloading delete operator " << endl;
        free(p);
    }
};

int main()
{
    student * p = new student("Yash", 24);
    p->display();
    delete p;
}
```

```
Overloading new operator with size: 16
Constructor is called
Name:Yash
Age:24
Overloading delete operator
```

## בעיות עם ניהול זיכרון

- דליפת זיכרון: אי שחרור זיכרון שהוקצה על הערימה.
- שחרור כפול של זיכרון: שחרור כפול של אותו איזור בזיכרון.
- קורה כאשר שני פינטרים מצביעים לאותו מקום בזיכרון.
- עלול לפגוע בערימה או לפרק בטעות אובייקט אחר שהוקצה.
- שימוש לאחר שחרור: שימוש באובייקט לאחר שכבר שוחרר מהזיכרון.
- הפינטר עדין מחזק את הכתובת של הזיכרון ששוחרר, ויתכן כי המיקום הזה מוקצה כבר בעבר אובייקט אחר.

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחס ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

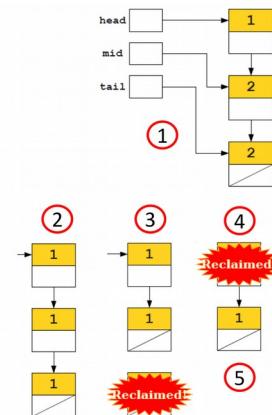
### Automatic Management – Reference Counting

- שומר עם האובייקט שדה שמספר את מספר הפינטרים המופנים לאותו אובייקט.
- כל יצירה של רפנס חדש לאובייקט מעלה את המונה.
- כל שחרור של רפנס כזה מורידה את המונה.
- אם המונה מגע ל-0 האובייקט משוחרר מהזיכרון.

דוגמה:

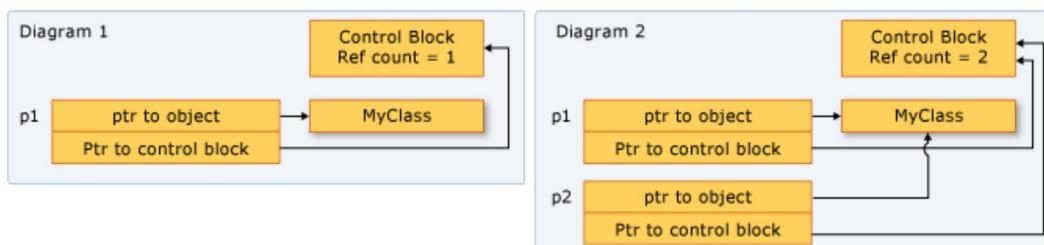
Reference Counting Example

```
struct LinkedList {
    LinkedList next;
}
void f() {
    LinkedList *head = new LinkedList;
    LinkedList *mid = new LinkedList;
    LinkedList *tail = new LinkedList;
    head->next = mid; ①
    mid->next = tail;
    mid = tail = null; ②
    head->next->next = null; ③
    head = null; ④ ⑤ // all reclaimed
}
```



### Smart Pointers – Shared\_ptr

- מחלוקת עם האופרטורים → \* שהועמסו.
- המחלוקת מכילה פינטר אמרית ומונה עבור מספר ההצבעות על אותו אובייקט.
- שימוש במבנה של shared\_ptr מעלה את המונה.
- שימוש במפרק של shared\_ptr מוריד את המונה.
- המפרק של shared\_ptr מפרק את האובייקט ומנקה אותו מהזיכרון כאשר המונה מגע ל-0.
- יכול להועתק, להועבר לפונקציות assigned by value, by reference shared\_ptr .



```
shared_ptr<string> p1; // shared_ptr to a string
shared_ptr<int> p2(new int(1024)); // direct initialization
shared_ptr<string> p4 = make_shared<string>(10, '9');
auto p = make_shared<int>(100);
auto q(p);
auto r = make_shared<int>(200);
r = q;
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

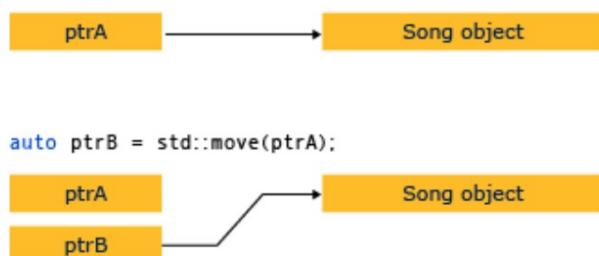
```
void use_shared()
{
    auto p = make_shared<int>(300);
    // use p
}
// p goes out of scope; the memory is automatically freed
```

### Smart pointers- unique\_ptr

- unique\_ptr הוא מצביע חכם קטן ומהיר לניהול משאבים.
- לunique\_ptr אין מונה הסופר את מספר ההצבעות אליו.

הוא לא יכול להיות מועתק לאובייקט unique\_ptr אחר, ולא ניתן להעבירו לפונקציה by value (כנראה שבמימוש שלו הגדרו שלא ניתן להשתמש במבנה המעתיק שלו ולען לא ניתן לבצע העתקה ולהעבירו לפונקציות by value).

ניתן "להציג" unique\_ptr, הבעלות על המשאב זיכרון תועבר לunique\_ptr אחר:



```
unique_ptr<double> p1;
// there is no library function make_shared (in c++14 make_unique added)
unique_ptr<int> p2(new int(42));
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1);// error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p2;// error: no assign for unique_ptr
// transfers ownership from p1 to p2
unique_ptr<string> p2(p1.release()); // release makes p1 null
unique_ptr<string> p3(new string("Trex"));
// transfers ownership from p3 to p2
p2.reset(p3.release()); // reset deletes memory to which p2 pointed
unique_ptr<int> clone(int p) { // copy ok: about to be destroyed
    return unique_ptr<int>(new int(p));
}
```

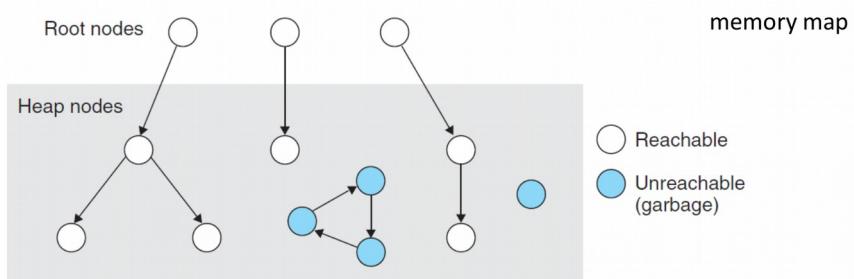
## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלו' ותכנים באינטרנט

### Garbage collection

- כאשר אנו מקצים מקום בעירמה באופן מפורש באמצעות `malloc`, באחריות המתכנת לשחרר את כל הדיארן שהוקצה.
- Garbage collector הוא משחרר אוטומטי בлокים שהוקטו על העירמה שאינם נחוצים יותר.
- ניתן להגיע לאובייקט אם קיימ רפרנס כלשהו אליו, אם לא קיים רפרנס לאובייקט אז הוא לא נחוץ.
- מטרת ה Garbage collector היא לזהות ולשחרר אובייקטים לא נחוצים.
- ה Garbage collector רואה בזיכרון גרף נגישות מכיוון



- הגרף מחולק ל root nodes | heap node: מתחאים למשתנים הנמצאים עם המחסנית.
- Heap node: מתחאים לבlokים שהקצו אותם על העירמה.
- קודקוד q הוא נגיש אם יש מסלול אליו מ root node כלשהו.
- קודקוד לא נגיש נחשב כ"זבל" מאחר שלא ניתן להשתמש בו שוב בתכנית.
- קודקוד מוגדר כזבל אם הוא אינו נגיש.
- שלב ה mark: שבו מסמנים את כל הקודקודים הנגישים.
- שלב ה sweep: שבו משחררים את הקודקודים הלא נגישים – שלא סומנו.

```
typedef void *ptr;
ptr isPtr(ptr p) // If p points to an allocated block,
// returns a pointer to that block

void mark(ptr p) {
    if ((b = isPtr(p)) == NULL)
        return;
    if (blockMarked(b))
        return;
    markBlock(b);
    len = length(b);
    for (i=0; i < len; i++)
        mark(b[i]);
    return;
}

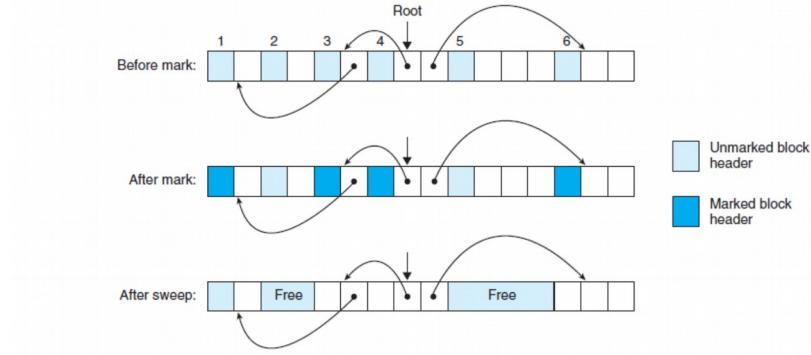
void sweep(ptr b, ptr end) {
    while (b < end) {
        if (blockMarked(b))
            unmarkBlock(b);
        else if (blockAllocated(b))
            free(b);
        b = nextBlock(b);
    }
    return;
}
```

## סיכום קורס תכנות מתקדם

נכתב על ידי: שי נאור <https://github.com/shaynaor>

מבוסס על הרצאות והמצגות של ד"ר פנחים ויסברג, ד"ר אראל סגל הלי ותוכנים באינטרנט

- The **mark function** is called for each root
- The **mark function** calls itself recursively on each word in block
- The **sweep function** is called once, it iterates over each block in the heap



דיון מעניין ב גיבוב סט אוברפלו על :Garbage collection <https://stackoverflow.com/questions/147130/why-doesnt-c-have-a-garbage-collector>