כללי

* **איך לקבל מערך!?!?!**

CONST:

* בסוף כל מתודה שלא משנה כלום
  + גם VIRTUAL
  + גם אופרטורים נדרסים
* בתוך הארגומנטים
  + דריסת אופרטורים
  + קונסטרקטורים

**לבטל מתודות!!!!**

* // COPY CONSTRUCTOR DISALLOW  
  survival\_kit(**const** survival\_kit& other) = **delete**;  
    
  // MOVE CONSTRUCTOR DISALLOW  
  survival\_kit(survival\_kit && other) = **delete**;

ירושה רגילה

* ירושה שהיא PURE VIRTUAL
  + עד שלא מממשים את כל מה שהוא וירטואל – המחלקה היורשת נחשבת אבסטרקטית
  + לשים CONST – גם במקור. אחרת זה לא ידרוס.
  + אי אפשר לאתחל את הMEMVERS ברשימת אתחול – רק בקונסטרקטור.

בירושת יהלום:

* אם מקבלים ערכים – **להעביר לקונסטרקטור של האב המשותף**, ואז לשני היורשים.
* אם באב יש מתודה וירטואלית, ודורסים אותה בשני הילדים – לדרוס גם בנכד המשותף, או לקרוא לרלוונטית עם B1::f().
* לשים PUBLIC בכל מקום שצריך – גם לפני כל אחד מהילדים – במחלקה הסופית.

טמפלייטים

* כשמחזירים אובייקט – להחזיר עם החתימה המלאה bla<T,S,A>
* כשדורסים הדפסה – לא צריך את החתימה המלאה.
* אתחול דיפולטיבי: \_t(T())
* אפשר לקבל משתנים בתוך ה- <>
  + למשל החתימה: template<class T, int size>
  + ובתוכה האתחול של המשתנה T arr[size] מאפשר אתחול של המערך בגודל size, כי הוא ידוע בזמן קומפילציה.
  + בדריסה של >> לא צריך להגיד other.size - זה משתנה גלובלי
* **התמחות**:
  + תהיה מחוץ למחלקה
  + לפני ההכרזה נשים template<>
  + נוסיף את ההגדרה של המחלקה כדי שידע על מי אנחנו מדברים.
  + בכל מקום שהיה T נשים את האובייקט הייעודי
  + החתימה צריכה להיות מדוייקת!!! לרמת ה const
* עובד גם אם יש כמה סוגים של משתניםםם
* ביחד עם זה:
* T bla() **const** {  
   **return** first+second+third;
* T first; U second; S third;
* נוסיף את זה:
* **template**<>  
  Complex survival\_kit<Complex, Complex, Complex>::bla() **const** {  
   **return** Complex(first.getReal()+second.getReal()+third.getReal());  
  }

ועוד דברים טובים: TEMPLATE ITERATOR

המתכון:

* + ניתן לו שם (טייפדף) וגם לא יודעים מה הוא אז נשים טייפניים, ואז מה שנרצה (עם T):
* **typedef typename** std::vector<T>::const\_iterator const\_iterator

ואז קליל:

const\_iterator begin() **const** {

**return** \_matrix.cbegin();

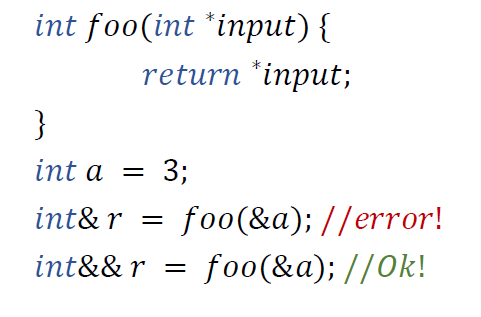
}

EXCEPTIONS

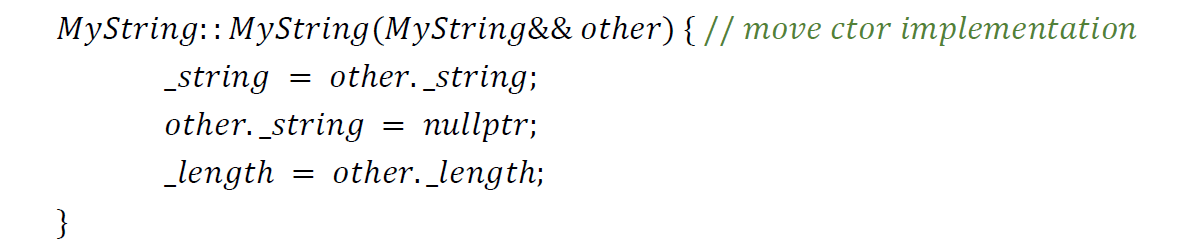
* + לרשת מ- runtime\_error או מ logic\_error
  + לעשות את הקונסטרקטור PUBLIC !!!!
    - לקבל הודעה כ- const char\*
    - להעביר אותו ברשימת אתחול למי שירשתי ממנו.
  + תפיסה של כל אקספשן – (...) אבל אי אפשר לדעת מי זה
  + אם צריך רק לזרוק בלי לממש – פשוט לעשות   
    throw std::logic\_error("blab la bla");
* תפיסה בשגיאה של קונסטרקטור – שמים את ה- TRY לפני המימוש:
* suck() **try**{  
   positives.push\_back(100);  
   **throw** (bad\_alloc());  
   nothing = 100;  
  }  
  **catch** (bad\_alloc &e)  
  {  
   **throw** stupid("I got you babe");  
  }

Move semantics

* המרה L\_val🡪R\_val : הוספת &
* Int var = 10;
* Int \*addr = &var
* המרה R\_val 🡪 L\_val : הוספת \* באמצעות פוינטר
* Int arr[] = {2,3};
* Int \*p = &arr;
* \*(p+1) = 30;



MOVE CONSTRUCTOR –



* נראה בדיוק כמו קופי, רק עם שניים &&
* יגיעו לפה רק R\_value - כלומר שאין להם מקום בזיכרון. אפשר לעשות מה שרוצים.
  + נגנוב להם את הערכים
  + נשים להם ערכים חלופיים שאין בעיה שימחקו.
* RULE OF 5
* 𝑑𝑒𝑠𝑡𝑟𝑢𝑐𝑡𝑜𝑟
* 𝑐𝑜𝑝𝑦 𝑐𝑜𝑛𝑠𝑡𝑟𝑢𝑐𝑡𝑜𝑟
* 𝑜𝑝𝑒𝑟𝑎𝑡𝑜𝑟=
* 𝑚𝑜𝑣𝑒 𝑐𝑜𝑛𝑠𝑡𝑟𝑢𝑐𝑡𝑜𝑟 – כמו הרגיל אבל עם &&
* 𝑚𝑜𝑣𝑒 𝑜𝑝𝑒𝑟𝑎𝑡𝑜𝑟= – כמו הרגיל אבל עם &&

המרות

**class** a {  
**public**:  
 a(**int**) {  
 cout << "converting given int" << endl;  
 }  
  
 a (**bool**) {  
 cout << "converting given boolean" << endl;  
 }

וחשוב חשוב – לממש operator=

a&**operator**=(**const** a& other) {  
 cout << "in = operator" << endl;  
 **return** \***this**;  
}

---- אם לא רוצים את התענוג הזה – נשים explicit על הקונסטרקטור שמקבל int

להמיר בכיוון ההפוך

* נבנה קונסטרקטור מתאים!
* נדרוס את החלופי האוטומטי –

𝑜𝑝𝑒𝑟𝑎𝑡𝑜𝑟 𝑖𝑛𝑡() 𝑐𝑜𝑛𝑠𝑡 {

𝑠𝑡𝑑∷𝑐𝑜𝑢𝑡 << "𝑜𝑝 𝑖𝑛𝑡 (𝑖𝑛𝑡 𝑓𝑟𝑜𝑚 𝑅𝑎𝑡𝑖𝑜𝑛𝑎𝑙)" << 𝑠𝑡𝑑∷𝑒𝑛𝑑𝑙;

𝑟𝑒𝑡𝑢𝑟𝑛 \_𝑛𝑢𝑚𝑒𝑟𝑎𝑡𝑜𝑟/\_𝑑𝑒𝑛𝑜𝑚𝑖𝑛𝑎𝑡𝑜𝑟;

}