C++

שמירה אחרונה: ‏יום שלישי 19 נובמבר 2019 ‏15:08:00

Contents

[Labmda expressions 2](#_Toc7956899)

[Using Lambda 2](#_Toc7956900)

[Lambda capture 3](#_Toc7956901)

[rval-lval, Move semantics, perfect forwaring 3](#_Toc7956902)

[lvalue-rvalue 3](#_Toc7956903)

[Move c-tor 4](#_Toc7956904)

[Perfect forwarding 5](#_Toc7956905)

[std::move vs std::forward 7](#_Toc7956906)

[COPY C-TOR & REF TYPES 8](#_Toc7956907)

[Explicit ctor 9](#_Toc7956908)

[uninform initialization 9](#_Toc7956909)

[C++ methods given free by the compiler (Meyers, item 5) 9](#_Toc7956910)

[Explicitly disallow the use of compiler-generated functions (Meyers, Item 6) 10](#_Toc7956911)

[Virtual d'tor (Meyers, Item 7) 10](#_Toc7956912)

[Don't allow destructors to throw exceptions (Meyers, item 8) 10](#_Toc7956913)

[Templates 10](#_Toc7956914)

[Using an existing template 10](#_Toc7956915)

[Function templates 11](#_Toc7956916)

[Class templates 11](#_Toc7956917)

[Template specialization 11](#_Toc7956918)

[Special usage templates 12](#_Toc7956919)

[typename keyword 13](#_Toc7956920)

[Containters 14](#_Toc7956921)

[Standard containters 14](#_Toc7956922)

[vector 14](#_Toc7956923)

[List 15](#_Toc7956924)

[Map 15](#_Toc7956925)

[Range based `for` 16](#_Toc7956926)

[Headers you should know 16](#_Toc7956927)

[count; count\_if; for( iterator); all\_of; any\_of; none\_of 17](#_Toc7956928)

[find, find\_if 18](#_Toc7956929)

[sort 20](#_Toc7956930)

[C++11 Language Features 22](#_Toc7956931)

[auto, decltype 22](#_Toc7956932)

[`const` before or after 24](#_Toc7956933)

[Mutable 24](#_Toc7956934)

[Scoped enum/ enum-class 24](#_Toc7956935)

[static\_assert 25](#_Toc7956936)

[Etc… 26](#_Toc7956937)

[בידוד הקוד מהקבצים שמפיק הקומפיילר 26](#_Toc7956938)

[Assertions 26](#_Toc7956939)

[SEH exceptions 27](#_Toc7956940)

[Emit Debug messages 27](#_Toc7956941)

[Named Casts 27](#_Toc7956942)

[Time measuring in windows 29](#_Toc7956943)

[MEMBER FUNCTION POINTER 29](#_Toc7956944)

[Heap Corruptions: Heap Segments 30](#_Toc7956945)

[Smart Pointers 30](#_Toc7956946)

[unique\_ptr 30](#_Toc7956947)

[passing unique\_ptr as formal argument 31](#_Toc7956948)

[shared\_ptr 32](#_Toc7956949)

[Solid Principles/ Patterns 32](#_Toc7956950)

[OCP – Open Close Principle 32](#_Toc7956951)

## Labmda expressions

### Using Lambda

צורה כללית של ביטוי למדא

auto l = [](int a, int b) -> int {return (a+b)};

הכתיב " -> int " נקרא trailing return type והוא לפי הסטנדרט החדש של השפה. אפשר לוותר על זה במידה והקומפיילר יכול להסיק זאת בעצמו.

במידה והביטוי לא תופס ערכים מהסקופ שמסביב אפשר שהטיפוס של l יהיה מצביע לפונקציה (מתאימה) והקומפיילר יבצע את ההמרה. אם הוא כן תופס ערכים, אז הדרך היחידה היא auto ואז כדי להעביר אותו לפונקציה צריך תבנית משום שאי אפשר שארגומנט של פונקציה יהיה auto.

לא ניתן בביטוי הנ"ל לשנות בגוף הלמדא את a,b משום שהועברו לפי ערך (בסתם פונקציה אין הגבלה כזו, כידוע). אם רוצים בכל זאת, אפשר לציין mutable לפני הסוגריים המסולסלים.

### Lambda capture



## rval-lval, Move semantics, perfect forwaring

### lvalue-rvalue

PS-Korban-C++11

rvalue הוא ערך "בר חלוף", 'משתנה' שניתן "לגנוב" את תוכנו מבלי שיהיה אכפת למישהו.

*גונן נתן את ההגדרה הבאה:*

*ל lvalue יש שם, יש כתובת, ניתן לבצע אליו השמה, והוא 'ממשיך' מעבר לשורה בה הוא הוגדר. כל השאר הם rvalue.*

אם, לדוגמא, מעבירים אובייקט אל פונקציה בצורה הבאה

func(MyClass());

אז אומרים שעבר אל הפונקציה משהו שנקרא rvalue. משום ששורה לאחר מכן, אותו מופע של MyClass כבר לא מעניין.

אחת הדרכים לזהות rvalue הינה שאין לו שם ושלא ניתן לזהות את כתובתו.

דוגמא, אין לביטוי (a+b) שם, הפעולה &(a+b) אינה חוקית כך שאין לו כתובת.

פונקציה יכולה לסמן במפורש שהיא מקבלת rvalue

void func(Widget&& w);

פירוש הדבר שכאשר יועבר אובייקט לפונקציה זו ויש לו move ctor הקומפיילר יאתחל אותו באמצעות move ctor ולא copy ctor.

קוראים ל w : rvalue-ref.

הערה: w לכשעצמו הינו lvalue. (מיירס, בהקדמה לפרק 5).

מקובל לחלק rvalue לסוגים, אלה העיקריים:

* **Prvalue** – pure r-value
* **Xvalue** – expiring value

ליטרל הוא pure rvalue, וכן פונקציה שמחזירה ליטרל וכן אובייקט ללא שם כגון.pushback(MyClass(10))

אפשר להשתמש באופרטור std::move() כדי לומר לקומפיילר במפורש שמדובר ב xvalue והוא יכול "לגנוב" את המשאבים שלו והאחריות על המתכנת.

שתי השורות הבאות שקולות בקירוב.

A a1 = (A&&)a2;

A a1 = std::move(a2);

לא תמיד rvalue מבטיח שימוש ב move-ctor. אם לדוגמא ה ctor מכיל const אשר מבטיח שהאובייקט הנכנס לא ישתנה, לא ניתן לבצע move שהרי move בוודאי **כן** משנה את האובייקט הנכנס (בדרך כלל הוא קובע את המצביע למשאב ל nullptr ויש דברים נוספים). במקרה כזה יקרא ה copy-ctor.

### Move c-tor

תכלית עניין זה הוא למנוע העתקות מיותרות של אובייקטים.

עצם העניין מוכר היטב לכל מתכנת, כאשר מכניסים אובייקט אל רשימה בדרך כלל אין צורך לשכפל ולעיתים יש להימנע מלשכפל והפתרון הפשוט הוא להעביר מצביע ולא את האובייקט עצמו. אולם זה מצריך תחזוקה נוספת וכמובן שמירה מפני זליגת זיכרון.

מנגנון חדש זה של C++ נועד למנוע שכפול משאבים כשזה לא נדרש למרות שמעבירים את האובייקט עצמו.

לשם כך התכנת נדרש להגדיר עבור האובייקט שצפוי להיות מועתק\מועבר שתי מתודות

Move c-tor

Move assignment operator

כאשר הקומפיילר מזהה שיש את המתודות האלה הוא **אוטומטית** משתמש בהם כאשר יש לו אפשרות לכך.

נתבונן בקוד הבא

ResClass CreateResource()

{

ResClass rc("rc2");

return rc;

}

int main()

{

ResClass rc = CreateResource();

לפי מה שהיינו רגילים בעבר, יווצרו כאן שני אובייקטים כאשר השני יווצר על ידי העתקה.

במקרה זה, מאחר שמראש ResClass הוגדר עם move-ctor, והקומפיילר מזהה ש CreateResource מחזיר r-value האובייקט השני יווצר על ידי **העברה**, כלומר יופעל move-ctor ושם המתכנת כבר ידאג לכך שהמשאב אותו מחזיק ResClass לא יועתק אלא יועבר.

דוגמא נוספת:



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. הקומפיילר יצור אובייקט **חדש** אשר יוכנס אל הווקטור, אבל במקום לקרוא ל copy-ctor הוא יקרא ל move-ctor והתכנת נדרש למממש שם קוד אשר **יעביר** את המשאבים שמחזיק הקלאס במקום לשכפל אותם.

אופרטור בשם std::move יכול לעזור למתכנת להבהיר את כוונתו. אופרטור זה מביא לכך שהארגומנט שלו יוצג בתור rvalue, הקומפיילר מבין את הרמז ומפעיל את ה move-ctor

זה קונסט' העתקה. שדה name **מועתק**:



זה קונסט' העברה. שדה name **מועבר**:



הערה: אין const באופרטור העברה. המתכנת הרי משנה את אובייקט המקור כשהוא 'לוקח' ממנו את המשאבים שלו. (זה לא שאי אפשר לכתוב שם const אלא שאם נכתוב פשוט לא יופעל move ctor אלא copy ctor. מיירס מעיר על כך ב Item 23)

### Perfect forwarding

#### הצגת הבעיה

הקונטיינר vector מכיל, במקביל ל push\_back, מתודה הנקראת emplace\_back.

1 vector<Class1> v;

2 v.push\_back(Class1(2, 3.14f));

3 v.emplace\_back(2, 3.14f));

בשורה 2 נעשות שלוש פעולות

1. c-tor
2. move or copy ctor
3. d-tor.

די מיותר.

לעומת זאת בשורה 3 יש מעין קסם, האובייקט נוצר ישירות בתוך הקונטיינר. פעולה אחת בלבד – c-tor

איך נעשה הקסם הזה?

הרי emplace\_back צריך בסופו של יום לקרוא ל c-tor של Class1 ולהעביר אליו את הארגומנטים "כמות שהם", כלומר

א) מבלי לשכפל אותם

ב) לשמר את תכונת ה rvalue/lvalue שלהם.

לא פשוט.

לכאורה אפשר להעביר ארגומנט לפי ref

f(int & n)

אבל אז לא ניתן לתת rvalue.

בשיפוץ קל כן אפשר לתת rvalue:

f(const int & n)

אבל זה מונע שינוי של הארגומנט ואי אפשר לאלץ את Class1 לקבל דווקא const.

כך שאת הפתרון צריך לחפש במקום אחר.

#### Reference collapsing

שים לב לקוד הבא

template <typename T>

void baz(T t) {

T& k = t;

}

int i = 4;

baz<int&>(i);

צורת הכתיבה מכריחה את T להיות int&, אם כן השורה שמאתחלת את k מתפרשת כך

int& & k = t

כלומר lval-ref to lval-ref

אין לזה משמעות C++.

בעבר הקומפיילרים היו פותרים את זה בדרכים יחודיות. כיום זה סטנדרט וזה נקרא reference collapsing.

הכלל הוא פשוט:

rval-ref to rval-ref => rval-ref

all other combinations => lval-ref

ואם כן

int& & k = t => int& k = t

נשתמש בכללים אלה בהמשך הדיון.

#### הקסם

נאמר שהפונקציה wrapper רוצה להעביר את הארגומנטים שלה כמות שהם אל צד ג'.

יש כתיב יחודי למטרה זו והוא

template <typename T>

void wrapper(**T&&** t1)

{ }

הטיפוס T מתנהג בצורה לא שגרתית (=קסם)

**יתכן** שהוא יהיה lvalue-ref **ויתכן** שהוא יהיה סתם lvalue תלוי איך קוראים לו.

נתבונן בקוד הבא:

1 int x=10;

2 wrapper(x); // *=> T is int&, wrapper(int& && t1) => wrapper(int& t1)*

3 wrapper(10); // *=> T is int, wrapper(int && t1)*

בשתי השורות התבנית אותחלה עם int אולם בשורה הראשונה זה היה lval ובשני rval. הקומפיילר קבע את T לפי תכונה זו.

מעין התנהגות פולימורפית, אולם מטרה א' הושגה - כאשר x עבר לא היה שכפול וכאשר 10 עבר אז מעיקר הדין אין שכפול.

נמצא שכאשר נכנס lval (או גם lval-ref) אז t1 הינו lval-ref

וכאשר נכנס rval אז t1 הינו lval (כזכור, בתוך גוף הפונקציה, rval-ref הינו lval)

ועדיין צריך להגיע למטרה ב' (לעיל)

את זה עושה forward שעליו פירוט בסעיף הבא, אולם בקצרה, אופרטור זה ממיר את המשתנה הנכנס עליו כך ש lval הופך להיות rval-ref ואילו lval-ref נשאר כמות שהוא.

#### std::forward<>()

זהו המימוש של forward באופן סכימטי.

template<typename T>

T&& forward(typename remove\_reference<T>::type& param)

{

return static\_cast<T&&>(param);

}

וכך נעשית הקריאה אליו

template <typename T>

void wrapper(**T&&** t1)

{

v.AddItem(forward<T>( t1));

}

נניח שקוראים ל forward עם lval-ref

אז T הינו int&

ואז המימוש של forward הינו

int& && forward(typename remove\_reference<int&>::type& param)

{

return static\_cast<int& &&>(param);

}

אז

int& forward(int& param)

{

return static\_cast<int&>(param);

}

נכנס lval-ref ויצא lval-ref. האופרטור לא עשה דבר. לפי המצופה.

נניח עכשיו שקוראים למתודה עם lval אז T הינו int

המימוש של forward הינו

int && forward(typename remove\_reference<int>::type& param)

{

return static\_cast<int&&>(param);

}

אז

int&& forward(int& param)

{

return static\_cast<int&&>(param);

}

נכנס lval ויצא rval-ref. לפי המצופה.

#### דוגמא מסכמת

כך זה נראה:

struct PerfFwd

{

template<typename T>

void AddItem(T && t) {

v.emplace\_back(forward<T>(t));

}

}

יש כאן כמובן הנחה ש V הוא ווקטור של אובייקטים המקבלים ארגומנט אחד בלבד ב c-tor

במקרה הכללי, יש להשתמש ב variadic template.

struct PerfFwd

{

template<typename ... Args>

void AddItem(Args&& ... args) {

v.emplace\_back(forward<Args>(args)...);

}

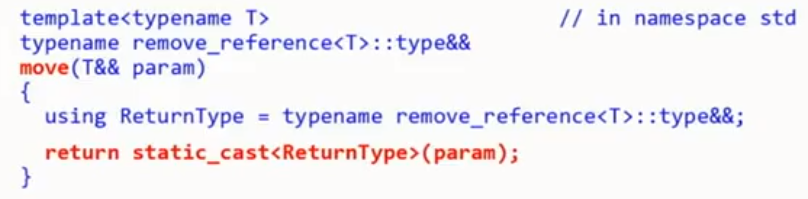
}

### std::move vs std::forward

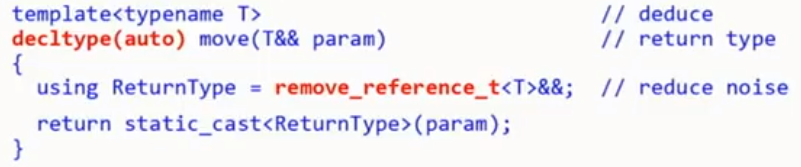
Mayers, item 23.

גם move וגם forward מבצעים בסופו של דבר static\_cast<”rvalue”>. ההבדל הוא ש move אינו מותנה והכתיב פשוט יותר בעוד ש forward מבצע את ההמרה רק בתנאי שהטיפוס המומר הינו lvalue (מה שאומר שהוא היה בעברו rvalue).

זה המימוש (הסכימתי) של move:



עם העדכונים של C++/14 אפשר לכתוב את הנ"ל בצורה יותר תמציתית. הקומפיילר מסיק בעצמו מהו הטיפוס החוזר וגם השימוש ב remove\_reference**\_t** חוסך את כתיבת typename.



מיירס כותב שאולי שם נכון יותר ל move הינו rvalue\_cast אבל בנסיבות מסוימות זה עלול לבלבל ועדיף להשאיר את המצב כמות שהוא.

## COPY C-TOR & REF TYPES

נתבונן בקוד הבא (הורץ ב studio 19)

ResourceClass GetRes() { return ResourceClass(); }

int main()

{

ResourceClass rc = GetRes();

לכאורה הקומפיילר צריך לייצר שני מופעים של המשאב. אחד רגיל והשני בהעתקה אולם לפי הסטנדרט החדש של **השפה** הקומפיילר מזהה שאין צורך בשניים ומייצר רק אחד. זה נקרא Copy Elision.

השורה הבאה בעייתית מסיבות מובנות

ResourceClass & rc = GetRes();

והקומפיילר כלל לא מסכים לקמפל (עם כי בגרסאות קודמות הוא כן הסכים ונתן למתכנת להתמודד עם תוצאות מעשיו...)

אבל כך

const ResourceClass & rc = GetRes();

או גם

ResourceClass && rc = GetRes();

הוא כן מסכים וזה פלא.

גונן הסביר שהוא מקצה מקום עבור האובייקט איפשהוא במרחב הגלובלי. אמנם במקרה של lval-ref צריך לסמן במפורש שהמתכנת לא ישנה את הערך במצב כזה ולכן const.

אמנם לא הבנתי למה הקומפיילר מאפשר לשנות את rc כאשר הוא rval-ref.

מכל מקום, כאן הקומפיילר מייצר מופע אחד בלבד.

מה אם הפונקציה עצמה מחזירה lval-ref

ResourceClass & RefGetRes() { ResourceClass rc; return rc; }

אבל התוצר הולך אל lval?

ResourceClass rc = RefGetRes();

זה ממש חסר יעילות. הקומפיילר מבצע cast מ lval-ref אל lval והתוצאה היא ייצור של שני מופעים של המשאב. אחד רגיל והשני בהעתקה.

אגב, השורה הבאה

(במקומות שהקומפיילר כן מסכים לקמפל)

ResourceClass & rc = RefGetRes();

rc.x = 10;

תביא כמובן לקריסה עקב חריגת זיכרון, הרי המשאב כבר לא נמצא על המחסנית.

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.

### 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;

מובן שאם היה יותר מארגומנט אחד לא ניתן היה להשתמש בסימן '='.

השימוש ב '=' לא תמיד טוב, הקומפיילר יעשה כמיטב יכולתו להמיר טיפוסים לצורך אתחול כזה, למשל:

Resouces r = ‘a’;

וזה פתח לבאגים.

הדרך למנוע זאת היא פשוט להשתמש במילה השמורה explicit

class Resource

{

public:

**explicit** Resouces(const int size);

string GetName() const;

};

כאן, ההצהרה

Resouces r = ‘a’;

תגרום לשגיאת קומפילציה.

והדרך היחידה לאתחל אובייקט היא כך:

Resouces r(1);

באופן כזה הקומפיילר לא יסכים להמרות.

בתור ברירת מחדל, כדאי תמיד לתת **explicit**.

הערה: ניתן לתת explicit גם על אופרטורים של המרה.

### 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.

הערה: גם move ctor/ move assign oper ניתנים בחינם. זה קובע כאשר לקלאס המועבר יש שדות שהם עצמם אובייקטים בעלי move ctor.

### 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.

ועל כן, בנוסף להגדרתם כ Private צריך להימנע מלתת להם מימוש. (אמנם במקרה כזה מי שינסה להשתמש בהם יראה בעיה ברמת ה linker).

הערה: בגרסא המודרנית של השפה ניתן לעשות זאת בקלות בעזרת המילה השמורה delete.

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

כלל ידוע בבניית קלאסים שעתידים להיות להם יורשים לקבוע את ה דיסט' 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. אז אמנם יש רק טבלה וירטואלית אחת אבל שדה vptr מצטרף **לכל אובייקט** ואם יש הרבה אובייקטים זה בהחלט עלול להעמיס.

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.

### Ref types within class

ידוע ש ref types אינם עומדים בפני עצמם, דהיינו שיש לאתחל אותם מיד עם הגדרתם.

כאשר משתנה של קלאס או ref type אז אמנם אין צורך (ואי אפשר) לאתחל בשלב ההגדרה אבל חובה לאתחל ב ctor

דוגמא

struct MyClass {

MyClass(C1 & c1): c1ref(c1){}

C1 & c1ref;

}

יתרון משמעותי של זה, זו האפשרות לקבל ב ctor קלאס מופשט טהור, שהרבה פעמים מייצג ממשק, לאו דווקא בתור מצביע אלא בתור ref.

## 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.

### Class templates

דוגמא

template<typename T>

struct Point

{

Point(T x, T y) {};

T x;

T y;

};

ושימוש:

Point<int> p1(10, 11);

Point<double> p1(0.1, 0.2);

* מבחינת הקומפיילר מדובר כאן בשני קלאסים שונים - הטיפוס הוא חלק מהחותמת של הקלאס.
* הרחבת התבנית קצת דומה למה שעושה include אלא שכאן הקומפיילר מייצר את הקוד ישירות ב obj.
* נמצא אם כן שהקוד הנ"ל גרם ליצור של שני קלאסים ואם שני קבצי cpp השתמשו בתבנית עם אותו טיפוס אז לכל אחד מהם יש הרחבה משלו שנמצאת ב obj שלו. מעשית הלינקר יודע לזהות זאת ולא צועק על התנגשות בשמות. אפשר להשתמש במילת extern כדי לחסוך הרחבות כפולות.
* אם לא משתמשים בתבנית אז הקומפיילר כלל לא מייצר עבורו קוד, כלומר לא מקמפל[[1]](#footnote-1) אותו (אין שגיאות קומפילציה). כדי לאלץ קומפילציה צריך לעשות פשוט שימוש בתבנית או להשתמש בתכונה חדשה של השפה, ההצהרה : explicit Point<in> תאלץ את הקומפיילר לקמפל ולמצוא טעויות.

### Function templates



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 שנדרש לומר במפורש).

קביעת 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

בקלאס תבניתי אין להפריד את ה H מה CPP. כלומר צריך להכניס את גוף המתודות במקום הגדרתם (זה לא אילוץ של השפה, זה פשוט בא למנוע מצב שמי שעושה include יראה גם את המימוש).

### 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.

### Variadic Templates

תבנית שמקבלת מספר לא מוגדר מראש של ארגומנטים.

הדוגמא הבאה מחשבת מינימום.

template<typename T>

constexpr T minElement(T v)

{

return v;

}

template<typename T, typename... Args>

constexpr T minElement(T first, Args... args)

{

auto m = minElement(args...);

if (m < first)

return m;

else

return first;

}

* typename... Args is called a template parameter pack,
* Args... args is called a function parameter pack
* (Args is, of course, a completely arbitrary name and could be anything else)

הערות לגבי קוד זה:

1. כל הארגומנטים מאותו טיפוס [משמע שאין אפשרות אחרת]
2. הרקורסיה בדוגמא הנ"ל מתבצעת בזמן **קומפילציה**.
3. כאשר יש ארגומנט אחד בלבד, הקומפיילר יעדיף את התבנית אשר מכילה ארגומנט אחד בלבד (כלומר את הראשונה בדוגמא הנ"ל).

דרך נוספת לשימוש ב var-temp אבל ללא רקורסיה מודגם בפרויקט varTemplates.sln

### Special usage templates

בחינה האם טיפוס מסוים הוא float

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

שימושי כאשר בתוך תבנית רוצים לשנות את המהלך לפי הטיפוס.

הגדרת טיפוס לפי תנאי

using xType = typename[[2]](#footnote-2) std::conditional<bool, TypeA, TypeB>::type

xType shall be TypeA or TypeB, according to Boolean condition, the condition must be known at compile time.

הרחבה של הנ"ל - 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){}

אין קוד אופרטיבי ל CLong וחבריו. הם נועדו רק לתמוך בבחירת התבנית המתאימה

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());

}

זו דרך מיידית ופשוטה. אולם יש דרך יותר נקיה ויפה.

בהנחה שההחלטה היא בוליאנית – צריך למתג בין שתי פונקציות בלבד, אחת מטפלת ב float והשנייה בכל השאר.

האופרטור std::conditional יכול כאן לעזור.

דוגמא:

יש שתי פונקציות זהות, אחת משמשת ל int ואחת ל float (בדיקת שוויון ב float צריכה להיות קצת טולרנטית)

template<typename T>

bool IsEqual(T lhs, T rhs, true\_type) // check float

{

return true;

}

template<typename T>

bool IsEqual(T lhs, T rhs, false\_type) // check int

{

return true;

}

והפונקציה השלישית היא זו שבה משתמשים, והיא בוחרת לאיזה מהשתיים הנ"ל לפנות.

template<typename T>

bool IsEqual(T lhs, T rhs) } {

return IsEqual(lhs,rhs,**conditional\_t**<**is\_floating\_point**<T>::value,true\_type,false\_type>{});

}

according to is\_floating\_point, the operator conditional\_t shall select which type (true\_type, false\_type) will be forwarded as third argument, and the compiler shall select the appropriate template.

[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

return IsEqual(lhs, rhs, conditional\_t<is\_floating\_point<T>::value, true\_type, false\_type>{});

return IsEqual(lhs, rhs, typename conditional<is\_floating\_point<T>::value, true\_type, false\_type>::type{});

### typename keyword

<https://en.wikipedia.org/wiki/Typename>

המילה השמורה typename מוכרת בהגדרת תבניות, אולם יש לה שימוש נוסף שקצת קשור לצורה המודרנית של השפה.

הקדמה:

בקוד הבא, הטיפוס של p תלוי באינסטסיאציה של הפונקציה foo.

template <typename T>

void foo(const T& t)

}

T::bar \* p;

{

אומרים ש T::bar הוא dependent type

בהינתן

struct StructWithBarAsType

{

typedef int bar;

};

הקוד הבא, קובע את p להיות מצביע ל int

int main(){

StructWithBarAsType x;

foo(x);

}

אבל קוד זה, כמות שהוא, לא יתקמפל!

איך הקומפיילר אמור להתייחס ל \* T::bar בגוף התבנית? זו פעולת כפל? זה מצביע?

כדי להבהיר את הכוונה יש להוסיף את המילה השמורה typename.

**typename** T::bar \* p;

וזהו השימוש הנוסף שיש ל typename.

### Type deduction

template <typename T>

void Foo(T& t)

{

t = 10; // won’t compile

T x = 10;

x++; // won’t compile

}

int main()

{

int const a = 0;

Foo(a);

}

מהו הטיפוס של t בקוד הנ"ל. לכאורה זה int& , אבל לא. משום שאין לומר שבתוך Foo ניתן יהיה לשנות את t. יש שם שתי שורות שלא מתקמפלות שממחישות נקודה זו. הקומפיילר מסיק:

T = const int

ואז

typeof (t) = const int &

אם הכותרת של הפונקציה היתה כזו:

void Foo(T const & t)

אז t היה נשאר כמו קודם אבל T היה מטיפוס int.

## Containters

### 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.

אפשר להגדיר מראש את גודלו של הווקטור ולחסוך את השכפולים הנ"ל.

v.reserve(10);

### 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 לווקטור 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.

### Range based `for`

לולאה על איברים של 'אוסף' מתבצעת כך

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

for(auto it = begin(v); it!=end(v); it++)

{

cout << \*it << "\n";

}

ואפשר לעשות איטרציה בכוון ההפוך בעזרת rbegin(),rend();

או cbegin(),cend(); כאשר c עומד עבור const. זה מכריח שימוש רק במתודות המסומנות const

בשפה התחדש כתיב מקוצר (קצת דומה ל foreach)

for (auto a : v)

{

std::cout << a.GetId() << "\n";

}

הפריסה של הצהרה כזו הינה כדלקמן:



C++11 Language Features/ Korban.

(אם יש תמיכה ב begin(seq) אז נבחרת האפשרות השניה)

אחת המשמעויות של שימוש בכתיב זה, הינה שיש בהכרח העתקה של האיבר הנוכחי אל משתנה חיצוני, כלומר מופעל copy-ctor. אם רוצים למנוע זאת אפשר להשתמש ב &.

אפשר להשתמש בכתיב זה גם למערכים **רגילים[[3]](#footnote-3)** ובאמת זה שקול (בערך) לכתיב:

for (auto a = begin(arr); a != end(arr); a++) {}

STL Algorithm

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

כתיב חדש, קצת דומה ל for-each

for (int i: v){}

for (int& i: v){}

for (auto& i: v){}

for(const auto& i: v){}

### 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;

//method 1

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

{

if (v[i] == target)

{

twos++;

}

}

// method 2

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

// method 3

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, decltype

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.



זה שימושי ב trailing return types

**auto** inc(int x) -> int

{

return x+1;

}

כאשר מדובר בתבניות, אפשר להגדיר את הטיפוס המוחזר על סמך טיפוסי הארגומנטים בשימוש ב ב decltype

**template <typename T>**

**auto** inc(T x) -> decltype(x)

{

return x+1;

}

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

טריק: איך אפשר לדעת מהו הטיפוס שהקומפיילר החליט לתת למשתנה מסוים, פשוט לנסות להכניס לו ערך שבוודאי לא שייך אליו ואז לראות על מה הקומפיילר מתלונן...



### `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.

### Scoped enum/ enum-class

שינוי בהגדרת אנומרציה מאפשר אנומרציה קצת דומה ל C#. אם משתמשים ב enum class במקום סתם enum אז האיברים של האנומרציה מוגבלים לסקופ של האנומרציה.



כמו כן, ניתן להגדיר את הטיפוס הפרמיטיבי עליו מתבססת האנומרציה:



במקרה של enum-class טיפוס ברירת המחדל הינו int.

הכתיב החדש גם מאפשר להתייחס לאנומרציה כ incomplete types ובכך מקטין את התלות בין קבצים.

החל מ מגרסא 11, אנומרציה זה לא int אלא טיפוס בפני עצמו. פירוש הדבר שאפשר להעמיס פונקציה על בסיס טיפוס האנומרציה אותו היא מקבלת, וכן אפשר להעמיס את אופרטור ++ (מה שאין כן בטיפוסים פרימיטיביים)

### static\_assert

אסרט מוכר בצורת זמן-ריצה. כאן יש אפשרות להגדיר בזמן קומפילציה. אסרט נכון ימנע קומפילציה כאשר תנאים מסוימים לא מתקיימים. זה מאד שימושי בתבניות.

בדוגמא הבאה מודגם גם שימוש בתבניות על בסיס מספרי (ולא על בסיס טיפוסי מידע)



שים לב בדוגמא הבאה לשימוש ב has\_virtual\_destructor



### noexcept

(מחליף את throw()).

כאשר מילה זו מופיעה בכותרת של פונקציה הפונקציה לא אמורה לזרוק חריגים.

אבל אם היא תזרוק, החריג **לא** ייקלט על ידי בלוק try/catch שעוטף את הפונקציה (אם יש כזה) אלא יגרום לסיום התכנית או לחילופין, לקריאה לפונקציה שהוגדרה באמצעות std::set\_terminate().

חיוני במקרה שהתוכנה נתקלה במצב שאין שום דרך לתקנו ויש לסיים את כל התכנית.

מיירס ממליץ להשתמש בזה כל היכא דאפשר משום שזה מאפשר לקומפיילר לייצר קוד יעיל יותר.

**noexcept** is equivalent to **noexcept(true)**

**noexcept(false)** does nothing

### init object with {}

לקלאס הבא אין ctor ואעפ"כ ניתן לאתחל את ערכיו.

template<typename T>

struct Point

{

T x;

T y;

};

Point<int> p = { 10, 11 };

זהו כתיב חדש של השפה. מובן מאליו שהשדות צריכים להיות פומביים.

באופן דומה אפשר לאתחל וקטור

vector<in> v = {1,2,3};

## 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));

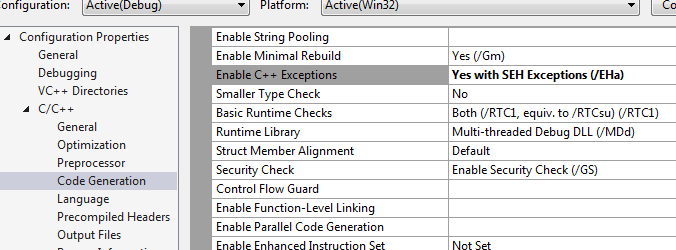
}

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

### SEH exceptions

כברירת מחדל, CPP לא תופס חריגים כגון חלוקה באפס או הפרת גישה, משום התקורה שבדבר.

ניתן לאפשר זאת כך:



### 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

המרת טיפוסים זו פעולה מוכרת

double d = 3.14

int n = (int)d;

int n = int(d); // functional cast

ב C++ פעולה זו מתעדנת.

static\_cast

זהו הפשוט ביותר. המרה בין טיפוסים "מאותה משפחה".

הערה: כרגיל, אם משתמשים בקאסט אשר כרוך באיבוד מידע הקומפיילר מזהיר. זו הדרך **למנוע** אזהרה זו.

const\_cast

מאפשר לעקוף הצהרה על מצביע ל'קבוע' .

void constcast(const tint & i) {

int & j = const\_cast<int&>(i);

j = 10;

}

זה עובד, כלומר, אף שהמשתמש חושב שה const בכותרת מבטיח אותו מפני שינוי, מתודה זו כן משנה את המשתנה שעובר אליה.

[לא הבינותי למה יש בכלל אפשרות כזו]

reinterpret\_cast

המרה שהיא חסרת משמעות מבחינת הקומפיילר, כגון המרת מספר למצביע, או מצביע של טיפוס אחד אל מצביע של טיפוס אחר כששני הטיפוסים לא קשורים אחד לשני. המתכנת מצהיר שהוא יודע מה הוא עושה (בדרך כלל כוונתו שבאמת המצביע של הטיפוס האחר היה במקורו המצביע של הטיפוס האחד)

dynamic\_cast

להמרת מצביעים על קלאסים תוך התחשבות ביחסי אב-בן.

משתמש במנגנון RTTI להבטיח שההמרה של קלאסים אכן תיתכן

ההמלצה הכללית היא לא להשתמש במנגנונים הישנים אלא באחד מהארבעה הנ"ל.

דוגמאות:



כ

### 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.



### 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



## Smart Pointers

### unique\_ptr

Class C1 {

public:

C1(int x){}

}

...

auto cp1 = unique\_ptr<C1>{}; // nullptr

auto cp2 = unique\_ptr<C1>{new C1(1);}; // ptr to new instance

auto cp3 = make\_unique<Res>(19); // ptr to **new instance** (with perfect fwd)

auto cp4 = std::move(cp3); // Res(19) moved to cp4

if (cp4) // check if cp4 owns resource

{

cout << “CP4 owns a resource” << “\n”;

}

Res copyOfcp4 = \*cp4; // copy ctor

Res & refTocp4 = cp4;

Res \* ptrTocp4 = cp4.get(); // don’t do it. this break uniqness

Res \* ptrTo = cp4.release(); // this is the right way. cp4 now empty

cp4.reset() // delete object and release

### shared\_ptr

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

המונה אשר מחזיק את מספר המופעים של המצביע נמצא מחוץ לאובייקט עצמו. 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 צריך להשתמש ב make\_shared.

class Class1{

public:

Class1(int a1, int a2)

{.. }

}

Class1 \* c1 = new Class1( 10, 11);

shared\_ptr<Class1> c2 = make\_shared<Class1>( 10, 11);

האובייקט c2 הינו פרוקסי. כלומר הוא חושף את אותן מתודות ש c1 חושף, ויש לו בנוסף את המתודות הקשורות לכך שהוא shaerd\_ptr.

כך למשל, אם \*c1 מחזיר את האובייקט המוצבע, כך גם \*c2.

אם c1->GetName() חוקי אז גם c2->GetName() קיים וחוקי.

## Solid Principles/ Patterns

### OCP – Open Close Principle

הכותרת כאן היא ברורה. סגור לשינויים, פתוח להוספות. כלומר הקוד צריך להיבנות בצורה כזו שעל מנת להוסיף לו תכונות או יכולות לא יהיה צורך לשנות קוד קיים אלא רק להוסיף.

הדוגמא כאן בעזרת Specification Pattern

חנות אינטרנטית מאפשרת למבקרים לראות את המוצרים לפי חתכים שונים. המתכנת מממש את הדרישות של בעל החנות אולם רוצה להשאיר את הקוד פתוח להרחבות לדוגמא יצירת חתכים נוספים או שילוב של חתכים או שינוי באופן הגדרת החתך.

זהו מוצר:

struct Product {

std::string name;

Color color;

Size size;

};

זהו ממשק המגדיר תנאי ליצירת חתך.

Template <typename T> struct Ispecification {

virtual bool is\_satisfied(T\* item) = 0;

};

וזהו ממשק רחב יותר המגדיר מסנן אשר משתמש בממשק הנ"ל

template <typename T> struct IFilter {

virtual std::vector<T\*> filter(std::vector<T\*> items, ISpecification<T>& spec) = 0;

};

ממשק זה מקבל ווקטור ומחזיר ווקטור מצומצם יותר על פי תנאי המסופק לו. המימוש הפשוט הוא טריוויאלי [לא הבינותי למה צריך ממשק לצורך כך]

זו דוגמא למימוש תנאי ספציפי

struct SizeSpecification: ISpecification<Product> {

Size size;

explicit SizeSpecification(const Size size) : size{size} { }

bool is\_satisfied(Product\* item) override {

return item->size == size;

}

};

וזו דוגמא למימוש תנאי המסתמך על שני תנאים קודמים

template <typename T> struct **AndSpecification** : ISpecification<T> {

ISpecification<T>& first;

ISpecification<T>& second;

AndSpecification(ISpecification<T>& first, ISpecification<T>& second):

first{first}, second{second} { }

bool is\_satisfied(T\* item) override {

return first.is\_satisfied(item) **&&** second.is\_satisfied(item);

}

};

לדוגמא המלאה ראה

New CPP 2. Files\OCP.cpp

### ISP – Interface segregation principle

הרעיון כאן הוא להימנע ממשקים מורכבים. לדוגמא מדפסת רבת פונקציות (הדפסה, סריקה, פקס וכו'). אם מספקים לקליינט ממשק עם כל הפונקציות, כל שינוי עלול להשפיע עליו. הוא מעוניין רק בהדפסה והנה שינוי בפונקציית הסריקה מאלץ אותו לקמפל מחדש. אז עדיף לספק שלש מממשקים שכל אחד אחראי על פונקציה אחת בלבד.

### Singleton

ב C++ לא מספיק לקבוע את הקונסט' כפרטי, צריך גם למנוע את copy, assign ctor. זה קל לעשות (בגרסא המודרנית) בעזרת delete.

בדרך כלל יוצרים מופע של הקלאס בעזרת new וסומכים על כך שהוא יחיה לאורך כל חיי התוכנית. כלומר אין delete על אותו new. . דרך אולי יותר יעילה ליצור מופע היא באמצעות static

static Singleton& GetInstance()

{

static Singleton s;

return s;

}

זה גם thread safe.

שים לב שהפונקציה מחזירה ref ולא מצביע.

## Overloading the << Operator

class Date

{

int mo, da, yr;

public:

Date(int m, int d, int y)

{

mo = m; da = d; yr = y;

}

friend ostream& operator<<(ostream& os, const Date& dt);

};

ostream& operator<<(ostream& os, const Date& dt)

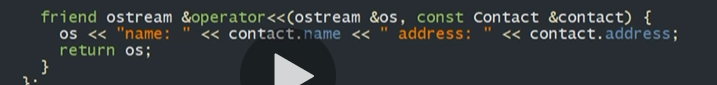
{

os << dt.mo << '/' << dt.da << '/' << dt.yr;

return os;

}

אפשר גם לכתוב את המימוש בגוף הקלאס:



1. יתכן שבקומפיילרים מסוימים כן עושים זאת אבל זה לא סטנדרט. [↑](#footnote-ref-1)
2. כאן, typename נועד להבהיר לקומפיילר שמדובר ב'טיפוס' ולא ב'ערך'. [↑](#footnote-ref-2)
3. ResClass arr[1] [↑](#footnote-ref-3)