סיכומים למבחן בנושאים מתקדמים בתכנות

סמסטר ב' 2010 (יוסי הלחמי)

C++ - כללי:

CTORs, DTORs:

- סדר השחרור האוטומטי שנקרא בסוף בלוק (למשל מתודה) הפוך לסדר היצירה. למשל אם יצרו את A1 ואז את A2, ביציאה ישוחרר A2 ואז A1.
- הקומפיילר יוצר אוטומטית, אלא אם אחד מה-CTORs מוגדרים באופן מפורש. **לכן כדאי תמיד להגדיר לבד את כל הארבעה האלה**. CTOR, DTOR, copy CTOR, assignment operator

new, delete:

- בכל קריאה ל-new עם [] (למשל new int[20]) הקריאה ל-delete צריכה גם להעשות עם []. אם נעשית קריאה רק ל-delete, רק האובייקט הראשון במערך ייהרס והשאר לא ישתחררו.
- הקצאה על ה-heap ולא על ה-stack, נדרשת קריאה מפורשת ל-delete. בהקצאה רגילה על ה-stack האובייקט מת בסוף הבלוק בו הוקצה.

Reference:

- חייב לקבל ערך באתחול וערכו לא יכול להשתנות לאחר מכן.

Const:

- להגדרת קבועים (עדיף על פני DEFINE)
- בסוף הגדרת פונקציה: אוכף שהפונקציה לא משנה את האובייקט אליו שייכת במהלך ריצה: void foo() const
- **this**: מצביע עצמי implicit עבור T מהטיפוס T* const.

Assignment overloading:

- אופן העמסת אופרטור השמה עבור טיפוס T:

```
const T& operator=(const T& t)
```

מקבל כקלט const reference ומחזיר const reference – את עצמו (this* – כאשר בהחזרה יש המרה מאובייקט ל-reference).

- בהשמה חשוב לבדוק שלא מנסים לשים אובייקט לעצמו ע"י בדיקה ש- (this != &t).
- במימוש זה אפשר לבצע השמה: t1 = t2 = t3 (אסוציטיביות מימין לשמאל – קודם t2 = t3).

Copy CTOR:

- מהצורה: T(const T& t) – בנאי שמקבל const reference כקלט.
- כאשר מתבצעת הצהרה עם השמה, למשל T t1 = t2 – נקרא ה-copy CTOR, לא ה-assignment operator.
- אין כאן צורך בבדיקת שוני, כי יוצרים אובייקט חדש.
- כשאובייקט עובר כפרמטר לפונקציה by value מופעל בקריאה ה-copy CTOR שלו ונוצר עותק מקומי בפונקציה. כמו כן בהחזרת אובייקט ע"י return (האובייקט המקומי מועתק, ההעתק זה מה שחוזר, ובסוף הפונקציה המקומי נהרס).

רשימת אתחול:

- דוגמא לסינטקס: T::T(...): field1(<value>),...,field_n(<value>) {...}
- סדר האתחול הוא לפי סדר הצהרת ה-members ולא סדר ההופעה ברשימת האתחול.
  - אתחול רגיל: T::T(S s) { my_s = s; } – עלות: copy CTOR לפרמטר s + CTOR ל-my_s כי בכל תחילת CTOR נקראים כל ה-CTORS של כל ה-members שלו.
  - אתחול עם רשימת אתחול: T::T(S& s):my_s(s) {} – עלות: רק copy CTOR הבונה את my_s מ-s.
- מתי חייבים רשימת אתחול: למשתני const (אחרת יוגדרו ע"י default CTOR ולא יהיה ניתן לשנות את ערכם), ל-reference members, ו-members ללא default CTOR (למשל כאלה שיש להם רק CTOR עם פרמטרים; זה מכיוון שבתחילת CTOR נקראים ה-default CTORs של כל ה-members, כאמור).

איך אוסרים copy CTOR או assignment operator:

צריך להכריז עליהם כדי שהקומפיילר לא ייצור אותם באופן אוטומטי, ואז: או שלא עושים להם מימוש ואז בניסיון שימוש בקוד תהיה שגיאת link, או שמגדירים אותם תחת private וזה יגרור גם כן נפילה בזמן קופילציה.

Standard C++ IO:

- שימוש ב-cout, cerr (מסוג ostream) ו-cin (מסוג istream).
- בדיקת טיפוסים נעשית בזמן קומפילציה ולא זמן ריצה כמו ב-c io.
- שימוש: הצהרה **#include <iostream>** ו-**using namespace std;**
  - דוגמא ל-out: cout << "My name is " << name << "\n";.
  - דוגמא ל-in: cin >> name;
- אם מערבבים c io עם c++ io סדר הפקודות מתבצע אסינכרונית ולכן לא ידוע סדר ביצוע הפעולות – לכן צריך להימנע מכך.
- הגדרת output לאובייקטים ע"י העמסת אופרטור >>:

ostream& operator<<(ostream& os, const T& t) { <os> ל-<השמה...; return os; }

- הגדרת input לאובייקטים ע"י העמסת אופרטור <<:

istream& is operator>>(istream& is, T& t) {... return is;}

בגוף מגדירים מתשנים מסוג מערך char, ואז ע"י is >> param1 >> ... >> param_n; מקבלים את הערכים לאותם משתנים. אז ניתן להשתמש בהם עם setters כדי לתת ערכים לשדות של t. בסוף מחזירים את is. **כאן t אינו const כי משנים את השדות שלו**.

- <u>הערות מימוש</u>: את ההצהרה עושים בקובץ ה-h מחוץ ל-class{...} או בפנים עם פרפיקס **friend**, ואת המימוש עצמו בקובץ ה-cpp עושים ללא תחילית T::.

Inline:

- הצהרת פונקציה כ-inline מגדירה לקומפיילר להחליף בכל קריאה לפונקציה את הקוד בגוף הפונקציה עם הפרמטרים המתאימים – יכול לייעל.
- <u>מתי לא</u>: פונקציות רקורסיביות, פונקציות virtual, בד"כ פונקציות עם לולאות ופונקציות גדולות. אם פונקציה רקורסיבית תוגדר כ-inline, היא עדיין תתנהג כרגילה.
- מימוש של פונקציות inline צריך להיות תמיד שורה מהצורה: ... return בלבד.
- פונקציות שממומשות בתוך ה-class T {...} הן אוטומטית inline. כל פונקציות ה-inline בקובץ header אחד רואות אחת את השניה בקומפילציה ללא חשיבות לסדר ביניהן.
- פונקציה שמוגדרת inline והמימוש לא עם ההצהרה בכל מקרה צריכה להיות ממומשת בקובץ ה-header כי בזמן קומפילציה עדיין לא נראה קובץ ה-cpp. כך שבכל מקרה: <u>מימוש inline צריך להיות בקובץ ה-header</u>.

Overloading:

- <u>Mangled name</u>: השם "מאחורי הקלעים" של הפונקציה המורכב משמה וה-type של הפרמטרים שלה, ללא התחשבות ב-type המוחזר.
- הפונקציה המתאימה נקראת <u>בזמן ריצה</u>.
- <u>אופרטור השוואה</u>: bool operator==(const T& t) const
- <u>אופרטור המרה</u>: operator int() const – מגדיר עבור טיפוס T (שם מגדירים את האופרטור) מה מוחזר במקרה שנעשה על האובייקט cast ל-int. המימוש צריך להחזיר ערך מהטיפוס אליו ממירים (בדוגמא זו – int).
- אופרטורים שלא ניתן לעשות עליהם overload: sizeof ?: . *. :: (האופרטור :: בעל העדיפות הגבוהה ביותר ב-c++).
- אסור להמציא אופרטורים שלא בשפה ולא ניתן לשנות קדימות אופרטורים

המרה:

- אם לטיפוס X יש CTOR שמקבל פרמטר אחד מסוג Y, ול-Y יש אופרטור המרה ל-X, הקריאה X(y) ל-y מטיפוס Y תקרא ל-CTOR של X עם פרמטר y, במקום לפונקציית ההמרה ל-X שהוגדרה במחלקה Y. כדי לקרוא להמרה צריך מפורשות: y.operatorX().

- אם יש לטיפוס Y בנאי שמקבל פרמטר אחד מטיפוס X, ויש פונקציה שפועלת על טיפוס Y, וקוראים לפונקציה עם פרמטר מטיפוס X, יופעל אוטומטית ה-CTOR של Y עם אותו X כפרמטר וישלח לפונקציה. כדי למנוע: להוסיף **explicit** לפני הגדרת הבנאי.
- המרה האוטומטית לעיל תתבצע לכל היותר 2 רמות.
- דוגמא להיזהר: עבור string s1, s2 הפעלת s1-s2 צריכה להיות שגיאה אבל implicit conversion ל-char* תבצע החסרת מצביעים.

[] Subscript operator:

- שתי פונקציות אופרטור:
  - assign: T& operator[](unsigned index)
  - retrieve: T operator[](unsigned index) **const**
- בעיה: אם נתון T t ומבצעים cout << t[2] << endl; למשל, אז כיוון ש-t אינו const תקרא פונקציית ההשמה. פתרון: המרה של המצביע ל-t להיות מצביע const ל-t: (*(const T*)&t).
- הבדל בין המרה ל-cast: המרה משנה מסוג לסוג, cast אומר להסתכל על אותו מקום (ביטים) לא כסוג 1 אלא כסוג אחר.

new, delete העמסת:

- New: void* operator new(size_t size) – צריך להקצות זיכרון ולהחזיר מצביע אליו.
- Delete: void operator delete(void* ptr) – משחרר את המקום של המצביע.
- קריאה לברירות המחדל ולא ל-overloaded: ::new, ::delete – למשל ::delete ptr;
- אלו אופרטורים סטטיים אבל לא צריך להגדירם כך כי הקומפיילר דואג לזה.
- אם נעשתה העמסה זו, צריך לדאוג גם להעמסת []new ו-[]delete

העמסה גלובלית ל-new, delete:

- העמסה גלובלית לכל הופעה של new ו-delete, לא פר מחלקה. מאפשר מעקב אחר דליפות זיכרון למשל.
- New: **inline void* operator new(unsigned int size, const char* filename, int lineNo)** (באופן דומה ל-[]new).
- Delete: דומה לקודם: **inline void* operator delete(void* ptr)** (באופן דומה ל-[]delete).
- בונים קובץ h לטיפול במעקב זיכרון: **CatchMemoryLeak.h** המכיל את המימושים ה-inline ל-new,... המימושים הללו קוראים ל-saveInStorage הנקרא בכל new, removeFromStrorage הנקרא בכל delete. מספקים גם reportUnreleasedHeap להדפסת מצב הזיכרון. ...report, ...remove, ...save ממומשות בקובץ ה-cpp.
- בקובץ זה מגדירים מקרו **#define DEBUG_NEW new(__FILE__,__LINE__)** כאשר **__FILE__** מקרו המאחסן את ה-path המלא של הקובץ, **__LINE__** מקרו למספר השורה בה נמצאים. מצגת ACPP עמוד 79: פירוט מלא.
- קובץ ה-cpp: מחזיקים map מ-unsigned long (כתובת הזיכרון, מפתח יחודי) אל HeapInfo_s (מבנה מיוחד שהוגדר להחזקת כל המידע על הקצאת זיכרון מסויימת – קובץ, שורה...) כאשר הקומפרטור הוא less<unsigned long>. כאן יוחזקו המיפויים מכתובת למידע. אין include לקובץ ה-h כדי שיהיה שמיש ב-new המקורי. כאן יש מימושים ל-saveInStorage, removeFromStorage, reportUnreleasedHeap.

Placement new:

- מקצים זיכרון פעם אחת על ה-heap, ואז על זיכרון זה דורסים כל פעם מה שרוצים.
- דורש: **#include <new>**
- דוגמא: char* mem = new char[100]; T* pt = new(mem) T;
- הריסה דורשת קריאה מפורשת ל-DTOR: pt->~T(); **לא לקרוא סתם ל-delete!** רק אחרי קריאה לכל DTOR של אובייקט ששמנו על הזיכרון צריך לשחררו, במקרה זה ע"י delete[] mem.

פונקציות ומשתנים סטטיים:

- הגדרה בתוך הגדרת המחלקה בקובץ ה-h ע"י הוספת static.
- חייבים להיות מאותחלים פעם אחת בקובץ cpp כאשר גישה (גם לאתחול) למשתנה סטטי או לפונקציה סטטית היא: <class name>::<f | m>
- פונקציה סטטית לא יכולה להיות const: כי זה אומר שהיא לא משנה את האובייקט, אבל כאן אין אובייקט.

ירושה:

- מחלקה יורשת לא יכולה להחזיק member מסוג המחלקה ממנה יורשת.
- אם A יורשת מ-B, נגדיר את B כך: **class A: public B {...}**
- גישה למימוש מקורי של פונקציה שנעשה עליה overload: a.B::foo() – קורא ל-foo של B ולא לזו של A.
- בעת קריאה ל-CTOR של A, נקרא אוטומטית ה-default CTOR של B. כדי לקרוא ל-CTOR אחר, נעשה זאת כמו רשימת אתחול:
  A::A(int i): B(i) {...} – דואג לקרוא ל-CTOR של B שמקבל int במקום ל-CTOR הדיפולטי.

Protected:

- נגדיר members (או פונקציות) להיות protected כדי לאפשר אליהם גישה ישירה (ויעילה) ממחלקות יורשות אבל עדיין להסתירם מכל השאר.

ירושה מרובה:

- סינטקס: **class B: public A1, public A2 {...}**
- בעיה: אם ב-A1, A2 יש פונקציות עם אותה חתימה, אז B חייבת לממש לבדה פונקציה זו כדי למנוע התנגשות.
- בעיה: אם A1, A2 יורשים מ-C, אז שדות C יופיעו פעמיים ב-B – לכן משתמשים ב-virtual:

```
class A1: virtual public C {...};   class A2: virtual public C {...};   class B: public A1, public A2 {...};
```

- במקרה של cast לאחד מה-base classes מוחזר מצביע למקום במפת הזיכרון של אותו טיפוס אליו מבצעים cast. כך ש-b(A1) שונה מ-(A2)b.

פולימורפיזם:

- **virtual**: הגדרת פונקציה כוירטואלית במחלקת הבסיס תגרום לכך שהמחלקה תחזיר טבלת פונקציות וירטואליות. ההחלטה לאיזו פונקציה לקרוא תהיה בזמן ריצה. אם אין virtual. פונקציה virtual היא פולימורפית.
- לא חובה להוסיף "virtual" גם במחלקח היורשת אבל כדאי לבהירות.
- פונקציה virtual לא יכולה להיות inline (כי מוחלטת רק בזמן ריצה).
- **בלי virtual**: הטיפוס מוחלט בזמן קומפילציה, לכן קריאה למתודה כלשהי תתייחס לטיפוס הקומפילציה, שיכול להיות הבסיס, גם אם בפועל הוא יורש ויש לו את המתודה הזו שדורסת את מתודת הבסיס. עדיין, המתודה בגרסה של הבסיס היא שתקרא.
- Virtual table: אובייקט בעל מתודות וירטואליות מחזיק כפרמטר חבוי ראשון טבלה וירטואלית, מערך לכתובות לפונקציות הוירטואליות. כך כל קריאה לא עושה קפיצה אחת, אלא 2 – אחת לטבלה ומהטבלה לפונקציה המתאימה.
- Virtual DTOR: כל מחלקה עם פונקציה וירטואלית אחת לפחות חייבת שה-DTOR יהיה virtual (למרות שלא נופל בקומפילציה). אז ה-DTOR יהרוס קודם את ה-derived ואז את ה-base (הפוך לבנאי – בנאי קודם קורא לבנאי של ה-base ורק אז של ה-derived). אם ה-DTOR לא יהיה virtual תהיה דליפת זיכרון (יהרס טיפוס זמן קומפילציה ולא זמן ריצה).
- גישה לפונקציה (... public / private) לא יכולה להשתנות במחלקה derived אלא נקבעת במחלקת ה-base. אם תוגדר הפונ' ב-derived בפחות מ-public, cast ל-base יאפשר גישה לאותה מתודה **של ה-derived** למרות שמוגדרת ביורשת כפחות מ-public.

Abstract base class:

- מאפשר ליצור מחלקה שתהווה בסיס למחלקות אחרות, אך שלא יהיה ניתן לבצע לה אינסטנציאציה בעצמה.
- הגדרת מחלקה כאבסטרקטית: לפחות פונקציה אחת מגדירים כ-virtual עם 0= בסוף.
- כל פונקציה יורשת מחוייבת לממש את המתודות שהוגדרו עם 0=. עם זאת, עדיין אפשר לכתוב לה מימוש גם במחלקת הבסיס.

Friends:

- הכרזה על מתודה void foo() של A שתהיה נגישה לאיברי B: במחלקה B נשים friend void A::foo(); - הצהרה על מתודה חיצונית זו כ-friend למחלקה B וכך תהיה לה גישה לכל ה-members וה-member functions של B.
- הכרזה על מחלקה כ-friend: friend class X; ההצהרה מגדיר את X להיות בעלת גישה לכל ה-members, member-functions במחלקה המצהירה עליה כ-friend.
- לא טרנזיטיבי.
- Friendship אינה עוברת בירושה: אם Base היא friend ב-X, ו-derived יורש מ-base, derived **לא יהיה friend של X**.

Templates:

- הגדרת פונקציה כתבנית, למשל int foo(int i): **template <class T> T foo(T t)**. קריאה לפונקציה באופן רגיל, למשל עבור int לא צריך foo<int>(...) אלא פשוט foo(...). אם הפונ' מקבלת שני פרמטרים T, אז קריאה ל-foo(int, float) היא רב משמעית – הקומפיילר לא ידע מהו T, ולכן כאן צריך מפורשות למשל foo<int>(i,f) (כאשר ל-f יעשה cast ל-int).
- הגדרת מחלקה כתבנית: לפני **class ... {...}** כותבים: **template <class T>** - T הוא הטיפוס הכללי איתו תוגדר המחלקה.
- כל הגדרת משתנה מטיפוס מחלקה זו על טיפוס ספציפי תהיה מהצורה: TemplateClassName<int> a; למשל.
- אפשר כמה פרמטרים כלליים או טיפוסים מוגדרים: **template <class T, int i>** יכול לקבל למשל <float,5> או **template <class T1, class T2>** יכול לקבל למשל <int, string>.
- הטיפוסים המוגדרים בתבנית (T) הם const ולא ניתן להשתמש בהם לצרכים אחרים, כמו int T; - לא חוקי במחלקה שמוגדרת תבנית מעל T.
- כל שימוש בפרמטר כללי צריך להיות אחרי שהוגדר, למשל: **template <class T, T* TP>** - T* TP לא יכול להיות ראשון, כי T לא מוגדר עדיין.
- <u>הכל נכתב בקובץ ה-h</u>.
- הגדרת פרמטר דיפולטי, למשל int: **template <class T = int>**. בכל מקרה בהגדרה צריך <> גם אם ריקים (ואז יהיה int). אחרי כל הגדרת default param יכולים להיות רק הגדרות default params, לכל כל הפרמטרים שנותנים להם ערך דיפולטי צריכים להיות בסוף. <u>Default params אפשרי רק ב-template classes ולא ב-template functions</u>!
- <u>צמצום קוד ב-template</u>: ה-template, שמוגדר כולו ב-h הוא כמו inline ויכול להווצר הרבה קוד. ניתן לחסוך ע"י הוצאת כל הפונקציונאליות שאינה תלויה בפרמטרים הכלליים למחלקה נפרדת – עם מימושים בקובץ cpp, ולגרום למחלקת ה-template לרשת ממנה. כך חלק מהקוד לא ישוכפל סתם (עבור על טיפוס ממשי שיבחר במקום הטיפוס הגנרי T).
- <u>משתנים סטטיים</u>: ניתן להגדירם במחלקה template באופן רגיל, עם שימוש או בלי שימוש בטיפוסים הכלליים, ואתחולם חייב להיות בקובץ cpp כלשהו – באחריות המשתמש לאתחל **עבור כל טיפוס פרקטי בו יש שימוש במחלקת התבנית**. למשל אם יש שני משתנים סטטיים, אחד int תמיד ואחד T, ומשתמש רוצה להשתמש במחלקה מסוג T = int* הוא יצטרך לאתחל את משתנה ה-int הסטטי ואת משתנה ה-T עבור T=int*. <u>בקיצור</u>: לכל משתנה סטטי ב-template class צריך להיות עותק מאותחל לכל טיפוס ממשי שייעשה בו שימוש.

Exceptions:

- ברגע שנזרק exception כל המשתנים על ה-stack בדרך ישוחררו אוטומטית, אך לא משתנים על ה-heap, ולכן בעת תפיסה צריך לדאוג לשחרר משתני heap (שהוקצו ע"י new) באופן מפורש בתוך בלוק ה-try-catch.
- <u>סינטקס</u>: **try {...} catch (type exp) { <release heap vars> }**. אם ב-catch יש הגדרה של טיפוס מסויים שהוא מקבל, אם יזרק טיפוס כזה הוא יתפס ב-catch הזה. <u>כדאי להשתמש בקבלת type& expr כדי למנוע העתקה של האובייקט בכל זריקה</u>. כדי לזרוק את אותו אובייקט שהתקבל מה-catch הנוכחי לבא אחריו, פשוט משתמשים ב-throw; בלי expression.
- <u>פקודת throw</u>: יכולה להיות עם ביטוי <expression> throw או בלי: throw;.
- אם נזרק Derived הוא יכול להתפס ע"י catch(Base& ex), אך אם מתוכו יזרק **throw ex** ולא **throw** – החלק ה-Derived **יחתך** וכאילו נזרק ביטוי מסוג Base. לכן כדי לשמור על האובייקט בשלמותו בהעברה הלאה צריך להשתמש רק ב-throw;.
- <u>חריגים ב-CTOR</u>: זריקת חריג ב-CTOR גורמת להריסה של כל החלקים שנעשתה להם בניה עד רגע הזריקה (members שנבנו עד כה).
- <u>עדיף</u>: לכתוב CTOR, DTOR ללא חריגים ולקרוא חיצונית למתודת אתחול והריסה שיכילו קוד מסוכן שעלול לזרוק exceptions.

C and C++:

- הוספת **extern "C"** בתחילת קובץ cpp אומר לקומפיילר להשתמש בקונבקנציית שמות של C, כלומר פונקציות שיוגדרו יהיו בעל השם שניתן להן ולא mangled name כמו c++.
- בכל קומפיילר קיים MACRO בשם __cplusplus שמציין האם עובדים עם קומפיילר c++, ואז ע"י #ifdef __cplusplus ... #else ... #endif אפשר לכתוב קוד שיתאים גם ל-c וגם ל-c++.
- ב-C תוים קבועים הם מטיפוס int וב-C++ הם מטיפוס char – ניתן להבחין ביניהם ע"י sizeof('a') למשל (כאשר 'a' הוא תו קבוע).

CTORs and DTORs:

- CTOR נקרא כאשר: הגדרת אובייקט או מערך אובייקטים, העברת פרמטר לפונקציה by value, החזרת return value (copy CTOR ל-temp).
- DTOR נקרא כאשר: נעשית קריאה ל-delete לאובייקט על ה-heap, הריסת אובייקטים סטטיים אחרי סוף ()main, סוף scope (של פונקציה למשל), הריסת temp בסוף ביטוי בו נוצרו (למשל הריסת temp אחרי החזרת ערך מפונקציה).
- חיסכון: כדאי להעביר by reference כשאפשר כדי לחסוך copy CTOR, DTOR וכדאי להשתמש במשתנים זמניים (למשל return <expression>) במקום להגדיר משתנים explicitly – כך חוסכים CTORs ו-DTORs נוספים.

Return by Value / by Reference:

- אופציות להחזרה by reference: בהחזרת אובייקט by reference צריך לבדוק שלא מחזירים reference לאובייקט שנהרס (אם למשל הוגדר ב-scope של הפונקציה המחזירה), לא לאובייקט שמוגדר על ה-heap שאח"כ תהיה דליפת זיכרון ולא לאובייקט שהוגדר סטטי מקומית בפונקציה – במידה ויותר מזמני אחד עלול להיות קיים (כמו בדוגמא (x1 + x2) == (x3 + x4)) – הזמני הסטטי בפונקציה זהה בשני המקרים – כי הוא סטטי – ולכן כאן תמיד יהיה שוויון).
- מסקנה: ב- +operator צריך להחזיר by value.
- ב- =+ operator משנים את האובייקט הקורא, לא מחזירים חדש, וכיוון שהוא קיים לפני הקריאה ואחריה אפשר להחזיר כאן reference (את *this).
- מימוש +operator ע"י =+operator: מחזירים t2 =+ t1(T) – ה-cast ל-t1 יוצרת אובייקט זמני חדש מסוג T ולא const T, כך ש-t1 בכל מקרה לא משתנה. עליו נעשית פעולת =+ עם t2 ובסוף ערך זה מוחזר by value.
- צריך להגדיר את +operator כ-friend למחלקה המקבל שני פרמטרים, במקום להגדירו ללא friend המקבל פרמטר אחד (ואז השני הוא האובייקט עליו מפעילים את האופרטור). זאת כיוון שעבור המימוש השני הבא ייכשל: 5+x – אין כזה דבר 5.operator+(x)

RTTI – Runtime Type Identification:

- Dynamic cast: אופרטור שניתן להפעלה רק על טיפוסים פולימורפיים (מוגדרים ע"י המשתמש עם לפחות פונ' אחת וירטואלית) באופן הבא: **T* newt = dynamic_cast<T*>(p)** – אם p הוא מסוג T נעשה cast רגיל ומוחזר *T, אחרת מוחזר 0. כלומר זה בדיקה + cast. מאפשר לנסות לעשות המרה רק אם טיפוס זמן הריצה הוא כמצופה, ואם התנאי מתקיים (שונה מ-0) מבצעים פעולות שמתאימות לטיפוס זמן הריצה.
- typeid: לשימוש צריך **#include <typeinfo>** והשימוש הוא: typeid(p) == typeid(T) – מחזיר true אם טיפוס זמן הריצה של p הוא T (typeid מחזיר const type_info& - לא מעניין). עובד על טיפוסים פולימורפיים ולא פולימורפיים.
- **חשוב**: typeid על מצביע מחזיר טיפוס סטטי (זמן קומפילציה) וtypeid על אובייקט מחזיר טיפוס דינאמי (זמן ריצה). למשל Base* t = new Derived; יקיים typeid(t) == typeid(Base*), typeid(*t) == typeid(Derived) ואם Base& tr = *t אז typeid(tr) == typeid(Derived) למרות ש-tr הוא מטיפוס סטטי Base – כי כאן זו התייחסות לאובייקט (reference של אובייקט – אותה התייחסות, לכן מוחזר טיפוס זמן ריצה).
- שם המחלקה: **typeid(<class>).name()** מחזיר את שם המחלקה ב-char*: "Class <class name>" ל-user defined או "<name>" לטיפוס פרמיטיבי.

I/O streams:

- אובייקט iostream: בעל שני מצביעים – אחד לקלט ואחד לפלט, אסינכרוניים. יותר בטוח להשתמש בשניים נפרדים (istream, ostream).
- מחלקת הבסיס ios:
  - **Enum io_state**: מציין את מצב ה-stream (goodbit, eofbit, failbit, badbit). קריאה נכשלת אם מנסים לקרוא קובץ לא פתוח או אנו בסוף קובץ, כתיבה נכשלת אם הקובץ לא פתוח או נגמר המקום בדיסק.
  - **Enum open_mode**: הגדרת מצבים לפתיחת קובץ (לקריאה, לכתיבה, פתיחה רק אם קיים, פתיחה לכתיבה בסוף קובץ ללא שינוי קודם, פתיחה בינארית וכו').
  - **Enum seek_dir**: מגדיר התמקמות בקובץ (beg, cur, end).
  - יש enum נוסף עבור formatting (כמו skipws – דילוג על רווחים לבנים).
- לאובייקטי stream ניתן לבדוק מצב ע"י קריאות לפונקציות: bad() – מחזיר ערך שונה מ-0 אם נעשתה בקשה לפקודה לא חוקית, eof() – מחזיר אם הגענו לסוף הקובץ, fail() – מחזיר אם אחד מהקודמים קורה, good() – מחזיר את ההופכי של fail() (והפוך).

- במקרה של כישלון ניתן להשתמש ב-clear() כדי לנקות את הדגלים הבעייתיים ולנסות שוב. למשל אם עבור int k; cin >> k; מוכנס תו לא נומרי ומורם דגל כישלון, ע"י cin.clear(); cin >> str; (בטוח נצליח לקרוא את התוכן כמחרוזת) ניתן להתקדם. נשים לב שמ-cin בשלב זה עדיין לא נקרא דבר.
- **cin.ignore(5,'‘)** : התעלמות מ-5 תוים ראשונים או עד שנתקלים ברווח.
- **fstream** :
  - ע"י **#inclide <fstream>** מאפשרים גישה לקבצים. קבצים צריך לפתוח ע"י שתי פקודות : ifstream in; in.open("<file path>"); ואופן דומה ל-ofstream. אפשר גם בבנאי : ifstream in("<file path>");.
  - אם לא נסגור את הקובץ עם close() הוא יסגר ב-DTOR בסוף, אך סגירתו תאפשר פתיחה מחדש של קובץ אחר עם אותו אובייקט (חסכוני).
  - עבור char c : in.get(c) קורא תו מהקובץ, out.put(c) כותב את התו לקובץ. לחילופין : c << in ו-c >> out.

Stream manipulators : פירוט במצגת ACPP_prt עמודים 51-53

- לשימוש : **#include <iomanip>**
- User defined manipulators : יוצרים פונקציה שמקבלת (למשל) ostream& os ומחזירה ostream&, בה משנים את os ומחזירים.

...

Effective Templates :

- עבור מחלקה שרוצים לעשות כ-template :
  - יוצרים מחלקה רגילה (לא template) מעל הטיפוס *void.
  - יוצרים את מחלקת ה-template שיורשת **באופן פרטי** ממחלקת ה-*void, וכל מה שהקוד שלה עושה הוא קריאה לקוד של מחלקת ה-*void עם cast לטיפוס T בהתאם.
  - כך הקוד הארוך ממומש פעם אחת (כי מחלקת ה-*void היא רגילה), והקוד המשוכפל של התבנית לכל טיפוס ממשי יהיה קצר יותר.

Pitfalls and Tips :

- עבור T a = b :
  - אם b מסוג T מופעל copy CTOR.
  - אם יש ל-T בנאי שמקבל את הטיפוס של b, הוא יופעל.
  - אם לא זה ולא זה, ויש לטיפוס של b המרה ל-T, הוא יופעל ואז יהיה שימוש בבנאי שמקבל T (copy CTOR).
  - אם ל-T בנאי שמקבל את טיפוס b ולטיפוס b יש cast ל-T, בעדיפות יופעל הבנאי של T המקבל b. כדי שיילקח ה-cast, או שמפעילים אותו בצורה מפורשת ע"י T a = b.operator(); או לפני חתימת ה-CTOR של T המקבל את טיפוס b להוסיף **explicit** ואז ה-cast ילקח תמיד למעט מקרים בהם באופן מפורש כתוב : T a = T(b);.
- עבור T a(b); - אם b מסוג T יופעל copy CTOR, אחרת יופעל CTOR שמקבל את טיפוס b.
- עבור T a : מופעל CTOR דיפולטי.
- עבור T a(); - זהו **function prototype ולא הפעלת default CTOR!**
- **Virtual DTOR** : תמיד לשים לב שאם יש הורשה ומתודות וירטואליות לדאוג שה-DTOR יהיה virtual כדי ש-delete על Base* t = new Der() תהרוס את כל t ולא רק את החלק ה-Base שלו.
- **Double DTOR** : אם לאובייקט T יש אובייקט S שמוקצה על ה-heap ומבצעים T t1 = ...; T t2 = t1, ביציאה מ-main ייהרס t2 ובו ה-s שהוא משותף ל-t1, ולכן בהריסת t1 אחריו תהיה בעיה כי ייעשה ניסיון שחרור נוסף של אותו מקום בזיכרון של s (לכן המימוש צריך לבצע שכפול מלא).
- **מערך Derived נשלח לפונקציה כמערך Base** : במקרה כזה arr[i] בתוך הפונקציה יהיה מקום בזיכרון של התא ה-i על הטיפוס Base, ולא על הטיפוס הדינאמי Derived (שיותר גדול מ-Base). לכן לא ניתן לשלוח מערך Derived לפונקציה המקבלת מערך Base (אלא אם ייעשה cast).
- **CTORs עם ארגומנט int יחיד** : עבור אובייקט Array שיש לו בנאי עם int שהוא גודל המערך ואופרטורים [], =, אם נבצע לאחר אתחול a מסוג Array כזה a = <some number> (שכחנו []) אז יקרא constructor ל-Array עם ארגומנט <some number> ואז אופרטור ההשמה – כל מה שהושם לפני כן למקומות שונים ב-a כבר לא רלוונטי כי a אובייקט חדש. צריך להמנע מ-CTORs כאלה.

- **שימוש ב-eof()**: צריך להיזהר משימוש ב-cin.eof()! כיוון שאם ה-stream עובר למצב fail מסיבה שאינה eof (כמו input לא מהטיפוס אליו מצפים) לא תהיה התקדמות אבל cin.eof()! יהיה true תמיד – עלולים להכנס ללולאה אינסופית.
- **שימוש ב-fstream**: תמיד עדיף להשתמש ב-fstream כך שאם יש בעיה עם הקובץ ונזרק חריג, ביציאה מהפונקציה הקובץ ייסגר אוטומטית ב-DTOR של ה-fstream. אם נשתמש במתודות fopen וכאלה, צריך לעטוף את ניסיון הקריאה ב-try-catch ובתוכו לסגור את הקובץ.
- **המרות מ ול-string**: ניתן להשתמש בפונקצית template עם ostringstream/istringstream בשביל להדפיס אליו T / לקחת T מתוכו, ואז להחזיר את ה-stream.str() או את הטיפוס המתקבל.
- **Order of construction**: אובייקטים באותה compilation unit נבנים בסדר בו מופיעים, כך שאם יש אובייקט A_t שיש לו static member מסוג B_t, ונעשה אתחול לאובייקט A_t שבבנאי שלו יש קריאה לאותו static b ורק אח"כ נעשה אתחול לאותו b זו בעיה. הפתרון: ליצור פונקציה שמחזירה B_t& בה מוגדר ה-static b הזה, ובה יעשה שימוש בבנאי של A_t. כך b לא צריך להיות מאותחל בקוד בהכרח לפני קריאה לבנאי זה, כי יבנה בקריאה הראשונה ל-getB() ע"י הבנאי של A.
- **Object pointer crash**: אם נתונה פונקציה <u>לא וירטואלית</u> foo על אובייקט X כלשהו, ויוצרים X* px = NULL, קריאה ל-**xp->foo()** תיפול <u>בשורה הראשונה שיש implicit this-></u>. למשל, אם בשורות הראשונות של המתודה יש פעולות על אובייקט שהתקבל בארגומנט, או משהו שלא קשור למבנה הפנימי של X, לא תהיה נפילה. ברגע שתהיה למשל גישה לשדה a של X, שהיא למעשה גישה ל-this->a – כאן תהיה נפילה. אם המתודה <u>היא וירטואלית</u>, תהיה נפילה ב-xp->foo() כי יש ניסיון גישה ל-this->vt (ה-virtual table) – לכן מיד ניפול.
- **delete this**: מותרת אבל לא כדאי. אם כן, צריך לבדוק שהאובייקט אותחל ע"י **new** בלבד, אסור לשום member function שנוגעת ב-this לפעול אחרי המחיקה, ואף אחד לא נוגע במצביע this.

<u>STL</u>:

דרישות מינימליות מאובייקט שיוכל להיות טיפוס container: default CTOR, copy CTOR, אופרטורים: ==, <, =.

וקטור:

- הכנסה ומחיקה מהירה לסוף, איטית יותר בכל מקום אחר בגלל הצורך להזיז את כל האיברים אחרי מיקום ההכנסה / מחיקה.
- **אם insert משנה את ה-capacity כל האיטרטורים הופכים בלתי שמישים**.
- **אם insert/erase לא משנים את ה-capacity, כל האיטרטורים שנמצאים אחרי המיקום אליו הוספנו/מחקנו בלתי שמישים**.
- <u>בנאים</u>: חוץ מדיפולטי יש בנאי שמקבל גודל כמו v(100), בנאי שמקבל גודל וערך כמו v(100,val) (ממלא הוקטור ב-100 עותקים של val), ובנאי שמקבל v.begin() ו-v.end() של איטרטור וממלא את הוקטור בכל האלמנטים מההתחלה עד הסוף (אפשר להעביר מערך ע"י העברת המצביע arr והמצביע arr+numOfElements).
- v1.swap(v2): מחליף את התכנים בין הוקטורים.
- אופרטור ==: מחזיר האם יש שוויון בין כל האלמנטים בוקטור אחד לכל האלמנטים בשני (לפי הסדר וגודל הוקטורים).
- אופרטור <: כמו השוואת strings.

Deque (deck):

- כמו וקטור עם הכנסה ומחיקה יעילה גם בהתחלה ולא רק בסוף.
- הוספה מבטלת את כל האיטרטורים.
- מחיקה מהאמצע מבטלת את כל האיטרטורים, מחיקה בהתחלה / בסוף מבטלת את האיטרטור begin() / end() בהתאם.
- אין לו capacity, reverse אבל יש לו pop_front(), push_front() (בנוסף ל-pop_back(), push_back()).

List:

- אין random access, הוספה ומחיקה מהירים.
- תמיכה ב-bidirectional iterators כמו vector.
- פעולת erase מבטלת איטרטור לאלמנט המחוק, insert לא מבטלת שום איטרטור.
- <u>בנאים</u>: דיפולטי שמאתחל רשימה ריקה, בנאי שמקבל int, val ויוצר רשימה בגודל ה-int של עותקי val, ובנאי שמקבל arr ו-arr+numOfElems ויוצר רשימה האיברים במערך arr.

- מספק: push_front/back, pop_front/back, front/back.
- **insert**: פעולת **insert(iter,val)** מכניסה במקום iter את val דוחפת את הישן ב-iter ומה שאחריו אחד קדימה.
- **מכאן**: רשימה יעילה יותר ל-insert, erase מאשר מערך אך פחות יעילה ל-find כיוון שאין לה random access.
- **splice**: חלוקת הרשימה לשתי רשימות, **merge**: איחוד שתי רשימות לאחת.

'C' array:

- שימוש ע"י הוספת **#include <algorithm>**
- מאפשר שימוש ב-**sort** המקבלת מצביע למקום כלשהו במערך ומצביע למקום **אחרי המקום האחרון** במערך אותו רוצים למיין. למשל אם רוצים למיין את כל המערך arr מגודל size, נפעיל: **sort(&arr[0],&arr[size])** (ולא עד [size-1]arr). יכול לקבל פשוט begin() ו-end().
- מאפשר שימוש ב-**for_each**: בנוסף לפרמטרים האלה מקבלת גם מצביע לפונקציה הפועלת על סוג איברי arr (או ה-container עליו עובדים), והפונקציה תופעל סדרתית על כל האיברים (שוב, לא כולל האיבר שאחרי האחרון שהוא הארגומנט השני – end() או arr[size]&).
- **find**: מקבלת begin(), end(), value ומחזירה איטרטור למקום בו נמצא value או end() אם לא נמצא. אם רוצים לבצע מחיקה, צריך לבדוק שמה שהוחזר אינו end() ורק אז ניתן למחוק.
- **merge**: מקבלת a.begin(),a.end(),b.begin(),b.end(),c.begin – וממלאת את c ב-merge של a, b (כדאי לעשות להם sort לפני כן).

Associative Containers:

העיקריים: **set, map**. מכילים את הערכים באופן ממויין לפי אופרטור "<" אך ניתן לבחור מיון אחר.

pred: פונקציה המחזירה true, false. שני סוגים: אונארי ובינארי.

Set:

- רשימה של אלמנטים ממויינים. כל הפעולות (חיפוש, הכנסה, מחיקה) לוקחות logN.
- פעולת erase מבטלת איטרטורים של הפריט הנמחק, פעולת insert לא מבטלת שום איטרטורים.
- אתחול יכול לקבל פרמטר אחד של סוג המיכל או שניים: סוג המיכל ופונקציית מיון. פונקצית המיון הדיפולטית היא <.
- מכיל פונקציות ייחודיות: **includes, set_intersection, set_union, set_difference, set_symmetric_difference**. כולן (חוץ מהראשונה) מקבלות: a.begin(), a.end(), b.begin(), b.end(), c.begin() ושמות ב-c את התוצאה.

Map:

- בדיוק כמו set רק מקבל שני אלמנטים, הראשון key לפיו נעשה המיון והשני value.
- אלמנט שלישי אופציונאלי כמו ב-set: קומפרטור (על key).
- קיים אופרטור [] המקבל בערך טיפוס key. לא קיים ב-multimap. משמש להשמה וקריאה (השמה – יעדכן את הערך הממופה ל-key).

Pair:

- Struct שמכיל שני members: T1 first, T2, second בו שניהם פומביים.
- בנאים: דיפולטי וכזה המקבל שני ערכים pair(expr1, expr2) (מהטיפוסים המתאימים).
- make_pair: מקבלת שני expr ויוצרת pair מעל הטיפוסים של אותם expr (כלומר לא צריך pair<t1,t2>(...) אלא make_pair(...)).
- x < y: אם x.first < y.first או x.first >= y.first וגם x.second < y.second.

עוד Map:

- lower_bound: מקבל key ומחזיר איטרטור לאיבר הראשון עם מפתח גדול או שווה ל-key.
- upper_bound: מקבל key ומחזיר איטרטור לאיבר הראשון עם מפתח גדול ממש מ-key.
- equal_range: מחזיר זוג איטרטורים שהם lower, upper.

STL Algorithms:

רובם לא מבצעים שינוי באלגוריתם ולכן מקבלים const iterator, ולא ניתן לבצע cast מ-iterator ל-const iterator. יתרון: שימוש באיטרטורים, לא צריך גישה למיכל עצמו.

פונקציית remove: **remove(v.begin(), v.end(), t)** – "מעיף" את כל האלמנטים בעלי ערך t ע"י צמצום כל האלמנטים לכיוון ההתחלה תוך דריסת האלמנטים בעלי ערך t. לא משנה את גודל המיכל, מחזיר איטרטור ל-end החדש לאחר צמצום. לא עובד על associative containers.

שימוש בפרדיקט לדוגמא: struct myPred { bool operator()(int x) const { return <some Boolean expression>; } והשימוש בו למשל עבור וקטור v: vector<int>::iterator it = find_if(v.begin(), v.end(), **myPred()**); - מחזיר איטרטור התחלה לוקטור חדש בו איברים מ-v שמקיימים את הפרדיקט myPred.

Design Patterns:

- Abstract Factory (creational, object)
- Prototype (creational, object)
- Memento (behavioral, object)
- Interpreter (behavioral, class)
- Chain of Responsibility (behavioral, object)
- Adapter (structural, class)
- Singleton (creational, object)
- Factory Method (creational, class)
- Observer (behavioral, object)
- Strategy (behavioral, object)
- Bridge (structural, object)
- Proxy (structural, object)
- Decorator (structural, object)

(1) Singleton:

הבעיה: רוצים לאפשר יצירת מופע יחיד בלבד למחלקה מסויימת.

הפתרון: מגדירים את הבנאי להיות private או protected, ומחזיקים מופע יחיד כמשתנה סטטי.

מימוש 1: reference based:

- מגדירים שדה private למופע המחלקה: **static Singleton sng;**
- מגדירים מתודה סטטית public שמחזירה reference למופע הסטטי: **static Singleton& getObj() {return sng;}** – מחזירה ref כדי למנוע שימוש ב-copy CTOR.
- מגדירים private CTOR וכמו כן private copy CTOR, operator= כדי למנוע copy CTOR והשמה.
- יצירת מופע יחיד סטטי: בקובץ cpp מגדירים **Singleton Singleton::sng(3);** (למשל) – אתחול יחיד. בתחילת main כל המשתנים הסטטיים והגלובליים כבר מוגדרים.

מימוש 2: pointer based:

כמו המימוש הראשון עם שינוי:

- במקום **Singleton sng** מחזיקים **Singleton* sngPtr**.
- במקום יש מתודה public: **static Singleton* CreateObj()** שבודקת האם sngPtr==0 (לא אותחל). אם כן, מאתחלת. בכל מקרה בסוף מחזירה את sngPtr. האתחול ל-sngPtr בקובץ ה-cpp יהיה עם הערך 0 (ללא new, פשוט הפעלת constructor עם 0).
- ה-CTOR, copy CTOR, operator= עדיין private.
- מתודה נוספת public: **static void Destroy(){ delete sngPtr; sngPtr = 0; };**

המימוש הראשון יותר בטוח: התוכנית בונה את האובייקט ועובדים עם ref. במימוש השני ניתן ליצור עותקים של המצביע ואז לעשות עליהם delete.

המימוש השני יותר יעיל: יוצרים את האובייקט רק כשרוצים, והורסים אותו כשלא צריכים (Destroty), במקום שיחיה לאורך כל חיי התוכנית.

(2) Factory:

הבעיה: רוצים ליצור מופע חדש של אובייקט מסוג מסויים (ממבחר טיפוסים שכולם יורשים מטיפוס אחד משותף) ע"י העברה של שם הטיפוס כמשתנה.

הפתרון: בטיפוס המשותף ממנו יורשים יוצרים פונקציית "בית חרושת" שמקבלת את סוג הטיפוס המבוקש ומחזירה מופע חדש של אותו טיפוס. אוכפים שיצירתם תהיה רק דרך פונקציה זו ע"י השמת בנאי הטיפוסים היורשים כ-private ומגדירים את טיפוס האב להיות friend לכל הצאצאים.

מימוש (דוגמא על Shape כאב, Circle כצאצאים):

- ב-Shape תהיה מתודה public: **static Shape* factory(string type)**
- ב-Circle מחלקה Shape תוגדר כ-friend: **class Circle{ friend class Shape; ...}**

- ב-Circle הבנאי יהיה פרטי.
- מימוש factory: יוצרים **Shape* ptr = 0** ואז עוברים ב-if על כל הטיפוסים האפשריים (**if (type == "Circle") ptr = new Circle;**...) ובסוף מחזירים את ptr.

יתרונות: קל להוסיף צורות חדשות (תתי-טיפוסים חדשים), וניתן להוסיף מונה סטטי לעקוב אחר כמה מופעים נוצרו מכל טיפוס, או לממש GC ע"י החזקת וקטור של כל המופעים שנוצרו עד כה וניתן לשחררם.

חסרונות: האב צריך להיות friend של כל בניו ולהכיר את כל בניו כדי לכלול אותם ב-factory.

הערה: ניתן לממש את פונקציית ה-factory במחלקה נפרדת, ואז היא תהיה friend לבנים והיא תצטרך להכיר את כל הבנים, ולא מחלקת האב.

Observer (3) (subject-observer או publisher-subscriber):

נועד לניהול תלויות של אובייקט אחד באחר, כאשר התלוי הוא ה-observer וזה שתלויים בו הוא ה-subject. כאן נרצה מימוש שקושר כמה שפחות את השניים. חלקי המימוש:

- Subject: מחזיק רשימה של Observer, מספק ממשק ל-Attach, Detach של אובייקטי Observer אליו ומתודת Notify להפצת הודעה אליהם ע"י קריאה ל-Update של כל Observer.
- Concrete Subject: מחלקה כלשהי שתירש מ-Subject, מחזיקה במידע רלוונטי ל-Concrete Observer, והיא שולחת notification ל-Observers כשהמצב הרלוונטי משתנה (שינוי במידע הרלוונטי ל-Observer).
- Observer: הגדרת ממשק Update לעדכון כאשר ה-subject מודיע על שינוי.
- Concrete Observer: מחזיק ref ל-Concrete Subject, שומר מצב שצריך להיות קונסיסטנטי עם ה-Observer. מממש את מתודת ה-Update בשביל שמירה על קונסיסטנטיות.

מימוש:

Observer:

- בקובץ ה-h של Observer צריך להופיע forward declaration – **class Subject;** מבטיח ש-Subject יופיע בהמשך (אחרת נופל ב-link).
- מתודה וירטואלית לעדכון: **virtual void Update(Subject* ChngSubject) = 0;** - מחייב מימוש ליורשים.
- שדה protected: **Subject* sbj**.

Subject:

- בקובץ ה-h של Subject צריך להופיע: **class Observer;**
- שתי מתודות פומביות: **void Attach(Observer*); void Detach(Observer*);** - הוספה והסרה של Observer לרשימת המתעניינים.
- מתודה protected: **void Notify();**. מימושה ב-cpp: קריאה ל-Update של כל Observer ב-m_observers. ועליה לבצע **Update(this)**. היא protected כי רק הנושא צריך להפעילה, לא המתבוננים.
- שדה פרטי: **vector<Observer*> m_observers;** - להחזקת ה-Observer הרלוונטיים.

דוגמא למימוש קונקרטי: בנק ולקוחות:

Bank: ה-Subject

- יורש מ-Subject.
- בעל מתודה פומבית: **void ChangePercent();** - שינוי עליו צריך להודיע, במימוש תהיה קריאה ל-**Notify()**.

Account: ה-Observer

- יורש מ-Observer.
- CTOR: מקבל *Subject, במימוש שומר אותו בשדה הפרטי sbj (מ-Subject) ומבצע **sbj->Attch(this)** – מתחבר לבנק.
- DTOR: צריך להתנתק ע"י **sbj->Detach(this)**.
- מימוש מתודת העדכון: **void Update(Subject*)** – המימוש צריך לבדוק אם הארגומנט שניתן הוא sbj, ואם כן לבצע את פעולת העדכון (למשל להדפיס הודעה למסך שקיבל את העדכון).

הרחבות:

- ניתן להחזיק לכל Observer כמה Subject בוקטור, ובהפעלת update לבדוק ע"י RTTI או משתנה מיוחד ל-Subject (למשל שם ב-string) התאמה כדי להפעיל את ה-update המתאים.
- ניתן להעביר יותר מפרמטר ה-*Subject במתודת ה-Update (למשל להודיע על הודעת string כלשהו).
- הוספת סוג (למשל ע"י enum) ל-Observer והחזקת כמה וקטורים של Observer – וכך ניתן להודיע לפי קבוצות ולא לכולם תמיד.
- יתכן מצב בו עבור מחלקות A,B כל אחד הוא Observer של השני. פשוט ע"י ירושה מרובה.

Strategy (4):

מאפשר החלפת אלגוריתם בזמן ריצה, עבור קבוצת אלגוריתמים המממשת את אותו מנשק, שיופעלו על אותו אובייקט.

מחלקת Context:

- מספקת ממשק בעזרתו מספקים את האלגוריתמים, למשל ע"י הגדרת prototype כללי לפונקציה.
- דוגמא ל-ptototype: **typdef void(*fnc)()** – הגדרת fnc כטיפוס פונקציה שמקבלת void ומחזירה void. דוגמא נוספת: **typedef double(*fnc)(double)** – מקבלת double ומחזירה double.
- Context צריכה לספק מתודה להפעלת ארגומנט מסוג fnc: **void apply(fnc f) { f();}**.
- מימוש אפשרי: החזקת *Strategy כאשר הטיפוס הדינאמי יהיה concrete strategy כלשהו, והמתודות שלו יופעלו.
- מימוש אפשרי אחר: מחזיקים מערך של fnc ולפי הצורך בוחרים במימוש הנדרש.

מחלקת Strategy:

- מספק מנשק לכל ה-concrete strategies – אילו פונקציות עליהם לממש (מגדירים פונקציות אלו עם virtual ו-0=).

מחלקות concrete strategy:

- יורשות מ-Strategy ומספקות מימוש לפונקציות הנדרשות.

Bridge (5):

המטרה היא הפרדת design מאימפלמנטציה, למשל כדי לחסוך אימפלמנטציות שנגררות בגלל שינויים קטנים או לאפשר כמה מימושים ל-design. מאפשר יותר גמישות מאשר יצירת interface ומחלקות שממממשות אותו. מאפשר בחירת מימוש בזמן ריצה והחלפתו בזמן ריצה. מחזיקים היררכיה אחת ל-abstract interface והיררכיה אחרת ל-implementation. דוגמא למימוש על Date:

מחלקה אבסטרקטית Date:

- קובץ ה-h צריך להכיל forward declaration: **class DateImpl;**
- שדה protected: **DateImpl* m_Date** – מצביע לאובייקט מימוש ספציפי.
- מתודות וירטואליות שיקראו לאימפלמנטציה, למשל: **virtual void print()**. מימוש: קריאה ל-()m_Date->print.
- בנאי: בנאי של Date (עם int day, int month) קורא לבנאי של DateImpl עם פרמטרים אלו כדי לאתחל את m_Date. השדות הללו לא נשמרים ב-Date אלא רק ב-DateImpl. הבנאי צריך לקבל גם סוג impl למשל ע"י string (אפשר לשלב כאן factory שבהנתן string מחזיר DateImpl).

מחלקת מימוש DateImpl:

- הבנאי ייקרא ע"י הבנאי של Date.
- צריך לספק מתודה וירטואלית של מימוש המתודות שמספק Date: **virtual void print()**. כל מחלקת מימוש יורשת מ-DateImpl תממש מתודה זו לפי המימוש שלה.

Abstract Factory (6):

ריבוי מפעלים היורשים מ-abstract factory. כאשר כל מפעל מספק מימושים למשפחת מחלקות. דוגמא עבור InputDevice ו-OutputDevice:

מחלקות הבסיס ומחלקות ממשות:

- מחלקת InputDevice: בעלת מתודה וירטואלית **virtual void DisplayDevice() = 0**, נורשת ע"י מחלקות IBMKeyboard, SamsungMouse.
- מחלקת OutputDevice: בעלת מתודה וירטואלית זהה, נורשת ע"י מחלקות IBMPrinter, SamsungMonitor.

מחלקת ה-Factory האבסטרקטי:

- מתודה וירטואלית לבניית InputDevice: **virtual InputDevice* FInputDevice() = 0**
- כנ"ל עבור OutputDevice.

מחלקות Factory יורשות מהמפעל האבסטרקטי:

- מחלקת IBMFactory: מספק מימוש לפונקציות הבנייה במפעל האבסטרקטי, שמחזירות new IBMKeyboard או new IBMPrinter בהתאם.
- מחלקת SamsungFactory: באופן דומה מחזירה SamsungMouse ו-SamsungMonitor.

חסרונות: המפעל האבסטרקטי מכתיב למפעלים שיורשים ממנו אלו מחלקות ייצרו, ומחלקה יורשת לא יכולה להוסיף אובייקט חדש לייצור. אם מוסיפים מוצר למחלקה האבסטרקטית צריך להוסיף לה תמיכה בכל המחלקות היורשות.

Prototype (7):

מאפשר יצירת מופעים חדשים של מחלקה ע"י שכפול אובייקט אב-טיפוס על מנת לחסוך. דוגמא: הגדרת אלמנטים קבועים ב-GUI (כמו Layout). לעומת factory, כאן לא צריך היררכית מחלקות מקבילה להיררכית המחלקות של המוצרים.

כאן יש מחלקת אב שמספק ממשק למתודת clone המחזירה מצביע לטיפוס אותה מחלקה. כל מחלקה שיורשת תחזיר עותק של עצמה – אב טיפוס לאותו מימוש. כאן נזדקק למחלקת Creator שתחזיק את אבי הטיפוס. דוגמא על Shape (כמו הדוגמא ל-factory):

מחלקת Shape, מחלקות Circle ו-Square:

- מחלקת Shape: מספקת ממשק ל-clone: **virtual shape* clone() = 0**.
- מחלקות Circle, Square: מימוש clone להחזיר מצביע לאובייקט הבנוי ע"י copy CTOR לעצמי: **return new Circle(*this);**.

מחלקת Creator:

- מחזיקה **static shape* prototypes[2]** – מערך למופעים אבי הטיפוס של Circle, Square עליהם ייעשה ה-clone ע"י המשתמש.
- מתודה סטטית ליצירת מופע של ה-Shape הרצוי ע"י אינדקס במערך ה-prototypes: **static Shape* createShape(int i){ return prototypes[i]->clone(); }**
- מערך ה-prototypes צריך להיות מאותחל ל-{new Circle, new Square}.

שימוש בתוכנית משתמש: Shape* myShape = Creator::createShape(0) – יחזיר עותק של Circle.

Memento (8):

נועד לשמר תמונת מצב (memory snapshot) של מצב אובייקט. משמש לצורכי אפשרות חזרה למצב קודם, כמו בפעולת undo. דוגמא על תוכנית ציור השומרת מצב בכל משיכת מברשת: נעשה שימוש ב-FactoryStrokes לשמירת כל משיכות המברשת. לכל משיכה ישמר Memento של כל המשיכותעד אליה, כך שבהינתן Memento ניתן לשחזר את מצב הציור.

מחלקת Stroke וטיפוס memento_t:

- עבור struct Point { float x; float y; };, מחזיקה **Point beginPoint; Point endpoint** – הגדרת stroke יחיד.
- memento_t: מוגדר ב-FactoryStrokes כוקטור של Stroke.

מחלקת FactoryStrokes:

- כאן מוגדר memento_t ואיטרטור שלו it_t (וקטור של Stroke ואיטרטור שלו).
- **mementos_t**: וקטור של memeto_t* - מחזיק את כל ה-memento_t לאורך התוכנית (וקטור של וקטורים). **mit_t**: איטרטור שלו, **mrit_t**: איטרטור reverse שלו.
- מתודת הוספת stroke: **void AddStroke(Stroke* lastStroke)**: לוקחים את m_mementos.rbegin() (מסוג mrit) וממנו לוקחים את התוכן (הוצאת תוכן מאיטרטור – ע"י *) – ה-memento_t האחרון שנשמר. מאתחלים memento_t חדש באותו גודל ומעתיקים אותו ל-memento_t החדש. בסוף מבצעים **newMemento.push_back(lastStroke)** – שמירת המשיכה החדשה, ואת **&newMemento** דוחפים ל-m_mementos.
- מתודת החזרת memento_t: **memento_t* getAllStrokes()**: מחזירים את **(*m_mementos.rbegin())** – הממנטו האחרון שנשמר.

- מתודת undo: **void Undo()** – אם m_mementos ריק, חוזרים. אחרת פשוט מוחקים מ-m_mementos את האיבר האחרון וכך מסירים את ה-memento האחרון (עושים erase ל-**(m_mementos.end()--)**). תמיכה ב-redo: שימוש ב-remove ולא ב-erase.

Interpreter (9):

בהינתן שפה, מגדירים לה ייצוג ו-interpreter שמשתמש בו כדי לפרש משפטים בשפה. למשל: חיפוש לפי ביטוי רגולרי (המגדיר שפה). ה-DP הזה מתאר איך להגדיר דקדוק לשפה, לייצג משפטים בשפה ולפרש אותם. אין תבנית אחידה במקרה זה, תלוי שפה. חלקים:

Abstract Expression:

- מגדיר פעולת interpretation כללית ב-AST. יורשים ממנו:
  - Terminal expression: פירוש לטרמינל (סימן בא"ב של השפה), קיים אחד כזה לכל תו בשפה.
  - Nonterminal expression: נדרש אחד לכל חוק גזירה בשפה מהצורה R::=R1 R2 ... Rn כאשר כל אחד מהחלקים הוא AbstractExpression.

Context: מכיל מידע גלובלי למפרש.

Client:

- נותן (או מקבל) AST המייצג משפט בשפה המוגדרת ע"י הדקדוק הנ"ל. ה-AST מורכב ממופעים של terminals, nonterminals.
- מפעיל את פעולת ה-interpret.

Proxy (10):

השמת אובייקט חוצץ (proxy) בין אובייקט שבנייתו כבדה ובין המשתמש. המשתמש עובד מול אובייקט ה-proxy ולא מול האובייקט האמיתי, ולא מודע למתי יש פניה מאובייקט ה-proxy לאובייקט הכבד. סוגים:

- Remote proxy: ייצוג מקומי של אובייקט במרחב כתובות אחר. עותק אחר לחלוטין של האובייקט.
- Virtual proxy: בונים אובייקט "יקר" רק בדרישה, אחרת עובדים עם אובייקט קטן.
- Protection proxy: בקרה על גישה לאובייקט וניהול הרשאות על האובייקט.
- Smart reference: האובייקט הכבד נבנה פעם אחת ומנהלים לו smart pointers עם מונה למספר ההפניות, יכולת לשחרר אובייקט כאשר אין לו יותר הפניות, ניהול נעילות על האובייקט כך שלא ישנו אותו כאשר מישהו אחר עובד עליו.

מרכיבים:

- Subject: מנשק תואם ל-RealSubject ול-Proxy כך שה-Proxy יוכל לשמש מול המשתמש כ-RealSubject.
- RealSubject: האובייקט האמיתי שמיוצג ע"י ה-proxy. המשתמש **לא יכול** ליצור אותו, אלא רק לעבוד מול proxy.
- Proxy: המשתמש עובד מולו כמו מול ה-RealSubject, ה-Proxy מנהל את הגישה לאובייקט האמיתי. מפנה בקשות לאובייקט האמיתי בהתאם לסוג ה-proxy.

דוגמא:

- Image: האובייקט האמיתי, בעל מתודת Draw. מתודת **extern Image* LoadImageFile(const char*)** מחזירה אובייקט תמונה כזה, והוא אובייקט כבד.
- ProxyImage: מקבלת בבנאי *const char (כתובת התמונה). בעל מתודת **Image* LoadImage();** שבפעם הראשונה שנקראת יוצרת את אובייקט ה-Image האמיתי, ע"י קריאה ל-LoadImageFile של מחלקת Image (המתודה ה-extern לעיל) וכל שאר הפעמים מחזירה את המצביע הזה. דורסת שני אופרטורים:
  - אופרטור <-: **virtual Image* operator->() {return LoadImage();}** – כאילו <- מה-proxy הוא <- לאובייקט האמיתי, שבגישה ראשונה נוצר.
  - אופרטור *: **virtual Image operator*() {return *LoadImage();}** – אותו דבר, רק עם dereferencing.

Decorator:

הוספת פונקציונאליות לאובייקט באופן דינאמי בזמן ריצה, כחלופה לירושה. מאפשר הוספת פונקציונאליות לאובייקט יחיד ולא למחלקה שלמה.

מרכיבים:

- Component: מנשק לאובייקטים שרוצים להרחיב להם את הפונקציונאליות באופן דינאמי.
- ConcreteComponent: אובייקט קונקרטי שרוצים להרחיב לו את הפונקציונאליות (יממש את Component).
- Decorator: מחזיק גישה לאובייקטי Component ומגדיר מנשק בהתאם למנשק של Component.
- ConcreteDecorator: מממש את Decorator.

Chain of Responsibilities:

יצירת שרשרת אובייקטים המקבלים בקשה ומטפלים בה, על מנת למנוע קישור בין שולח הבקשה למקבלה. ניתן להשתמש בשיטה זו כאשר יתכן יותר מאובייקט אחד לטיפול בבקשה, וה-handler לא ידוע a-priori (למשל נקבע רק בזמן ריצה). שרשרת האובייקטים המטפלים תקבע דינאמית.

מרכיבים:

- Handler: מנשק לטיפול בבקשה.
- ConcreteHandler: מטפל בבקשה עליה אחראי, ובעל גישה ל-successor שהוא גם מסוג Handler.
- Client: שולח את הבקשה ל-ConcreteHandler בשרשרת.

Adapter:

יצירת מנשק עבור קבוצת מחלקות ע"מ להתאימן לאובייקטים אחרים שאין להם היכרות עמם. בפועל עוטפים פונקציונאליות במנשק אחר כדי לאפשר נגישות למחלקות שאין להן גישה ישירה לפונקציונאליות המקורית.

<u>הערות מעשיית מבחנים</u>:

- במקרה של disambiguation על overloaded functions, למשל על max(5, 3.14) כשל-max גרסת double וגרסת int – יזרק compilation error על disambiguation.
- עבור אובייקט X ואובייקט Y שיורש מ-X, אם נכניס את Y ל-vector<X>, הוא יכנס כ-X ויאבד את התכונות ה-"Y"יות שלו.
- <u>בכתיבת קוד</u>:
  - לשים לב ל-const correctness – אם מתודה **לא** משנה אובייקט, להוסיף בסופה const.
  - לעבור על כל האופרטורים ואופן הגדרתם.
  - Static לא יכול להיות virtual.
  - לא ניתן להגדיר במחלקה שלנו friend virtual – כי אותה מתודה היא mem-func של מישהו אחר (אולי שם היא virtual).
- ב-virtual: ה-vt נוצר לכל טיפוס, ויחידה לכל מופע של אותו טיפוס (לכל מופעי ה-Base טבלה יחידה, לכל מופעי Derived טבלה יחידה...).
- אם במהלך זריקת exception נזרק עוד אחד (למשל בגלל קריאה אוטומטית של DTOR של אובייקט בבלוק ה-try, כאשר ה-DTOR זורק exception) – הוא נתפס ע"י מערכת ההפעלה שהורגת את התוכנית. לעטוף את זה בבלוק try-catch נוסף מתברר כלא עובד...
- אם יש רק DTOR, הקומפיילר עדיין כן מייצר default CTOR וכו'.
- עבור int*: **const int*** - לא ניתן לשנות את <u>תוכן המצביע</u> (אם למשל זה מערך, לא ניתן לבצע השמות למערך כמו a[0] = 2 וכו'). **int* const** – לא ניתן לשנות את <u>המצביע עצמו</u> אך כן ניתן את התוכן.
- בשאלות בדיקת קוד, כנראה שמצופה מאיתנו לוודא שה-**DTOR יהיה virtual** (אחרת זה כנראה design error כי מצופה לתמוך בפולימורפיזם).
- תמיד לשים לב אם מצביע כלשהו משמש כמערך (למשל int* a ישמש למערך של int) – ואז צריך **[]delete** ולא **delete**.
- מתודה שהיא **pure virtual ניתנת למימוש במחלקת הבסיס**. פשוט אם יש מחלקה יורשת ממנה, היא תחוייב לעשות לה override.
- מתודה שהיא **pure virtual**, לפי ההגדרה של יוסי, **מחייבת לבצע הורשה ע"י מחלקה אחרת ממחלקת הבסיס המכילה מתודה זו**. כלומר נניח בשאלה כזו שכוונתו היא: מה צריך לעשות אם רוצים להשתמש במחלקה / במתודה. <u>אי אפשר</u> לבצע אינסטנציה למחלקת בסיס שיש לה מתודה **pure virtual**, אפשר רק ל-derived שלה, והיא <u>חייבת</u> לממש פונקציות (אחרת גם היא abstract ולא ניתן לבצע לה אינסטנציה).
- אם ב-Derived מתודה **virtual foo** היא private / protected וב-Base היא public, אז עבור Base* a = new Derived() **הגישה ל-a->foo() ניגשת ל-foo של Derived אך מקבלת את הרשאות Base, כלומר כאילו היא public**.
- ברגע שעושים dereference מתקבל משהו מהטיפוס הסטטי. למשל עבור A ו-B שיורשת מ-A, עבור A* a = new B(); הטיפוס של a* הוא **A**.
- אם יש פניה לפונקציה ב-**CTOR** אותה מחלקה **חייבת** שיהיה לה מימוש לאותה פונקציה.
- אם למחלקת Base אין default CTOR **זה לא מונע** מ-Deriv לרשת ממנה (כל עוד לא משתמשת, גם לא implicit, ב-Base default CTOR).

סיכום overloading operators:

- virtual **const** T& operator=(const T& t) {... **return *this;** }
- virtual bool operator==(const T& t) **const**;
- virtual bool operator<(const T& t) **const**;
- **friend** ostream& operator<<(ostream& os, const T& t) {... return os; } // only os changes, so t is const
- **friend** istream& operator>>(istream& is, T& t) {... return is; } // here also t **is** changed
- **friend T** operator+(const T& t1, const T& t2);
- virtual T& operator++(); // for "t++"
- virtual **T** operator++(**int**); // for "++t" ?
- virtual T& operator[](unsigned index); // assign
- virtual **T** operator[](unsigned index) **const**; // retrieve
- virtual **X** operator **X()** const; // casting to type X