C++

שמירה אחרונה: ‏07 יוני 18. שעה: ‏10:15:00

Contents

[Labmda expressions 2](#_Toc515868669)

[Lambda capture clause 2](#_Toc515868670)

[COPY C-TOR & REF TYPES 3](#_Toc515868671)

[Explicit ctor 5](#_Toc515868672)

[templates 5](#_Toc515868673)

[Using an existing template 5](#_Toc515868674)

[defining function template 6](#_Toc515868675)

[class templates 7](#_Toc515868676)

[Standard containters 8](#_Toc515868677)

[vector 8](#_Toc515868678)

[List 10](#_Toc515868679)

[map 10](#_Toc515868680)

[STL Algorithm 10](#_Toc515868681)

[headers you should know 11](#_Toc515868682)

[count; count\_if; for( iterator); all\_of; any\_of; none\_of 11](#_Toc515868683)

[find, find\_if 12](#_Toc515868684)

[sort 13](#_Toc515868685)

[C++11 Language Features 15](#_Toc515868686)

[auto 15](#_Toc515868687)

[decltype 16](#_Toc515868688)

[Etc… 16](#_Toc515868689)

[בידוד הקוד מהקבצים שמפיק הקומפיילר 16](#_Toc515868690)

[Assertions 17](#_Toc515868691)

[Emit Debug messages 17](#_Toc515868692)

[Named Casts 18](#_Toc515868693)

[Time measuring in windows 18](#_Toc515868694)

[MEMBER FUNCTION POINTER 19](#_Toc515868695)

[Smart Pointers : Demo: unique\_ptr 20](#_Toc515868696)

[shared\_ptr 21](#_Toc515868697)

## Labmda expressions

### Lambda capture clause

*C++ fund. 2. Kate.*

הפונקציה for\_each מבצעת איטרציה על איברים של 'אוסף' ומגישה כל איבר אל פונקציה שהמתכנת מגדיר.

הקוד הבא טריוואלי



בעזרת ביטוי למדא אפשר לחסוך את הצורך להגדיר את פונקצית print() בנפרד.



שים לב שזו ממש העתקה של הפונקציה, אותו פרמטר פורמלי, אותו קוד לביצוע.

ביטויי למדא גם מחזירים ערכים. הביטוי הבא מקבל int ומחזיר דאבל. אפשר לוותר על ה -> double במידה והקומפיילר יכול להסיק בעצמו מה טיפוס הערך המוחזר (בדרך כלל כאשר הביטוי הוא שורה אחת שמחזירה ערך), אבל אם בתנאי מסוים הביטוי מחזיר int ובתנאי אחר הוא מחזיר double המתכנת צריך לרשום במפורש מה כוונתו



הכתיב הזה ששם את הדבל לאחר החץ הוא מאולץ משום שכתיבת הדבל לפני הסוגריים הייתה נותנת כאן משמעות אחרת. ומכל מקום כתיב זה מותר גם בהגדרה רגילה של פונקציות ולפעמים מאד נוח להשתמש בו לכתחילה.



## COPY C-TOR & REF TYPES

#include <iostream>

#include <string>

using namespace std;

class Resource

{

public:

string GetName() const;

};

Resource r3;

// the ‘return’ shall create a copy of r1 (in the same way that an object is copied when passed as formal arg), and thus copy ctor shall be activated.

Resource GetRes1()

{

Resource r1;

return r1;

}

// here, ‘return’ does NOT duplicates r1. This will cause an exception at run time, since the return object is located on the stack which is no longer valid when the method returns. gcc gave a warning, VS gave warning just after increasing the warning level to 4.

Resource**&** GetRes2()

{

Resource r1;

return r1;

}

// here also the copy ctor will be activated. This means that the caller will NOT get r3 but rather **a copy of** r3.

Resource GetRes3()

{

return r3;

}

// this one is good. The global r3 shall be returned. No copy ctor.

Resource**&** GetRes3a()

{

return r3;

}

void f()

{

**const Resource& r = GetRes1();**

זה תקין כל זמן שיש const אשר מבטיח שלא יהיו נסיונות לשנות את r. אבל אם לא נותנים const אז gcc צועק:

*error: invalid initialization of non-const reference of type 'Resource&' from an rvalue of type 'Resource'*

שהרי ההעתק שהקומפיילר יצר לפני החזרה אינו 'נגיש' במובן המקובל.

לעומת זאת הסטודיו נתן רק אזהרה וגם זה רק ברמה 4 והקוד רץ.

(דומה שאין סיבה אמיתית לכך שהקוד לא יתקמפל מבלי const, אמנם יש כאן rvalue אבל אפשר 'לעבוד איתו'. נראה ש gcc קצת יותר מרובע...)

// C

{

Resource r = GetRes2();

cout << r.GetName() << endl;

}

זו פעולה לא תקינה. כאמור לעיל הקומפיילר נותן אזהרה בגוף המתודה ואם המתכנת לא התייחס אליה, בזמן ריצה ה GetName יזרוק מצב חריג שהרי r מתייחס למקום לא חוקי.

// D

{

Resource& r = GetRes2();

cout << r.GetName() << endl;

}

כמו C, גם כאן הקומפילציה תעבור והבעיה תופיע בזמן ריצה.

// E

{

Resource r = GetRes3();

cout << r.GetName() << endl;

}

כדאמרינן לעיל. הפעולה בזבזנית וגם מטעה משום שעובדים כאן על העתק, אבל זה עובד.

// F

{

const Resource& r = GetRes3();

cout << r.GetName() << endl;

}

כנ"ל. זה שהמשתנה המקבל הוא ref לא משנה את העובדה שהפונקציה עצמה משכפלת את האובייקט המוחזר. גם כאן gcc יקמפל רק אם יש const בעוד שהסטודיו הוא סלחני (לטוב ולמוטב)

// G

{

Resource r = GetRes3a();

cout << r.GetName() << endl;

}

תקין אבל בזבזני. **לאחר** חזרת הפונקציה יופעל copy ctor וב r יהיה העתק.

// H

{

Resource& r = GetRes3a();

cout << r.GetName() << endl;

}

זו הדרך הנכונה. מוחזר ref לאובייקט הגלובלי.

}

### Explicit ctor

class Resource

{

public:

Resouces(const int size);

string GetName() const;

};

לקלאס זה יש ctor אשר מקבל ארגומנט אחד. זה מאפשר אתחול בצורה הבאה:

Resouces r = 10;

זהו אתחול סמוי, implicit.

לא תמיד זה טוב משום שהקומפיילר יעשה כמיטב יכולתו להמיר טיפוסים לצורך אתחול כזה, למשל:

Resouces r = ‘a’;

ולא בטוח שהמתכנת הסכים לכך. הדרך למנוע זאת היא פשוט להשתמש במילה השמורה explicit

class Resource

{

public:

explicit Resouces(const int size);

string GetName() const;

};

כאן, המתכנת יהיה חייב לכתוב

Resouces r(‘a’);

### uninform initialization

Also called brace initialization.

*see UniformInit.cpp*

Examples:

ResourceWithCtor r(""); eqv to ResourceWithCtor r1{ "" };

ResourceWithoutCtor rwc; eqv to ResourceWithoutCtor rwc1{};

ResourceWithoutCtor rwc1 = {};

init directly public vars (can’t be done if ctor exists)

ResourceWithCtor2args { "", 1 };

ResourceWithoutCtor rwc3 = { "", 1};

init array

int v[] = { 1, 2, 3 };

int \* vp = new int[]{1, 2, 3};

## templates

### Using an existing template

דוגמת שימוש ב pair

using std::pair;

using std::string;

pair<int, string> ValueToString(int val)

{

string s = "someString";

return pair<int, string>(val, s);

}

הערה: בעוד ש pair מוגבל לשני ארגומנטים בלבד, tuple **אינו מוגבל**, זהו variadic template ואם כן נשאלת השאלה איך אפשר לקבל את הארגומנטים ללא שם שמזהה אותם?

כאן נכנס כתיב יחודי למקרה זה:



Get<0> מחזיר את הארגומנט הראשון (וזה לאו דווקא ב tuple)

### defining function template



כאשר משתמשים בפונקציה תבניתית אז בניגוד לקלאס, אין צורך לומר במפורש מה טיפוס הארגומנטים. הקומפיילר מסיק זאת בעצמו לפי סוג המשתנים ואם אפשר לפרש זאת בשני דרכים אז המתכנת צריך לכתוב מפורש. לדוגמא

max<double>(10, 10.1)

כאן, 10 יומר לדבל. כאן יש שימוש ב decltype משום שהפונקציה עצמה משתמשת ב pair אשר מבקש שיגדירו לו את הטיפוסים מראש (אמנם היה אפשר להשתמש בשורה שבהערה, זה סתם תרגיל)

חשוב להבין שקביעת טיפוס הארגומנטים נעשה בזמן קומפילציה כלומר הקומפיילר מייצר קוד המותאם לסוג זה של המשתנים ואם יש שימוש באותה פונקציה עם משתנים מטיפוס אחר הקומפיילר לא יוכל להשתמש בקוד הקיים אלא לייצר קוד אחר)

### class templates

כאן יש חסרון מאד משמעותי. בקלאס תבניתי אין להפריד את ה H מה CPP. כלומר להכניס את גוף המתודות במקום הגדרתם. (VS2013)

אמנם יש דרכים להתגבר על זה אבל הם די מסורבלות.

## Standard containters

*PS: C++ Advanced Topics; Kate Gregory*

**

### vector

(כלל אצבע: אם צריך קונטיינר, זה ברירת המחדל)

std::vector<Resource> vp;

{

Resource r("x");

vp.push\_back(r);

}

מה קורה בקוד הנ"ל? r הוא מקומי ויוצא מסקופ בסיום הבלוק. ואם כן מה מצבו של הווקטור?

התשובה הפשוטה שהיא שהווקטור לא לוקח את r כמות שהוא, הוא מעתיק אותו! בעזרת copy c-tor

נמצא שבווקטור יש העתק **נפרד** של r, ואם כן, במידה וצריך העתקה עמוקה, באחריות המתכנת להעמיס את ה copy c-tor

בצורה הבאה, השימוש ב unique\_ptr מונע את הצורך בשכפול האובייקט

vector<unique\_ptr<Resource>> vp1;

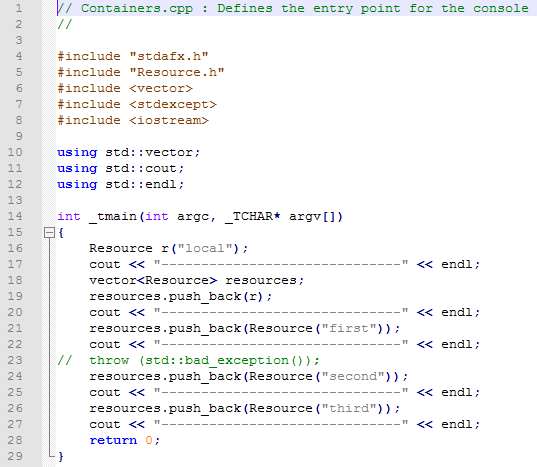
vp1.push\_back(unique\_ptr<Resource>(new Resource("x1")));

הדוגמא הבאה מתוך Cpp.Adv. Topics. Kate.G.

C++ Advanced Topics by Kate Gregory

adv-cpp\materials\advcpp-03-containers-exercise-files\Demo\Begin\Containers\Containers.sln

הדגמה של הפעלת ctors ובפרט לאור העובדה שהווקטור בונה את עצמו מחדש כאשר האורך שלו לא מספיק ומשמעות הדבר הפעלת ה copy ctors של כל האלמנטים שבתוכו.



line 19: copy ctor “local”

line 21: vector enlarge. existing elements copy constructed to new instance, and elements of old instance destroyed

1. first ctor (volatile instance)
2. first copy-ctor.
3. local copy ctor.
4. local dtor
5. first dtor (volatile instance)

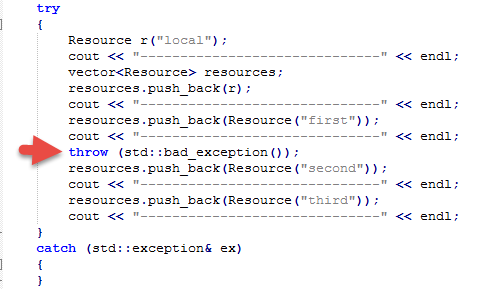
line 24: vector now contains 2 elements.

1. second ctor (volatile instance)
2. second copy-ctor
3. first copy-ctor.
4. local copy ctor.
5. local dtor
6. first dtor
7. second dtor (volatile instance)

בשורת 28 בהצהרת return הווקטור יוצא מסקופ וכל ה dtors של האלמנטים המוכלים נקראים.

המחברת מציעה לסגור את הקטע ב try-catch אפילו אם אין כלום ב catch כדי שאם תהיה בעיה אז הווקטור יצא מסקופ לפני הכניסה ל catch וישחרר את כל האלמנטים שלו [כאשר נמנעתי מ try-catch אכן ראיתי שנוצרה זליגת זכרון]

כך זה נראה:



### List

PS: C++ Advanced Topics by Kate Gregory

adv-cpp\materials\advcpp-03-containers-exercise-files\Demo\Begin\ListAndVector\

וקטור שומר את האיברים שלו ברצף. דילוג אל איברN הוא טריוואלי. החסרון הוא כמובן כשרוצים להוסיף אלמנטים לוקטור נתון. הוא פשוט צריך לבנות את עצמו מחדש ואז מופעלים ה copy-ctor של כל אלמנט וכו'. קצת כבד.

לעומת זאת List זו רשימה משורשרת, האיברים אינם רצופים בזכרון.

אמנם אפשר בקלות להוסיף אלמנטים אבל גישה לאיבר N אינה פשוטה, ואפילו חישוב מספר האיברים זו משימה שדורשת לעבור על כל האיברים.

בדוגמא הנ"ל יש השוואת ביצועים

### map

This is a sorted lookup table.

### iteration

vector<string> v = {"one", "two", "three"};

for(auto it = begin(v); it!=end(v); it++) { cout << \*it << "\n"; }

for(vector<string>::reverse\_iterator it = v.rbegin(); it != v.rend(); it++) …

for(vector<string>::const\_iterator it = v.cbegin(); it != v.cend(); it++) …

for(auto & x : v) { cout << x << "\n"; }

## STL Algorithm

PS: Beautiful C++: STL Algorithms, Kate.G

### headers you should know



### count; count\_if; for( iterator); all\_of; any\_of; none\_of

// Count.cpp :

#include "stdafx.h"

#include <vector>

#include <string>

#include <algorithm>

#include <map>

using namespace std;

int main()

{

vector<int> v{ 2,7,1,6,2,-2,4,0 };

//count how many entries have the target value (2)

int twos = 0;

int const target = 2;

for (size\_t i = 0; i < v.size(); i++)

{

if (v[i] == target)

{

twos++;

}

}

twos = count(v.begin(), v.end(), target);

twos = count(begin(v), end(v), target);

//count how many entries are odd

int odds = 0;

for (auto elem : v)

{

if (elem % 2 != 0)

{

odds++;

}

}

odds = count\_if(begin(v), end(v), [](auto elem) {return elem % 2 != 0; });

map<int, int> monthlengths{ { 1,31 },{ 2,28 },{ 3,31 },{ 4,30 },{ 5,31 },{ 6,30 },{ 7,31 },{ 8,31 },{ 9,30 },{ 10,31 },{ 11,30 },{ 12,31 } };

int longmonths = count\_if(begin(monthlengths), end(monthlengths), [](auto elem) {return elem.second == 31; });

//are all, any, or none of the values odd? (Conclude from number of odd entries)

bool allof, noneof, anyof;

allof = (odds == v.size());

noneof = (odds == 0);

anyof = (odds > 0);

allof = all\_of(begin(v), end(v),

[](auto elem) {return elem % 2 != 0; });

noneof = none\_of(begin(v), end(v),

[](auto elem) {return elem % 2 != 0; });

anyof = any\_of(begin(v), end(v),

[](auto elem) {return elem % 2 != 0; });

return 0;

}

### find, find\_if

// Find.cpp :

#include "stdafx.h"

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

int main()

{

vector<int> v{ 4, 6, 6, 1, 3, -2, 0, 11, 2, 3, 2, 4, 4, 2, 4 };

string s{ "Hello I am a sentence" };

//find the first zero in the collection

auto result = find(begin(v), end(v), 0);

int weLookedFor = \*result;

//find the first 2 after that zero

result = find(result, end(v), 2);

if (result != end(v))

{

weLookedFor = \*result;

}

//find the first a

auto letter = find(begin(s), end(s), 'a');

char a = \*letter;

//find first odd value

result = find\_if(begin(v), end(v), [](auto elem) {return elem % 2 != 0; });

weLookedFor = \*result;

//find first even value

result = find\_if\_not(begin(v), end(v), [](auto elem) {return elem % 2 != 0; });

weLookedFor = \*result;

// Finds the first character equal to one of the characters in the given character sequence.

vector<int> primes{ 1,2,3,5,7,11,13 };

result = find\_first\_of(begin(v), end(v), begin(primes), end(primes));

weLookedFor = \*result;

//  Searches for the first occurrence of the subsequence of elements

vector<int> subsequence{ 2,4 };

result = search(begin(v), end(v), begin(subsequence), end(subsequence));

weLookedFor = \*result;

result++; result++;

int six = \*result;

string am = "am";

letter = search(begin(s), end(s), begin(am), end(am));

a = \*letter;

result = find\_end(begin(v), end(v), begin(subsequence), end(subsequence));

result++; result++;

if (result != end(v))

{

weLookedFor = \*result;

}

result = search\_n(begin(v), end(v), 2, 4);

result--;

int two = \*result;

result = adjacent\_find(begin(v), end(v));

six = \*result;

result++;

six = \*result;

return 0;

}

### sort

// Sorting.cpp :

#include "stdafx.h"

#include <vector>

#include <algorithm>

#include <cmath> // for abs

#include "Employee.h"

#include <random>

using namespace std;

int main()

{

vector<int> v{ 4,1,0,1,-2,3,7,-6,2,0,0,-9,9 };

auto v2 = v;

sort(begin(v2), end(v2));

sort(begin(v2), end(v2),

[](int elem1, int elem2) {return elem1 > elem2; });

sort(begin(v2), end(v2),

[](int elem1, int elem2) {return abs(elem1) > abs(elem2); });

std::vector<Employee> staff{

{ "Kate", "Gregory", 1000 },

{ "Obvious", "Artificial", 2000 },

{ "Fake", "Name", 1000 },

{ "Alan", "Turing", 2000 },

{ "Grace", "Hopper", 2000 },

{ "Anita", "Borg", 2000 }

};

//std::sort(begin(staff), end(staff)); //- only works if operator< defined for Employee

std::sort(begin(staff), end(staff),

[](Employee e1, Employee e2) {return e1.getSalary() < e2.getSalary(); });

std::sort(begin(staff), end(staff),

[](Employee e1, Employee e2) {return e1.getSortingName() < e2.getSortingName(); });

std::sort(begin(staff), end(staff),

[](Employee e1, Employee e2) {return e1.getSortingName() < e2.getSortingName(); });

std::stable\_sort(begin(staff), end(staff),

[](Employee e1, Employee e2) {return e1.getSalary() < e2.getSalary(); });

auto sorted = is\_sorted(begin(v2), end(v2));

sorted = is\_sorted(begin(v2), end(v2),[](int elem1, int elem2) {return abs(elem1) > abs(elem2); });

int high = \*(max\_element(begin(v), end(v)));

int low = \*(min\_element(begin(v), end(v)));

sort(begin(v2), end(v2));

low = \*begin(v2);

high = \*(end(v2)-1);

int positive = \*upper\_bound(begin(v2), end(v2), 0);

std::sort(begin(staff), end(staff),

[](Employee e1, Employee e2) {return e1.getSortingName() < e2.getSortingName(); });

auto p = std::lower\_bound(begin(staff), end(staff), "Gregory, Kate",

[](Employee e1, std::string n) {return e1.getSortingName() < n; });

int sal = p->getSalary();

random\_device randomdevice;

mt19937 generator(randomdevice());

shuffle(begin(v2), end(v2), generator);

partial\_sort(begin(v2), find(begin(v2), end(v2), 4), end(v2));

int breakpoint = \*is\_sorted\_until(begin(v2), end(v2));

vector<int> v3(3);

partial\_sort\_copy(begin(v), end(v), begin(v3), end(v3));

v2 = {1,5,4,2,9,7,3,8,2};

int i = \*(begin(v2) + 4);

nth\_element(begin(v2), begin(v2)+4, end(v2));

i = \*(begin(v2) + 4);

return 0;

}

is\_sorted מוודא שאוסף הוא כבר מסודר.

stable\_sort מתייחס לסידור משני. נאמר שמסדרים את העובדים לפי משכורות, ואת העובדים בעלי אותה משכורת מסדרים לפי שם.

מציאת אלמנט מקסימלי או מינימלי מבלי לסדר את האוסף: max\_element(begin(v), end(v)); min\_element(begin(v), end(v));

וכן, מציאת אינדקס האיבר הראשון הגדול מערך נתון, או הראשון הקטן מערך נתון lower\_bound upper\_bound

shuffle מערבל אוסף. הפוך מ sort.

## C++11 Language Features

PS- Alex Korban

### auto

המילה השמורה auto אשר בעבר שימשה לסמן משתנים מקומיים, משמשת עכשיו כהנחייה לקומפיילר להסיק בעצמו את הטיפוס הנדרש. יש מקומות שזה נוח מאד ומקל על שינויים עתידיים בפרט כאשר עובדים עם קונטיינר:

for (**auto** it = a.begin(); it < a.end(); it++)

אפשר להחליף את טיפוס הקונטיינר מבלי לשנות את הקוד שמתייחס אליו. חשוב!

ויש מקומות שזה הכרחי כמו ב templates כאשר לא ידוע מראש מהם הטיפוסים ורוצים להגדיר טיפוס חדש המבוסס עליהם.



אמנם זה לא מספק, משום שבאופן זה אי אפשר להגדיר טיפוס מוחזר על סמך טיפוסי הכניסה, לשם כך ניתן להשתמש ב decltype.

אי אפשר לתת auto בתור member (שהרי אי אפשר לאתחל וממילא אין אפשרות לקומפיילר להסיק) ופשוט שאין לתיתו בארגומנט של פונקציה.

### decltype

decltype גמיש יותר מ auto ולמעשה ב templates משתמשים בו. ראה דוגמא:



## Etc…

### בידוד הקוד מהקבצים שמפיק הקומפיילר

1. לשנות את Project prop>config prop>general>intermediate dir.
2. בהגדרות של הסטודיו עצמו צריך לומר לו איפוא לשים את קבצי ה IntelliSense. Tools > Options > Text Editor > C/C++ > Advanced. לקבוע שם את **Fallback Location** ולומר לו להשתמש בו**. (**את זה עושים כמובן פעם אחת בלבד.)
3. לקבוע את קובץ ה sln במקום נפרד (שים לב שה filters צריכים להיות ליד ה vcxproj) פשוט להעתיק אותו לשם ואחר כך לטעון מחדש את הפרויקט האבוד.

### **Assertions**

**ASSERT(pointer);**

יש גרסא סטנדרטית וגרסא מיקרוסופטית ברמת ה CRT

\_ASSERTE מוגדר ב crtdbg.h

הגרסא הסטטית מתגלה בשלב הקומפילציה בלבד

static\_assert(sizeof(float) == 4, "I can’t float like that!");

והיא יכולה להופיע שלא בתוך מתודה.

קני קאר הגדיר Verify

#ifdef \_DEBUG

#define VERIFY ASSERT

#else

#define VERIFY(expression) (expression)

#endif

כלומר בגרסת דיבאג זה אסרט, אבל בגרסת ריצה זה לא 'נעלם' (כפי שאסרט נעלם) אלא משאיר את הביטוי הפנימי. מאד שימושי. למשל כאן בדיסטרקטור.

~ManualResetEvent()

{

VERIFY(CloseHandle(m\_handle));

}

קני קאר אוהב להשתמש בקוד הסטנדרטי הבא

#include <crtdbg.h>

#include <windows.h>

#define ASSERT \_ASSERTE

#ifdef \_DEBUG

#define VERIFY ASSERT

#else

#define VERIFY(expression) (expression)

#endif

### Emit Debug messages

הקוד הנחמד הזה ידפיס אל ה output של הסטודיו בזמן דיבאג

wchar\_t msgbuf[256];

swprintf\_s(msgbuf, 20, L"My variable is %d\n", 10);

OutputDebugString(msgbuf);

ואל DebugView בזמן ריצה חופשית.

### Named Casts

static\_cast

זהו הפשוט ביותר. כרגיל, אם משתמשים בקאסט אשר כרוך באיבוד מידע הקומפיילר מזהיר. זו הדרך **למנוע** אזהרה זו.const\_cast

סכנה. מאפשר לעקוף הצהרה על מצביע ל'קבוע' .

reinterpret\_cast

המרה של מצביע לטיפוס אחד ממצביע של טיפוס אחר. המתכנת מצהיר שהוא יודע מה הוא עושה (בדרך כלל כוונתו שבאמת המצביע של הטיפוס האחר היה במקורו המצביע של הטיפוס האחד)

Dynamic\_cast

משתמש במנגנון RTTI להבטיח שההמרה של קלאסים אכן תיתכן (יש יחס אב בן)



### Time measuring in windows

*PS: C++ Advanced Topics; Use Standard Containers : Demo: vector and list*



## MEMBER FUNCTION POINTER

*https://isocpp.org/wiki/faq/pointers-to-members*

אם רוצים מצביע למתודה של קלאס,

יש שתי דרכים עקריות, א) פשוט להפוך את המתודה לסטטית, ואז this לא נכנס לרשימת הארגומנטים ולא "מקלקל" את החותמת של המתודה ב) להשתמש בפונקציה מתווכת כמו בדוגמא המצורפת

=> יש גם דרכים אחרות. חפש ברשת.



## Smart Pointers : Demo: unique\_ptr

*PS: Modern C++ Libraries, Kenny Kerr*





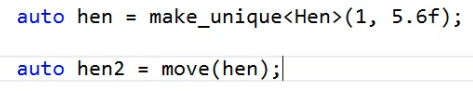
שתי ההצהרות הנ"ל זהות.



זו יצירה ואתחול.



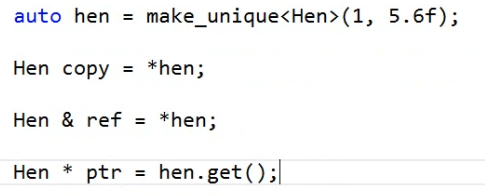
וזו הדרך המועדפת על כולם.



לא ניתן להעתיק אבל ניתן להעביר בעלות. לאחר ה move, המשתנה hen לא יצביע יותר על המשאב אליו הוא הוקצה.



פירוש שאלה זו: האם לhen יש בעלות על משאב?



**\*hen** refers to the object itself, and hence copy=\*hen is copy constructed.

**hen.get()** returns pointer to the object itself. This breaks the uniqness!

~~כל הפעולות הנ"ל הן חוקיות ואינן סותרות את עצם זה ש hen הוא הבעלים היחידי. הן פשוט מאפשרות גישה ישירה לאובייקט.~~



שחרור בעלות וקבלת מצביע לאובייקט. השחרור לא אומר שה d-tor של האובייקט עצמו יקרא. זה רק אומר שהאובייקט כבר לא בבעלות של ה uniqe\_ptr



לעומת זאת reset כן מוחק את האובייקט.



ואפשר באותה פעולה למחוק את הקודם ולהגדיר חדש. (במקרה זה, הפעולה פשוט מחזירה חזרה את הבעלות.)

### passing unique\_ptr as formal argument

כאשר מעבירים unique\_ptr אל פונקציה, יש להשתמש באופרטור move שהרי לא ניתן לשכפל unique\_ptr

שים לב שאחרי שהוא עבר, הוא כבר לא חוקי בפונקציה הקוראת, ואם רוצים אותו בחזרה הפונקציה הנקראת צריכה להשתמש שוב ב move.

unique\_ptr<int> WorkOnRes(unique\_ptr<int> r) {

// work on r...

return move(r);

}

int main()

{

auto r = unique\_ptr<int>(new int {10});

unique\_ptr<int> r2 = WorkOnRes( move(r));

}

דרך פשוטה יותר היא להשתמש ב const l-value ref

WorkOnRes(unique\_ptr<int> & r) {

## shared\_ptr

*Modern C++ Libraries, Smart Pointers : shared\_ptr*

בניגוד ל COM, כאן האובייקט אינו מודע לכך שהוא תחת ref counting. קוראים לזה non-invasive ref count כלומר לא פולשני. בעוד שב COM המונח הוא intrusive. המשמעות היא שצריך להקצות אובייקט מיוחד שחי מחוץ לאובייקט העבודה עבור מניית התיחסויות. ועל כן רצוי מאד להשתמש ב make\_shared.

יש ל shared\_ptr move-ctor, וזה כמובן נותן יעילות במקום שהגדלת ה ref-count היא זמנית ואין בה צורך אמיתי. אמנם אם יש קונטיינר שלא מספק התיחסות ל move-ctor אז אין אפשרות אחרת אלא להשתמש ב copy-ctor (אגב, קונטיינר כזה לא יצלח לשימוש עם unique\_ptr ורק shared יתאים)

auto sp = shared\_ptr<int>{}; // null pointer. doesn’t has an object

sp. reset(new int {123}); // ok, sp now has meaning

auto sp = shared\_ptr<int>{new int(10)}; // another way to init.

auto sp = make\_shared<int>{new int(10)}; // preferred way to init/

auto sp = make\_shared<int>(10); // short way

sp.use\_count();

assert(sp); // use boolean operator

## Heap Corruptions : Heap Segments

Ref: Advanced Windows Debugging - Part 1

heap block

heap\_segment struct

dt command

סגמנטים מכילים heap blcoks באשר אותם בלוקים הם המוחזרים למתכנת לפי בקשתו. בלוקים אלה מרופדים בתחילה ובסוף, ריפודים אלה עוזרים למנהל של ה heap בניהול הבלוקים ובגילוי גלישה מתחום הבלוק.

Low Fragmentation Heap

תת סגמנטים כאשר בכל בכל תת סגמנט יש בלוקים באותו גודל. זה קצת עוזר למניעת פרגמנטציה.

Application Verifier של מיקרוסופט מאד עוזר בדיבוג של בעיות מהסוג הזה.

הוא מכניס בתים בתצורה ידועה לפי ואחרי הבלוק שהוקצה ועוקב אחריהם. כאשר יש דריסה הוא גורם לדיבאגר לעצור, כך שהמתכנת יכול לראות בדיוק מתי נעשתה הדריסה.

youtube: Debugging Windows Memory Corruption issues using Page Heap

(כנראה

