

נקודות חשובות בנושאים מתקדמים בתכנות

נכון או לא נכון: בתוכנית ++C, הקוד שנמצא בפונקציה main הוא הראשון לרוץ? **לא נכון!** בנאים של אובייקטים יכולים להיקרא אפילו לפני main (1-34)

אובייקט גלובלי או סטטי נהרס בסוף main או בקריאה ל()exit.

שחרור של מצביע ל-NULL באמצעות free היא פעולה חוקית ב-C++.

בתוך סקופ ספציפי, DTORS נקראים בסדר הפוך מה-CTORS (1-42).

הפקודה: Person p3=p1; אינה פעולת השמה אלא קריאה ל-Copy CTOR.

ברגע שבמחלקה יש פוינט כ-Data Member (למשל char*) עלינו לממש בנאי ברירת מחדל, DTOR, Copy CTOR ואופרטור השמה.

בעת העברת משתנים by-value בפונקציה, ++C קורא ל-Copy CTOR. אם המשתנה הוא פרמטר של הפונקציה, הקריאה ל-Copy CTOR תהיה בכניסה לפונקציה. אם המשתנה הוא ערך מוחזר, הקריאה תתבצע בשורת return.

סדר ביצוע רשימת אתחול הוא לפי השדר בו מופיעים השדות באובייקט ולא לפי הסדר בו מופיעה רשימת האתחול.

במקרים בהם יש משתנה const שאמור להיות מאותחל בעת אתחול המחלקה או במקרה בו יש שדה של מחלקה ללא בנאי ברירת מחדל או במקרה בו שדה של המחלקה הוא מסוג Reference, חייבים להשתמש ברשימת אתחולים.

אם למחלקה X קיים אופרטור המרה לטיפוס Y וגם קיים לה בנאי המקבל משתנה מטיפוס Y, הפקודה y(X) תתורגם ל-y(X) (כי זו דרך נוספת לקרוא להמרה). כיוון שיש גם בנאי המקבל טיפוס Y, במקום לבצע המרה, תתבצע קריאה לבנאי זה. לכן, כדי לקרוא להמרה, נצטרך לעשות זאת במפורש: y.operatorX();

לא ניתן לבצע המרה ל- [sizeof], [], [?:], [.*], [::].

אם יש לנו מחלקה Y ובה בנאי עם פרמטר אחד מסוג X, הבנאי עשוי לתפקד גם כאופרטור המרה! כדי למנוע זאת, עלינו להשתמש במילת המפתח explicit לפני חתימת הבנאי.

New operator – לא ניתן לדריסה. נקרא כאשר אנו כותבים את הפקודה: new C(); בעת כתיבת הפקודה, מתבצעת הקצאת זיכרון ע"י operator new (שאותו כן ניתן לדרוס) ואז מתבצעת קריאה לבנאי של הטיפוס C.

Placement New – מאפשר לבצע new על מקום שהוקצה כבר בזיכרון (X = new(mem) X; כאשר mem הוא מצביע לזיכרון שהוקצה כבר ב-Heap ע"י (למשל); char* mem = new char[128];). נשים לב שכיוון שלא השתמשנו ב-new רגיל, לא נשתמש ב-delete רגיל אלא נקרא באופן מפורש ל-DTOR של X (שקופית 144). בסוף הקצאת הזיכרון נשתמש ב-[delete] (כי הקצאת הזיכרון עם new הייתה למערך).

בפונקציות סטטיות לא ניתן להוסיף const לסוף החתימה של הפונקציה (כיוון שהוא מתייחס לאי שינוי המופע עליו מופעלת הפונקציה אבל פונקציה סטטית אינה קשורה למופע).

בהורשה, כדי לפנות לפונקציה ממחלקת בסיס, נשתמש בסינטקס: x.baseClass::foo();

למרות שהקומפילר לא דורש זאת, בכל מחלקה שיש בה לפחות פונקציה וירטואלית אחת, ה-DTOR חייב להיות וירטואלי.

שינוי Access של Virtual Member Functions אינו אפשרי באמת. כלומר, אם הגדרנו פונקציה foo עם הרשאת public בבסיס, לא משנה איזה הרשאה ניתן לה במחלקה יורשת, הרשאת הגישה תישאר public: אם ננסה ליצור אובייקט מסוג הטיפוס היורש, קריאה ישירה לפונקציה foo לא תתאפשר (כלומר אי אפשר לעשות d.foo()) כי היא לא מוגדרת public. עם זאת, אם נעשה cast למצביע של d להיות מצביע לאובייקט מסוג הבסיס, נוכל לקרוא לפונקציה foo (כיוון שבבסיס היא מוגדרת להיות public). בנוסף, כיוון שזו פונקציה וירטואלית, בפועל, תקרא פונקצית foo של הבסיס.

נשים לב שניתן לכתוב מימוש של פונקציה וירטואלית טהורה במחלקת הבסיס. למימוש זה ניתן לקרוא בכל הפונקציות האחרות במחלקה וכן בפונקציות של מחלקות יורשות (ע"י (base::foo()).

Friendship היא לא תכונה טרנזיטיבית ולא ניתן לרשת אותה (כלומר חבר שלי הוא לא בהכרח חבר של ילדיי או חבריי האחרים).

u.multinet.co.il

בעת זריקת חריג, קוד בפונקציות שנמצאות "בדרך" ל-catch לא יבוצע. כלומר, לאחר ה-throw יבוצע ישר ה-catch. לכן, משתנים שנמצאים על ה-stack ישוחררו אך משתנים שנמצאים על ה-heap לא ישוחררו.

בעת כניסה ל-catch מתבצע slicing. כלומר, הפרמטר שנזרק מועתק לפרמטר מסוג base שאמור לתפוס ה-catch (בהנחה שהוא יורש מה-base כמובן). לכן, אם נריץ בתוך אותו catch את הפקודה; throw ex; תיזרק שגיאה עם אובייקט מסוג base ולא מהטיפוס המקורי. לכן, עלינו להשתמש בפקודה; throw (ללא ציון שם האובייקט) כדי שכך ייזרק האובייקט המקורי.

כדי שיהיה אפשר להשתמש ב-STL, אובייקט T חייב לממש מספר תכונות: עליו לממש בנאי ברירת מחדל, בנאי העתקה, אופרטור -, אופרטור < ואופרטור ==.

אם הקיבולת של vector השתנתה כתוצאה מפעולת הוספה, כל האיטרטורים הופכים לבלתי שמישים. אם הקיבולת לא השתנתה, פעולות הוספה/מחיקה מבטלים את כל האיטרטורים שנמצאים לאחר המיקום אליו הוספנו/מחקנו.

ב-deque הוספה של פריטים (בכל מקום) או מחיקה מהאמצע מבטלת את כל האיטרטורים. מחיקה מההתחלה/הסוף מבטלת את איטרטור ההתחלה/הסוף.

ב-list, map, set איטרטורים שנמצאים על פריטים שנמחקו הופכים ללא שמישים. הוספה אינה מבטלת איטרטורים.

הקומפיילר מספק לכל טיפוס בנאי ו-DTOR ברירת מחדל, בנאי העתקה, אופרטור השמה, אופרטור new, אופרטור delete, אופרטור -> ואופרטור *.

```
typedef void (Card::*ptrToMemFunc)(void);
```

דרישות מנמליות ממחלקה עבור STL – בנאי ברירת מחדל, בנאי העתקה, אופרטור השמה, אופרטור < ואופרטור ==.

בוא נשחק: מה חסר בקוד שלי?

1. לבדוק האם במחלקה קיים לפחות Data Member אחד שהוא פוינטר. אם כן, לוודא שקיים Default CTOR, Copy Operator=, DTOR, CTOR.
2. לבדוק שכל ה-Getterים מחזירים משתנה שהוא const.
3. לבדוק שכל ה-Getterים הם פונקציות const.
4. לבדוק שעל new של מערך יש [] delete (וכל שאר הנגזרות).
5. לוודא שלא עושים פעמיים delete על אותו פוינטר (מבלי להפוך אותו ל-0).
6. לוודא שלא חסרים const בפרמטרים של פונקציות.
7. אם יש אתחול של מערך של אובייקטים, יש לוודא שלאובייקט הספציפי קיים Default CTOR.
8. לבדוק בהשמה שאכן ההשמה חוקית (כלומר קיים אופרטור השמה מתאים).
9. אם יש פולימורפיזם (ואולי גם אם לא) לוודא שה-DTOR הוא virtual.

Operator Overloading

```
Const Person& operator=(const Person& P);
friend ostream& operator<<(ostream& os, const Person& p);
friend istream& operator>>(istream& os, Person& p);
bool operator==(const Person& p) const;
operator int() const - המרה ל-int. לשים לב שהטיפוס המוחזר מופיע לאחר המילה operator.
assign - Person& operator[](unsigned i);
retrieve - Person operator[](unsigned i) const;
void* operator new(size_t size);
void operator delete(void * ptr);
```

Singleton – רוצים לאפשר יצירת מופע אחד בלבד של אובייקט מסוג מסוים.

Factory Method – רוצים ליצור מופע חדש של אובייקט מסוג מסוים (כל סוגי האובייקטים יורשים מטיפוס משותף), ע"י העברה של שם הטיפוס כמשתנה.

Observer – רוצים לעדכן אובייקטים מסוימים (Observers) על שינויים באובייקט כלשהו (Subject) מבלי שיצטרכו לדגום עבור שינויים כל פרק זמן מוגדר.

Strategy – מאפשר להחליף אלגוריתם בזמן ריצה (יוצרים מערך של פונקציות ופונים לפונקציה דרך המערך ע"פ הסוג שנקבע).

Bridge – מטרתו לעשות הפרדה מוחלטת בין עיצוב מחלקה למימוש שלה (בדוגמת Date: יוצרים מחלקה Date עם פוינטר ל-DateImpl. בבנאי של Date מקבלים שם של אימפלמנטציה ולפיה בונים אובייקט עליו יצביע DateImpl. שאר הפונקציות ב-Date פונות לפונקציות ב-DateImpl. כל המימושים של Date יורשים מ-DateImpl שמגדיר ממשק אחיד).

Abstract Factory – מטרתו לקבץ מספר סוגים של אובייקטים יחד. כלומר, זהו בית חרושת של בתי חרושת. למשל, יוצרים בית חרושת המחזיר בית חרושת של A, B וכו' ויודע להגדיר סוגים שונים של מוצרים (רכיב קלט, רכיב פלט וכו'). בית חרושת A יכול לייצר מקלדת ומסך ובית חרושת B מייצר עכבר ומדפסת. הרכיב עצמו לא משנה אלא הפונקציונאליות שלו. בית החרושת מגדיר מקבץ של רכיבים המיוצרים יחד.

Prototype – מאפשר בזמן ריצה לבנות אובייקט שבלעדיו היינו מבזבזים זמן רב כדי לבנות. כלומר, בונים "אב טיפוס" של האובייקט מראש ובכל פעם שזקוקים לאובייקט כזה פשוט משכפלים אב טיפוס זה (תחת ההנחה ששכפול האובייקט זול יותר מבניית אובייקט חדש).

Memento – נועד לצלם מצב של אובייקט. יוצר סרט של על המצבים של אותו אובייקט. בנוי מוקטור A של וקטורים Bi. וקטור A מכיל בעצם את כל המצבים. כל רצף מצבים הוא למעשה וקטור Bi כלשהו בו מופיעות כל הפעולות שבוצעו על האובייקט לפי הסדר.

Interpreter – נועד לעשותו פירוש למחרוזת ע"י parser שמבצע עליה אנליזה (למשל Interpreter לביטויים רגולרים או לתרגום מספר רומי למספר עשרוני).

Proxy – שימושי כאשר קיים אובייקט גדול (למשל תמונה) שברצוננו לעשות עליו פעולות שלא דורשות טעינה של כל האובייקט (למשל לברר מה גודלו). בונים מחלקה נוספת אשר מתווכת בין הלקוח לאובייקט האמיתי (עבור הלקוח היא מוצגת כאובייקט האמיתי) ופונה לאובייקט האמיתי רק במקרה שבאמת צריך אותו. (כנראה) ניתן לשלב את ה-Proxy גם באובייקט המקורי. בנוסף, ה-Proxy יכול לבצע בדיקה של הרשאות בטרם פנייה לאובייקט אמיתי.

Decorator – נועד כדי לאפשר להוסיף פונקציונאליות לאובייקטים בזמן ריצה. למשל, עבור ממשק GUI, נרצה לאפשר יצירה של מחלקה אחת מסוג כפתור ולהוסיף מאפיינים נוספים (כמו צל, גבול וכו') ע"פ דרישה. עושים זאת ע"י הגדרה של מחלקת בסיס אשר מגדירה ממשק משותף. כל אובייקט (סרגל, כפתור וכו') יוגדר להיות Visual Component. גם מחלקות ה-Decorator יהיו VC אך יכללו בתוכם פוינטר ל-VC נוסף. כך יוצרים בעצם VC אחד בתוך השני כאשר כל שכבה מוסיפה פונקציונאליות. פעולה על VC, תחלחל פנימה אל שאר ה-VC (ובדרך יבוצעו פעולות נוספות על פי הצורך).

Chain of Responsibilities – מאפשר העברת טיפול בבקשה לאובייקט אחר שרק אותו צריך להכיר, גם אם בפועל לא הוא זה שאמור לטפל בבקשה. יוצרים "שרשרת" של Handlers. כל Handler בשרשרת בודק האם הוא יכול לטפל בבקשה שמגיעה אליו. אם לא, הבקשה עוברת הלאה ל-Handler הבא בשרשרת. אין הבטחה שהבקשה אכן תטופל.

Adapter – מאפשר לחבר בין 2 אובייקטים אשר בעיקרון היו יכולים לתקשר אך יש אי התאמה בממשק שלהם (בדומה למכשיר חשמלי בחו"ל). שונה מ-Bridge אשר דורש ליצור אובייקט אחר בכל פעם (כאן אין צורך ביצירת אובייקט אחר).