מחלקות, בניה ופירוק

המחלקות בשפת ++C הן שילוב והרחבה של המבנים הקיימים ב-C וב-Java.

בשפת C, הדרך המקובלת ליצור מבנים מורכבים היא להגדיר struct. המשמעות של struct היא פשוט אוסף של משתנים שנמצאים ברצף בזיכרון. אפשר גם להגדיר משתנים מורכבים יותר ע"י struct בתוך struct וכו'. אבל ב-struct אין שיטות - אם רוצים לעשות איתו משהו צריך לכתוב שיטות חיצוניות.

בשפת Java יש מחלקות (class) שאפשר לשים בהן גם משתנים וגם שיטות - זה אמור להפוך את הקוד לקריא יותר כי כל המידע והפעולות הדרושות למימוש העצם נמצאים במקום אחד. אבל, בניגוד למבנים של סי, בג'אבה המשתנים לא תמיד נמצאים ברצף בזיכרון. לדוגמה, נניח שאנחנו מגדירים מבנה של "נקודה" ובו שני מספרים שלמים, ואז מגדירים מבנה של "משולש" ובו שלוש נקודות. בסי, כל משולש כולל פשוט שישה מספרים הנמצאים ברצף בזיכרון, ותופס 24 בתים בדיוק. בג'אבה, כל משולש כולל שלושה **מצביעים** (pointers), כל אחד מהם מצביע לנקודה שבה יש שני מספרים. כלומר המספרים לא נמצאים ברצף בזיכרון. למה זה משנה?

- בלי מצביעים, התוכנה תופסת פחות זיכרון ודורשת פחות זמן כשניגשים למשתנים.
- כשכל המבנה נמצא באותו מקום בזיכרון, ניהול הזיכרון מהיר יותר. בפרט, כשמשתמשים בזכרון מטמון (cache), זה מאד יעיל שכל המבנה נטען בבת-אחת לאותו בלוק בזיכרון.

שפת ++C משלבת את היתרונות של שתי השפות:

- יש בה struct כמו ב-C, אבל אפשר לשים בו גם שיטות.
- יש בה class כמו ב-Java, אבל המשתנים נשמרים ברצף בזיכרון ולא ע"י פוינטרים (נראה בהמשך).

למעשה, ב-++C, ההבדל היחיד בין struct לבין class הוא בהרשאות הגישה: בראשון, כברירת מחדל, הרשאת הגישה היא public; בשני, כברירת מחדל, הרשאת הגישה היא private. מכאן שההגדרות הבאות שקולות לחלוטין (ראו דוגמה בתיקיה 1):

```
struct X {
    private:
          ...
}
===
class X {
          ...
}
```

וכן ההגדרות הבאות שקולות לחלוטין:

```
struct X {
          ...
}
```

```
===
class X {
     public:
          ...
}
```

דרכים למימוש שיטות

יש שתי דרכים לממש שיטות של מחלקות ב-++C.

- דרך אחת היא לממש אותן ישירות בתוך המחלקה (בדומה ל Java). זה נקרא מימוש inline ודומה לפונקציות inline שראינו בשיעור הקודם, רק שכאן לא צריך לכתוב את המילה inline.
- דרך שניה היא לשים בתוך המחלקה רק **הצהרה** של השיטה, בלי גוף. את המימוש עצמו שמים מחוץ למחלקה (באותו קובץ או בקובץ אחר). ראו דוגמה בתיקיה 2.

המצביע this

במימוש של שיטה, המילה השמורה this מכילה מצביע לעצם הנוכחי. ניתן להשתמש בה כמו שמשתמשים במצביעים, למשל:

```
Point::Point(int x, int y) {
      this->x = x;
      this->y = y;
}
```

(זה דומה לג'אבה פרט לכך שמשתמשים בחץ במקום בנקודה).

שדות סטטיים

ניתן להגדיר שדות ושיטות סטטיים (שייכים לכל המחלקה ולא לעצם מסויים) בעזרת המילה השמורה static. כדי לגשת אליהם מבחוץ, מקדימים להם את שם המחלקה ופעמיים נקודתיים, למשל Point::MAXX. אם מגדירים שדה סטטי שהוא גם קבוע (const), ניתן לאתחל אותו בהגדרת המחלקה; אחרת, יש לאתחל אותו מבחוץ (ראו דוגמה בתיקיה 3).

קבצי כותרת למחלקות

מקובל (אם כי לא חובה) להגדיר כל מחלקה בשני קבצים (ראו דוגמה בתיקיה 4):

- קובץ הכותרת - בדרך-כלל עם סיומת hpp או h - כולל את הגדרת המחלקה, השדות והשיטות שלה - אבל בלי מימוש השיטות.
- קובץ התוכן - בדרך-כלל עם סיומת cpp - כולל את המימוש של השיטות.

עקרונית, ניתן לממש את כל השיטות בקובץ הכותרת (כמו בג'אבה), אבל זה לא כדאי. מדוע? כמה סיבות:

1. הנדסת תוכנה. אם ניתן את המחלקה שלנו למישהו אחר, הוא ירצה לראות איזה שיטות יש בה, אבל לא יעניין אותו לדעת איך בדיוק מימשנו אותן. לכן הוא יסתכל בקובץ הכותרת, ועדיף שהקובץ הזה יהיה קטן ופשוט ככל האפשר.
2. זמן קומפילציה. במערכות תוכנה מורכבות, קומפילציה לוקחת הרבה זמן. בכל פעם שמשנים קובץ, צריך לקמפל מחדש את כל הקבצים שתלויים בו. כששמים את המימוש בקובץ נפרד, שינוי במימוש לא דורש קימפול מחדש של קוד שמשתמש בקובץ הכותרת. כדוגמה ראו את ה-Makefile בתיקיה 4.

השימוש בקבצי-כותרת פותר את הבעיה באופן חלקי בלבד. מדוע? כי המשתנים הפרטיים - שהם חלק מהמימוש - עדיין נמצאים בקובץ הכותרת. המשתמשים של המחלקה שלנו רואים אותם, ואם נשנה אותם נצטרך לבנות את כל הפרוייקט מחדש. יש לזה פתרונות - נראה בהמשך.

בנאים - constructors

בנאי הוא שיטה ששמה כשם המחלקה, ואין לה ערך-חזרה. כמו כל שיטה אחרת, ניתן לממש אותה בקובץ הכותרת או בקובץ המימוש. ראו דוגמאות בתיקיה 4.

כמו כל פונקציה, גם בנאים אפשר להעמיס. כלומר, אפשר להגדיר כמה בנאים עם פרמטרים שונים, והקומפיילר יקרא לבנאי הנכון לפי השימוש.

איך קוראים לבנאי? תלוי:

- אם זה בנאי עם פרמטרים, פשוט שמים את הפרמטרים אחרי שם העצם, למשל Point; p(10,20) קורא לבנאי של Point המקבל שני מספרים שלמים.
- אם זה בנאי בלי פרמטרים, לא שמים שום דבר אחרי שם העצם, למשל Point p; (לא לשים סוגריים ריקים - זה יגרום לשגיאת קימפול כי הקומפיילר עלול לחשוב שאתם מנסים להגדיר פונקציה...)

שימו לב - בניגוד לג'אבה - לא צריך להשתמש ב-new! המשתנה נוצר "על המחסנית" ולא "על הערימה" (אם נשתמש ב-new, המשתנה ייווצר על הערימה).

בנאי ללא פרמטרים

יש כמה סוגי בנאים שיש להם תפקיד מיוחד ב-++C. אחד מהם הוא **בנאי ללא פרמטרים** (parameterless constructor). אם (ורק אם) לא מגדירים בנאי למחלקה - הקומפיילר אוטומטית יוצר עבורה בנאי כזה; הוא נקרא default parameterless constructor. בהתאם לגישה של ++C "לא השתמשת - לא שילמת", בנאי ברירת-מחדל של מחלקה פשוטה לא עושה כלום - הוא **לא מאתחל את הזיכרון** (בניגוד לג'אבה). לכן, הערכים לא מוגדרים - ייתכן שיהיו שם ערכים מוזרים שבמקרה היו בזיכרון באותו זמן.

בנאים נוספים נראה בהמשך.

מפרקים - destructors

אנחנו מגיעים עכשיו לאחד ההבדלים העיקריים בין C++ לג'אבה. כזכור, ב-C++ ניהול הזיכרון הוא באחריות המתכנת. בפרט, אם אנחנו יוצרים משתנים חדשים על הערימה, אנחנו חייבים לוודא שהם משוחררים כשאנחנו כבר לא צריכים אותם יותר. לשם כך, בכל מחלקה שמקצה זיכרון (או מבצעת פעולות אחרות הדורשות משאבי מערכת), חייבים לשים **מפרק - destructor.**

מפרק הוא שיטה בלי ערך מוחזר, ששמה מתחיל בגל (טילדה) ואחריו שם המחלקה; ראו דוגמה בתיקיה 5 (למה גל? כי זה האופרטור המציין "not" של סיביות. למשל: 0~=-1).

חידה: האם אפשר להגדיר מפרק עם פרמטריס? האם אפשר לבצע העמסה (*overload*) למפרק? מדוע?

המתכנת אף-פעם לא צריך לקרוא למפרק באופן ידני; זוהי האחריות של הקומפיילר להכניס קריאה למפרק ברגע שהעצם מפסיק להתקיים. מתי זה קורה?

- אם העצם נוצר על המחסנית - העצם מפסיק להתקיים כשהוא יוצא מה-scope (יוצאים מהבלוק המוקף בסוגריים מסולסלים המכיל את העצם).
- אם העצם נוצר על הערימה בעזרת new - העצם מפסיק להתקיים כשמוחקים אותו בעזרת delete.

מה קורה כששוכחים לשים מפרק? המשאבים לא ישתחררו, וכתוצאה מכך תהיה דליפת זיכרון; ראו הדגמה בתיקיה 5.

האופרטורים new, delete

האופרטור new מבצע, כברירת מחדל, את הדברים הבאים:

- הקצאת זיכרון עבור עצם חדש מהמחלקה;
- קריאה לבנאי המתאים של המחלקה (בהתאם לפרמטרים שהועברו).

האופרטור delete מבצע, כברירת מחדל, את הדברים הבאים:

- קריאה למפרק של המחלקה (יש רק אחד - אין פרמטרים);
- שיחרור הזיכרון שהוקצה עבור העצם.

האופרטור []new מבצע, כברירת מחדל, את הדברים הבאים:

- הקצאת זיכרון עבור מערך של עצמים מהמחלקה;
- קריאה לבנאי ברירת-המחדל של כל אחד מהעצמים במערך.

האופרטור []delete מבצע, כברירת מחדל, את הדברים הבאים:

- קריאה למפרק של כל אחד מהעצמים במערך.
- שיחרור הזיכרון שהוקצה עבור המערך.

כשמאתחלים מערך ע"י new[], חייבים לשחרר אותו ע"י delete[]. כאן הקומפיילר לא יציל אתכם משגיאה - אם תשתמשו ב-delete במקום delete[], הקוד יתקמפל, אבל הקומפיילר יקרא רק למפרק אחד (של האיבר הראשון במערך). כתוצאה מכך תהיה לכם דליפת זיכרון, או אפילו שגיאת ריצה (ראו תיקיה 6).

הרכבה ואיתחול של מחלקות

הרכבה - composition

לעצם ב-++C יכולים להיות שדות שהם עצמם עצמים. דוגמה קלאסית: מחלקה Line שיש לה שני שדות מסוג Point גם בג'אבה זה אפשרי, אבל יש הבדל - ב-++C העצמים מסוג "נקודה" פשוט מונחים אחד ליד השני בעצם מסוג "קו", בעוד שבג'אבה העצם מסוג "קו" מכיל רק מצביעים לעצמים מסוג "נקודה". לכן ב-++C כשיוצרים עצם מסוג "קו", עוד לפני שמאתחלים אותו, כבר יש בו "נקודות". בעוד שבג'אבה הנקודות עדיין לא קיימות - יש רק מצביע המאותחל ל-null. ראו הדגמה בתיקיה 7.

למה זה משנה?

- בלי מצביעים, התוכנה תופסת פחות זיכרון ודורשת פחות זמן כשניגשים למשתנים.
- כשכל המבנה נמצא באותו מקום בזיכרון, ניהול הזיכרון מהיר יותר. בפרט, כשמשתמשים בזכרון מטמון (cache), זה מאד יעיל שכל המבנה נטען בבת-אחת לאותו בלוק בזיכרון.

מה קורה כשלעצמים המוכלים יש בנאים ומפרקים משל עצמם? הקומפיילר קורא להם לפי סדר פשוט והגיוני - כמו למשל בבנייה ופירוק של מכונית:

- בבנייה הולכים מהקטן אל הגדול - קודם-כל יוצרים את כל הרכיבים (במקרה שלנו: שתי נקודות), ואז יוצרים את העצם הגדול (במקרה שלנו: קו).
- בפירוק הולכים מהגדול אל הקטן - קודם-כל מפרקים את העצם הגדול (קו) ואז מפרקים את כל הרכיבים (שתי נקודות).

רשימת איתחול - initialization list

כשהקומפיילר בונה את הרכיבים, הוא משתמש ב**בנאי-ברירת-המחדל** שלהם (default constructor) - בנאי בלי פרמטרים - אם יש להם.

בנאי-ברירת-מחדל של מחלקה פשוטה (בלי רכיבים) - לא עושה כלום.

בנאי-ברירת-מחדל של מחלקה מורכבת - קורא לבנאי-ברירת-המחדל של הרכיבים.

אם לרכיבים אין בנאי-ברירת-מחדל - אז כברירת-מחדל, תהיה שגיאת קומפילציה.

שאלה: באיזה מקרה למחלקה אין בנאי-ברירת-מחדל? (התשובה בשיעור הקודם).

מה עושים אם רוצים לקרוא לבנאי אחר במקום בנאי ברירת-המחדל?

--- משתמשים ב**רשימת איתחול**. יש לשים את הרשימה **אחרי** הכותרת של הבנאי אבל **לפני** הסוגריים המסולסלים של קוד הבנאי. ראו דוגמה בתיקיה 7.

שימוש ברשימת-איתחול הוא **מהיר יותר** ו**בטוח יותר** מאיתחול השדות בתוך הבנאי. מדוע?

- מהיר יותר - כי הוא חוסך את האתחול ע"י בנאי ברירת-המחדל.
- בטוח יותר - כי הוא מבטיח שבתוך הבנאי, כל הרכיבים כבר בנויים עם הפרמטרים הנכונים.

רשימת איתחול לא מבטיחה שום דבר לגבי סדר האיתחול של הרכיבים! סדר האיתחול של הרכיבים הוא תמיד לפי הסדר שבו הם מוגדרים במחלקה.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.

סיכם: אראל סגל-הלוי.