מחלקות

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:
          ...
}
```

מימוש שיטות – בפנים או בחוץ

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

- דרך אחת היא לממש אותן ישירות בתוך המחלקה (בדומה לJava). זה נקרא מימוש "inline" (לא צריך לכתוב את המילה inline).
- דרך שניה היא לשים בתוך המחלקה רק **הצהרה** של השיטה, בלי גוף. את המימוש עצמו שמים מחוץ למחלקה. זה נקרא מימוש "out-of-line".

עקרונית, גם את המימוש החיצוני (out-of-line) אפשר לשים באותו קובץ (ראו תיקיה 0). אפשר גם לשים את כל המימוש בתוך המחלקה (inline), כמו בג'אבה. אבל, מקובל ורצוי מאד לשים את המימוש בקובץ נפרד. מקובל להגדיר כל מחלקה בשני קבצים (ראו תיקיה 1):

- 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.

למה זה עדיף? שתי 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.

בניית תוכנות עם כמה קבצי מקור

בתיקיה 1 אפשר לראות דוגמה פשוטה הכוללת מחלקה אחת (Complex). בתיקיה יש שלושה קבצי-מקור: Complex.hpp – הצהרת המחלקה, Complex.cpp – מימוש המחלקה, ו-main.cpp – התוכנית הראשית המשתמשת במחלקה.

כדי לבנות את המערכת, יש לתרגם כל קובץ cpp בנפרד לקובץ בינארי, ואז לקשר את הקבצים הבינאריים (כמו שלמדתם בקורס שפת C):

```
clang++-5.0 -std=c++17 Complex.cpp -o Complex.o
clang++-5.0 -std=c++17 main.cpp -o main.o
clang++-5.0 -std=c++17 Complex.o main.o
```

```
./a.out
```

שימו לב: התוכנית main.cpp משתמשת בקובץ הכותרת Complex.hpp, אבל היא לא צריכה להכיר את הקובץ Complex.cpp. תחשבו שהמחלקה Complex נכתבה ע"י מתכנת א, והתוכנית הראשית שמשתמשת בה main נכתבה ע"י מתכנת ב. מתכנת ב לא צריך לדעת איך מתכנת א מימש את כל השיטות של Complex – הוא צריך רק להכיר את הגדרת המחלקה, הנמצאת בקובץ Complex.hpp. מתכנת א, שייצר את המחלקה, צריך לתת למתכנת ב רק את הקובץ Complex.hpp, וכן את הקובץ Complex.o (הבינארי) – הוא לא צריך לחשוף את פרטי המימוש הנמצאים בקובץ Complex.cpp. זה יוצר מִידוּר – כל מתכנת יודע רק מה שהוא צריך לדעת ולא יותר.

מה קורה אם מעדכנים את הקובץ Complex.cpp? אז צריך לבנות מחדש את Complex.o:

```
clang++-5.0 -std=c++17 Complex.cpp -o Complex.o
```

ואז לקשר מחדש:

```
clang++-5.0 -std=c++17 Complex.o main.o
```

אבל, לא צריך לבנות מחדש את main.cpp – כי הוא לא משתמש ישירות ב Complex.cpp.
באותו אופן, אם משנים את main.cpp, צריך לבנות מחדש רק את main.o ולקשר – לא צריך לבנות מחדש את Complex.cpp.
אבל, אם משנים את Complex.hpp, צריך לבנות מחדש גם אם main.o וגם את Complex.o – כי שניהם משתמשים ב-Complex.hpp.
ההבחנה הזאת היא חשובה, כי זמן הקומפילציה של מערכות בשפת C++ יכול להיות מאד ארוך, במיוחד כשיש הרבה קבצים ובכל קובץ הרבה קוד. ראו: https://xkcd.com/303/
לכן כדאי שקובץ הכותרת יהיה קטן ויכלול כמה שפחות מימושים, כך שלא נצטרך לעדכן אותו בתדירות רבה מדי.

בניית תוכנות בעזרת "מייק" (make)

ראינו למעלה, שכאשר יש לנו תוכנה עם כמה קבצי-מקור, וחלק מהקבצים משתנים, אנחנו צריכים לקמפל מחדש את קבצי-המטרה שהמקורות שלהם התעדכנו. כשמעדכנים כמה קבצים בו זמנית, זה קשה לזכור מה בדיוק התעדכן ומה בדיוק צריך לבנות מחדש. התוכנה make היא תוכנה המבצעת משימה זו עבורנו.
דוגמה לבניית תוכנה בעזרת "מייק" נמצאת בתיקיה 2. יש שם מחלקה בשם Point (המורכבת משני קבצים – cpp, hpp כפי שלמדנו), וכן מחלקות נוספות המשתמשות בה (Rectangle, Triangle וכו'), וגם תוכנית ראשית (main).

כדי לעבוד עם מייק, צריך ליצור בתיקיה שלנו קובץ טקסט אחד בשם Makefile (בדיוק בשם זה).
בקובץ זה שמים כללים ליצירת "מטרות" מ"מקורות". לדוגמה, אחת ה"מטרות" שצריך ליצור היא Point.o. הכלל המתאים הוא:

```
Point.o: Point.cpp Point.hpp
    clang++-5.0 -std=c++17 --compile Point.cpp -o Point.o
```

הכלל חייב להתחיל בתחילת שורה (בלי רווחים מקדימים).
המילה שלפני הנקודתיים (במקרה זה Point.o) היא שם קובץ ה"מטרה".
המילים שאחרי הנקודתיים (במקרה זה Point.cpp Point.hpp) הם שמות קבצי ה"מקור" – הקבצים הדרושים על-מנת ליצור את המטרה.

השורות הבאות חייבות להתחיל בטאב (tab), וכל שורה כזאת מייצגת פקודה שצריך לבצע על-מנת ליצור את המטרה מהמקורות.
משמעות הכלל היא: "אם הקובץ Point.o לא קיים, או שהוא ישן יותר מאחד המקורות שלו, אז בצע את הפקודה בשורה למטה".
כדי להפעיל את הכלל, נמחק את הקובץ Point.o ונריץ את הפקודה הבאה (במסוף לינוקס):

```
make Point.o
```

הפקודה מריצה את הקומפיילר ובונה את הקובץ Point.o.
אם נריץ שוב make Point.o, נקבל הודעה האומרת ש-Point.o כבר מעודכן – לא צריך ליצור אותו שוב. "מייק" בודק את זמן העידכון האחרון של קובץ המטרה, ואם הוא מאוחר יותר מזמן העדכון האחרון של קבצי המקור – הוא חוסך את פקודת הקימפול – וכך חוסך לנו זמן.
אם נשנה את הקובץ Point.cpp (גם שינוי מינימלי כגון תוספת/מחיקת רווח) ונריץ שוב make Point.o, הפקודה שוב תריץ את הקומפיילר וכך תעדכן את המטרה Point.o.

ב-Makefile אפשר גם לשים מטרות התלויות במטרות אחרות. לדוגמה, הכלל:

```
a.out: Point.o main.o Rectangle.o Triangle.o Circle.o
    clang++-5.0 -std=c++17 Point.o main.o Rectangle.o Triangle.o Circle.o
```

אומר שהמטרה a.out (- קובץ ההרצה) תלויה בחמש מקורות – חמישה קבצים בינאריים. כל אחד מהמקורות הללו, הוא בעצמו מטרה באחד הכללים האחרים באותו Makefile. כאשר מריצים:

```
make a.out
```

ה"מייק" עובד באופן רקורסיבי: קודם-כל, הוא בונה כל אחד ואחד מהמקורות לפי הכלל המתאים (אם יש), ואחר-כך בונה את a.out.
כך למשל, אם נמחק את Point.o, או נעדכן את Point.cpp, ונריץ שוב make a.out, יתבצעו רק שתי פקודות – קומפילציה (כדי ליצור מחדש את Point.o) וקישור (כדי לעדכן את a.out):

```
clang++-5.0 -std=c++17 --compile Point.cpp -o Point.o
clang++-5.0 -std=c++17 Point.o main.o Rectangle.o Triangle.o Circle.o
```

אבל מה יקרה אם נעדכן את Point.hpp? - כיוון שכל המטרות בקובץ תלויות בו, "מייק" יבנה מחדש את כל המטרות – יתבצעו 5 פקודות קימפול ועוד פקודת קישור אחת. כפי שהסברנו למעלה, זו אחת הסיבות שכדאי לשים מימושים של שיטות בקובץ cpp ולא בקובץ הכותרת.

בתחילת קובץ ה-Makefile, יש כלל לבניית מטרה בשם all:

```
all: a.out
    ./a.out
```

משמעות הכלל הזה היא בדיוק כמו משמעות הכללים הקודמים שלמדנו: "בנה את a.out; בדוק אם קיים קובץ בשם all ואם הוא מעודכן; אם הוא לא מעודכן – הרץ את הפקודה בשורה הבאה". אבל יש הבדל אחד קטן – הקובץ all אף פעם לא קיים כי אנחנו לא יוצרים אותו. לכן, אם נכתוב:

```
make all
```

אז "מייק" תמיד יבנה את a.out ואז יריץ את a.out.

ב-Makefile יש עוד כלל המתייחס למטרה שאף פעם לא נוצרת:

```
clean:
    rm -f *.o a.out
```

משמעות הכלל הזה היא כמו כל הכללים הקודמים: "בדוק אם קיים קובץ בשם clean ואם הוא מעודכן; אם לא – הרץ את הפקודה בשורה הבאה". במקרה זה אין מקורות. וכיוון שהקובץ clean אף פעם לא נוצר, הפקודה make clean פשוט תריץ את הפקודה שמתחתיה ותמחק את כל הקבצים הזמניים בתיקיה.

פרט קטן נוסף לגבי make – כשמריצים make בלי פרמטרים, הוא תמיד מנסה לבנות את המטרה הראשונה בקובץ (במקרה שלנו, זו המטרה all). לכן, כשכותבים רק make, "מייק" יעדכן ויריץ את התוכנה שלנו עם כל העדכונים האחרונים – בדיוק מה שאנחנו רוצים.

להרחבה: בתיקיה הראשית של המאגר ariel-cpp-5779 יש קובץ בשם Makefile המשמש לבניה אוטומטית של קבצי pdf מתוך קבצי odt,odp במאגר זה. אתם מוזמנים להיכנס לקובץ ולנסות להבין איך הוא עובד.

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).

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.

סיכם: אראל סגל-הלוי.