C++

שמירה אחרונה: ‏31 ינואר 19. שעה: ‏10:52:00

Contents

[Labmda expressions 2](#_Toc533703711)

[Lambda capture clause 2](#_Toc533703712)

[COPY C-TOR & REF TYPES 3](#_Toc533703713)

[Move c-tor 4](#_Toc533703714)

[Explicit ctor 5](#_Toc533703715)

[uninform initialization 5](#_Toc533703716)

[C++ methods given free by the compiler (Meyers, item 5) 6](#_Toc533703717)

[Explicitly disallow the use of compiler-generated functions (Meyers, Item 6) 6](#_Toc533703718)

[Templates 6](#_Toc533703719)

[Using an existing template 6](#_Toc533703720)

[defining function template 7](#_Toc533703721)

[class templates 7](#_Toc533703722)

[Template specialization 7](#_Toc533703723)

[Standard containters 9](#_Toc533703724)

[vector 9](#_Toc533703725)

[List 9](#_Toc533703726)

[map 9](#_Toc533703727)

[איטרציה על איברים של קונטיינר 10](#_Toc533703728)

[STL Algorithm 10](#_Toc533703729)

[headers you should know 10](#_Toc533703730)

[count; count\_if; for( iterator); all\_of; any\_of; none\_of 10](#_Toc533703731)

[find, find\_if 12](#_Toc533703732)

[sort 14](#_Toc533703733)

[C++11 Language Features 16](#_Toc533703734)

[auto 16](#_Toc533703735)

[decltype 17](#_Toc533703736)

[Etc… 17](#_Toc533703737)

[בידוד הקוד מהקבצים שמפיק הקומפיילר 17](#_Toc533703738)

[Assertions 17](#_Toc533703739)

[Emit Debug messages 18](#_Toc533703740)

[Named Casts 18](#_Toc533703741)

[Time measuring in windows 20](#_Toc533703742)

[MEMBER FUNCTION POINTER 20](#_Toc533703743)

[Smart Pointers : Demo: unique\_ptr 21](#_Toc533703744)

[passing unique\_ptr as formal argument 22](#_Toc533703745)

[shared\_ptr 22](#_Toc533703746)

[Heap Corruptions : Heap Segments 22](#_Toc533703747)

## 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

|  |  |
| --- | --- |
| D:\GitHub\SharedDocs\NewCpp\CtorsAndRefVars\CtorsAndRefVars.cpp  ResClass \_resclass("GlobalResClass");  ResClass & getClassByRef()  {  return \_resclass;  }  ResClass getClassByVal()  {  return \_resclass;  }  int main()  {  ResClass & rc1 = getClassByRef();  ResClass & rc2 = getClassByVal();  ResClass rc3 = getClassByRef();  ResClass rc4 = getClassByVal(); | |
| קונסט' העתקה לא הופעל כלל. מופע אחד בלבד. | ResClass & rc1 = getClassByRef(); |
| קונסט' העתקה הופעל.  בעת החזרת הערך, הקומפיילר יצר מופע נוסף של RefClass אולם יש לנו רק ref אליו. מופע זה נחשב כ rvalue..  זו פעולה **לא** תקינה.  הקומפיילר (warning level = 4) נותן שתי אזהרות:  conversion from 'ResClass' to 'ResClass & '  וכן  A non-const reference may only be bound to an lvalue  שתי אזהרות אלה נעלמות אם מוסיפים const  בשורה תחתונה זה 'עובד'. | ResClass & rc2 = getClassByVal(); |
| קונסט' העתקה הופעל. מלבד המופע הגלובלי, הרי rc3 הוא גם מופע בפני עצמו. | ResClass rc3 = getClassByRef(); |
| קונסט' העתקה הופעל וצריך לומר שזה אותה פעולה כמו rc3 למרות השוני בניהם (לכאורה יש כאן אופטימיזציה מסוימת - הקומפיילר מבין שאין טעם לייצר מופע 'מיותר) | ResClass rc4 = getClassByVal(); |
|  |  |

שאלה: כאשר מופעל קונסט' העתקה על קלאס היורש מקלאס אב, איזה קונסט' יופעל בקלאס האב? הרגיל או קונסט' העתקה.

תשובה: אין כאן פעולה סמויה\מובנית, זה תלוי באופן בו מתייחס קלאס הבן לקלאס האב. פירוש, אם במימוש של קונסט' הבן יש קריאה מפורשת לקונסט' אב, וחייבת להיות קריאה כזו אם יש לקונסט' אב ארגומנטים, אז זו קריאה מפורשת אז אותו קונסט' שהבן קרא יופעל. אם אין קריאה מפורשת אז בוודאי שהקונסט' הרגיל יופעל שהרי רק הוא ללא ארגומנטים.

### Move c-tor

תכלית עניין זה הוא למנוע העתקות מיותרות של אובייקטים.

עצם העניין מוכר היטב לכל מתכנת, כאשר מכניסים אובייקט אל רשימה או שמעבירים אובייקט כארגומנט אל פונקציה בדרך כלל אין צורך לשכפל ולעיתים יש להימנע מלשכפל ועל כן מעבירים מצביעים ולא את האובייקט עצמו. אולם זה מצריך תחזוקה נוספת וכמובן שמירה מפני זליגת זכרון. מנגנון חדש זה של C++ מאפשר למנוע העתקה למרות שמעבירים את האובייקט עצמו.

התכנת נדרש להגדיר עבור האובייקט שצפוי להיות מועתק\מועבר קונסט' נוסף עם כתיב יחודי, ובמקביל גם להעמיס את ה assign-operator באופרטור נוסף, שוב עם כתיב יחודי. כאשר הקומפיילר מזהה אותם הוא **אוטומטית** משתמש בהם כאשר הוא מזהה אפשרות כזו.

נתבונן בקוד הבא



D:\GitHub\SharedDocs\NewCpp\MoveSemantics\MoveSemantics.cpp

הבלוק הראשון הוא 'רגיל'. בשורה 26 יופעל קונסט' רגיל ובשניה יופעל קונסט' העתקה. בזה אין חידוש.

אולם הבלוק השני מתנהג אחרת. הקומפיילר מזהה שהאובייקט המוכנס לווקטור הוא 'בר חלוף', הוא rvalue. בוודאי שאין להשתמש בו לאחר שעוברים את שורה 31. ועל כן הקומפיילר מחפש ומוצא שיש לו קונסט' העברה **ומשתמש בו**. קונסט' העברה יופעל **במקום** קונסט' העתקה.

זה **לא אומר** שלא ייווצר אובייקט נוסף. גם בבלוק א וגם בבלוק ב יש שני אובייקטים. אולם התכנת יממש את קונסט' העברה באופן שהמשאבים שמחזיק האובייקט יועברו ולא יועתקו.

זה קונסט' העתקה. שדה name **מועתק**:



זה קונסט' העברה. שדה name **מועבר**:

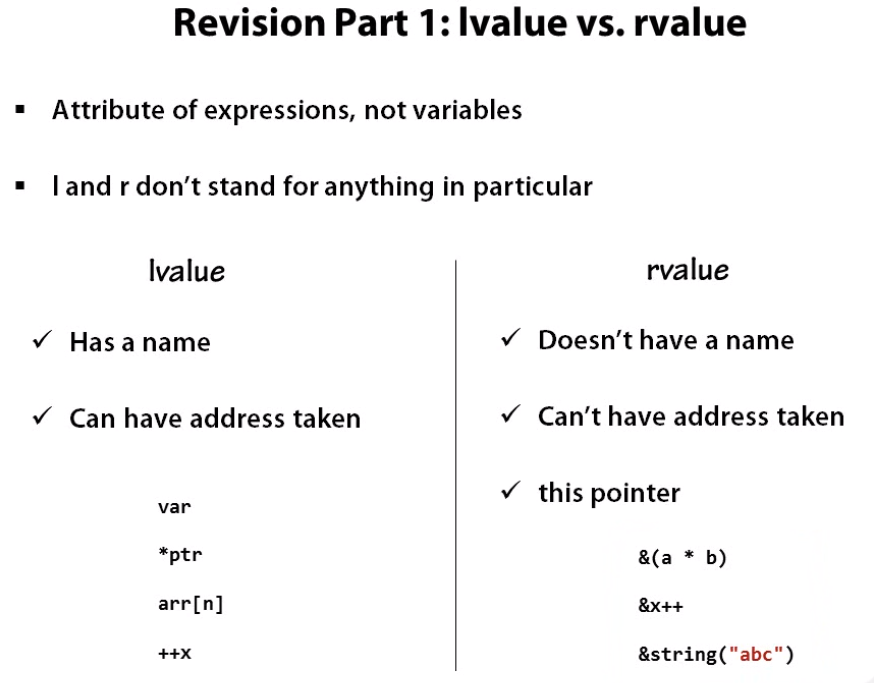


שים לב שכאן אין const וזה בכוונה, שהרי המתכנת משפיע על אובייקט המקור כשהוא 'לוקח' ממנו את המשאבים שלו.

פונקציה שרוצה להחזיר אובייקט שהוגדר על המחסנית שלה זה מעשים שבכל יום. עד עכשיו היה צריך להעתיק את האובייקט בשלב ההחזרה. כיום, אם יש לאותו אובייקט move c’tor אז היא פשוט מחזירה אותו כמות שהוא. יתרון משמעותי.

### Move constructor

PS-Korban-C++11



### 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};

### C++ methods given free by the compiler (Meyers, item 5)

הקומפיילר מייצר "בחינם" את ה:

* Default c’tor
* Copy c’tor
* Copy assign operator
* d’tor

(אגב, הקומפיילר מייצר אותם רק במידה ואכן משתמשים בהם)

אבל מה קורה אם אחד מחברי הקלאס הוא ref או const כמו בדוגמא הבאה



את השדות הנ"ל אפשר רק לאתחל בכותרת של הקונסט', לא ניתן להציב להם ערך חדש.

במקרה כזה הקומפיילר לא יכול לייצר בעצמו assign oper. ותתקבל שגיאת קומפילציה.

### Explicitly disallow the use of compiler-generated functions (Meyers, Item 6)

יצירת קלאס שלא ניתן להעתקה\שכפול מחייב מניעה אקטיבית של copy-const’, assign oper.

הדרך הפשוטה היא לקבוע אותם בתור Private

אולם זה עדיין לא מספיק, משום שמתכנת מתוחכם יכול לקרוא להם דרך מתודה אחרת או שאולי דרך friend class.

כנגד זה צריך להימנע מלתת להם מימוש. (אמנם במקרה כזה מי שינסה להשתמש בהם יראה בעיה ברמת ה linker).

### Virtual d'tor (Meyers, Item 7)

כלל ידוע בבניית קלאסים שעתידים להיות להם יורשים לקבוע את ה דיסטר' כווירטואלי, משום שאחרת, מצביע מטיפוס קלאס אב, יפעיל את הדיסט' של קלאס האב (למשל כשהוא יוצא מסקופ) למרות שבפועל הוא מצביע למופע של בן.

אין לעשות כן בקלאסים שלא צפויים להיות להם יורשים, משום שזה מעמיס את הישום של הקלאס ללא צורך. קלאס עם מתודה וירטואלית מקבל שדה (נסתר) בשם vptr אשר מצביע לטבלה וירטואלית vtable. זה גם מכביד וגם מונע את האפשרות להעביר קלאס זה לשפת C.

אם רוצים להגדיר קלאס כאבסטרקטי אבל אין מתודה ספציפית לצורך כך, הבחירה הטבעית היא הדיסט'.

### Don't allow destructors to throw exceptions (Meyers, item 8)

אם ווקטור מחזיק אובייקטים וכשהוא יוצא מסקופ הוא מוחק אותם אחד אחד, הרי אם אחד מהם יזרוק 'חריג' זה יקלקל את הפעולה כולם.

אמנם 'בליעת חריגים' זה הרגל רע אולם במקרה זה אין ברירה אחרת, וכמובן שצריך להוציא הודעת לוג מתאימה.

## 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.

שים לב לשימוש ב decltype זו מעין העברה של הטיפוסים אל צד שלישי. אמנם היה אפשר להשתמש בשורה שבהערה, אולם יש מקומות שזה נדרש.

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.

### Template specialization

תבנית באה ליצור כלל, דהיינו קלאס עם טיפוסים מידע כלליים. לעיתים נדרש יוצא מן הכלל, לדוגמא קלאס אשר מטפל בכל סוגי המשתנים המספריים, מטפל אחרת במחרוזות.

הקלאס הכללי:

template <typename T>

class Number

{

Number(T t){}

}

והקלאס הספציפי

Template<>

Class Number<std::string>

{

Number(std::string s){}

}

כאשר הקומפיילר יראה שכתוב Number<std::string> הוא ידע להתייחס לקלאס הספציפי ולא לכללי.

### Special usage templates

std::is\_floating\_point<T>::value

מחזיר true או false. מאד שימושי כאשר בתוך תבנית רוצים לשנות את המהלך לפי הטיפוס.

using xType = typename std::conditional<bool, TypeA, TypeB>::type

הגדרת טיפוס לפי ערך בוליאני. גם כאן שימושי בתוך תבניות. הערך הבוליאני חייב להיות **ידוע** בזמן קומפילציה.

נאמר שרוצים סט פונקציות שיתנו טיפול שונה לכל טיפוס מידע long, int, char.

מגדירים מספר פונקציות ומכניסים ארגומנט נוסף שכל תפקידו הוא לעזור בבחירת ה template המתאים.

template< typename T > string FormatNumber(T number, CLong) {}

template< typename T > string FormatNumber(T number, CInt){}

template< typename T > string FormatNumber(T number, CShort){}

template<typename T>

string FormatNumber(T number)

{

If (sizeof(T)==8)

FormartNumber(number, CLong());

If (sizeof(T)==4)

FormartNumber(number, CInt());

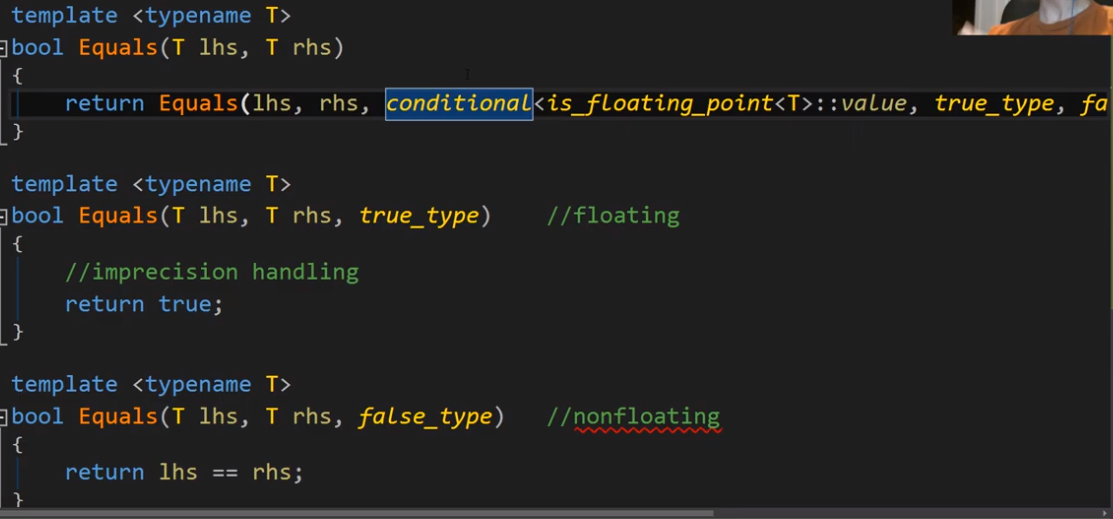
If (sizeof(T)==1)

FormartNumber(number, CChar());

}

יש דרך יותר נקיה ויפה. אם ההחלטה היא בוליאנית, נאמר שהשאלה היא האם מדובר ב float או ב int. אז אפשר להיעזר ב std::conditional אשר מצידו יקרא ל std::is\_floating\_point

זה נראה כך (הוא תיקן אחר כך ל conditionanl**\_t** לחילופין אפשר להוסיף typename לפני)



[ראה](https://www.youtube.com/watch?v=Vkck4EU2lOU&t=870s).

או חפש ביוטיוב: C++ Templates - Part 2: Basic syntax, specialization, tag dispatch and SFINAE

מימוש לדוגמא בפרויקט learnTemplates

### 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 קונסט' העתקה.

אם לא רוצים לשכפל, אפשר להשתמש במצביע ורצוי במצביע אוטומטי כמו unique\_ptr:

vector<unique\_ptr<Resource>> vp1;

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

חסרון משמעותי של ווקטור שאם נדרש להוסיף לו איברים הוא צריך לבנות את עצמו מחדש. לדוגמא אם יש לו 2 איברים ורוצים להוסיף שלישי. הוא יבנה את עצמו מחדש בתור ווקטור עם ארבעה איברים ויעתיק את שני האיברים הקיימים מהישן אל החדש. כמובן שקונסט' העתקה יופעלו כאן ובאחריות המתכנת לדאוג לבנות קונסט' העתקה במידת הצורך.

### 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.

### map

זוגות של ערכי key-value ניתן לגשת לערך לפי המפתח. הרשימה מסודרת.

מתאים ל lookup-table.

### איטרציה על איברים של קונטיינר

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:



### `const` before or after

**X const& x**

is equivalent to

**const X& x**

and

**X const\* x**

is equivalent to

**const X\* x.**

### Mutable

מאפשר למתודה המוגדרת כ const בכל זאת לשנות ערכים של קלאס – אותם אלה המוגדרים כ mutable. הרעיון כאן להפריד בין יצוג חיצוני של הקלאס שבו המתודה לא פוגעת לבין פרטי מימוש פנימיים שאותם המתודה כן רשאית לשנות.

## 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));

}

[אבל מצד שני, זה מקשה על קריאת הקוד, לא בטוח שכדאי...]

### 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

Performance.

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





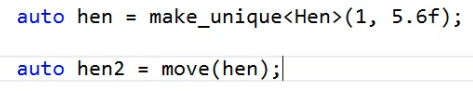
שתי ההצהרות הנ"ל זהות.



זו יצירה ואתחול.



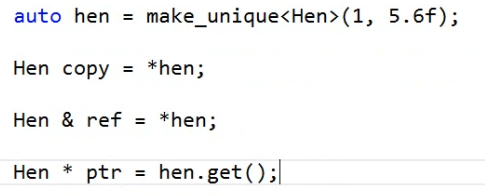
וזו הדרך המועדפת על כולם.



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

