ירושה

ירושה בשפת ++C עובדת, בגדול, באופן דומה ל-Java. שני הבדלים חשובים הם:

- אין ממשקים.
- יש ירושה מרובה - מחלקה אחת יכולה לרשת כמה מחלקות.

הערה על ג'אבה: בג'אבה, הממשקים ממלאים את התפקיד של ירושה - מחלקה אחת יכולה אמנם לרשת רק מחלקה אחת, אבל לממש אפס או יותר ממשקים. מג'אבה 8 ומעלה, ממשקים יכולים לכלול גם מימושים של שיטות, כך שזה מאפשר ירושה מרובה כמעט כמו ב-++C. ההבדל היחיד שנשאר הוא שבג'אבה אי-אפשר לעשות ירושה מרובה של שדות. אבל האפשרות הזאת ממילא לא מאד שימושית.

למה בכלל משתמשים בירושה? [תזכורת]

- כדי שהתוכנית שלנו תשקף את המציאות: ירושה מבטאת קשר של "is-a" -- "הוא סוג של". למשל, אם המחלקה של "מנהל" יורשת את המחלקה של "עובד", זה מעביר את המסר של "מנהל הוא סוג של עובד".
- כדי להשיג פולימורפיזם בזמן ריצה - ע"י החלפת שיטות (overriding). בניגוד לפולימורפיזם בזמן קומפילציה שמשיגים ע"י העמסת שיטות (overloading).

איך זה עובד?

נניח שהגדרנו מחלקה המייצגת אדם:

```
class Person {
    string _name;
    int _id;
public:
    Person(string name);
    string getName();
    string getId();   // ...
}
```

ואנחנו רוצים להגדיר גם מחלקה המייצגת מתכנת. אפשר להגדיר אותה כך:

```
class Programmer: public Person {
   string _company;
   Programmer(string name, string company): Person(name),
_company(company) {}
```

```
}
```

עכשיו לעצם מסוג Programmer יש גישה לשיטות של Person - getName, getId.

מאחרי הקלעים, כל עצם מסוג Programmer למעשה כולל שדה נסתר מסוג Person. כשקוראים ל-getName של Programmer, הקומפיילר למעשה קורא ל-getName של השדה הנסתר הזה.

שדות מוגנים

כפי שהגדרנו את המחלקה Person, המחלקה היורשת Programmer לא יכולה לגשת לשדות הפרטיים שלה - _id, name. במקרים רבים זה הגיוני, אבל אם רוצים לשנות את זה - המחלקה Person צריכה להגדיר את השדות האלה כ-protected - להוסיף "protected" ונקודתיים לפני הגדרת המשתנים (כמו בJava).

בניה ופירוק והשמה

הבנאים לא עוברים בירושה: אם למחלקה Person יש כמה בנאים, ולא הגדרנו שום בנאי למחלקה Programmer - אז למחלקה Programmer יש רק בנאי בלי פרמטרים - היא לא יורשת את הבנאים של Person.

כשבונים עצם ממחלקה מסויימת, חייבים קודם לבנות עצם מהמחלקה המורישה (כזכור, יש שדה נסתר מהמחלקה המורישה בתוך המחלקה היורשת). איך עושים את זה?

- אפשרות אחת היא לקרוא בפירוש לבנאי של המחלקה המורישה ברשימת האיתחול – בדיוק כמו שקוראים לבנאים של כל שאר השדות.
- אם לא קוראים בפירוש לבנאי של המחלקה המורישה - הקומפיילר יקרא אוטומטית לבנאי בלי פרמטרים, אם קיים (מה יקרה אם אין?)

המפרק גם-כן לא עובר בירושה. כשעצם מהמחלקה Programmer מתפרק, הקומפיילר מפרק אותו ואחר-כך קורא למפרק של Person באופן אוטומטי (בדיוק כמו שהוא קורא למפרקים של כל שאר השדות).

- סדר הבניה הוא: שדות מחלקת הבסיס -> בנאי מחלקת הבסיס -> שדות מחלקה יורשת -> בנאי מחלקה יורשת.
- סדר הפירוק הוא הפוך; דוגמה בתיקיה 1.

אופרטור השמה - טכנית עובר בירושה, אבל מעשית - אי אפשר להתשתמש באופרטור-השמה של המחלקה המורישה, כי הקומפיילר יוצר אוטומטית אופרטור-השמה חדש שמסתיר אותו.

ירושה ציבורית, מוגנת ופרטית

כשכותבים מחלקה יורשת, ניתן להגדיר בקרת-גישה למחלקה המורישה:

```
class Programmer : public Person
class Programmer : protected Person
class Programmer : private Person    // this is the default
```

כדי להבין מה זה אומר, נזכור שבאופן טכני, ירושה מתבצעת ע"י יצירת שדה מהסוג של המחלקה המורישה, בתוך המחלקה היורשת.

בדיוק כמו כל שאר השדות, גם השדה הזה יכול להיות ציבורי, מוגן או פרטי.

ברירת המחדל כשמחלקה יורשת מחלקה אחרת היא private - בדיוק כמו שדות רגילים. אבל במצב זה, אין אפשרות לעצם מחוץ למחלקה, להשתמש בשיטות של המחלקה המורישה. במצב רגיל אנחנו מעדיפים שהירושה תהיה public, ויש לציין זאת בפירוש.

החלפה - overriding

במחלקה יורשת, אם נגדיר שיטה שהחתימה שלה זהה לשיטה הקיימת כבר במחלקה המורישה, אז השיטה שהגדרנו תחליף את השיטה שירשנו. למשל, אם המחלקה Person מגדירה שיטה output, והמחלקה Programmer יורשת אותה ומגדירה שיטה output עם חתימה זהה - אז עצמים מסוג Programmer ישתמשו בשיטה המחליפה.

- שימו לב - החתימה של השיטה המחליפה חייבת להיות זהה לגמרי - כולל ה-const - אחרת זו לא תהיה החלפה אלא העמסה. כדי שהקומפיילר יוודא עבורנו שדייקנו בחתימה, ניתן להשתמש במילת-המפתח override (דומה לתג @Override בג'אבה).

אם רוצים להשתמש בשיטה של המחלקה המורישה מתוך המחלקה היורשת - כותבים את שם המחלקה המורישה, ארבע נקודות, ושם השיטה. למשל, מתוך השיטה output של Programmer כותבים:

```
Person::output(...)
```

כדי לגשת לשיטה שירשנו מ-Person . דוגמה בתיקיה 2.

ניתן גם לגשת לאופרטורים שירשנו. למשל, אם כותבים אופרטור השמה (=) ב-Programmer ורוצים לגשת לזה של Person, כותבים:

```
Person::operator=(...)
```

שיטות וירטואליות

ברירת-המחדל ב-++C היא, שהקומפיילר בוחר איזו שיטה להריץ, לפי **סוג המשתנה בזמן הקומפילציה**.

למשל, אם הגדרנו עצם מסוג Programmer, ואז שמנו עליו רפרנס או מצביע מסוג Person, אז הקומפיילר יבחר את **השיטה של Person** ולא את השיטה המחליפה (דוגמה בתיקיה 2).

זה מנוגד לג'אבה - בג'אבה השיטה נבחרת לפי **סוג העצם בזמן ריצה**.

אם רוצים לקבל ב++C את אותה התנהגות של ג'אבה, צריך להגדיר את השיטות הרלבנטיות כ**וירטואליות** - בעזרת במילת המפתח **virtual**. כשמסמנים שיטה כוירטואלית במחלקת-הבסיס, הקומפיילר יבחר את השיטה בכל המחלקות היורשות ממנה לפי סוג העצם בזמן ריצה (דוגמה בתיקיה 2).

הערות:

- כל שיטה המוגדרת כוירטואלית במחלקת-הבסיס, היא וירטואלית באופן אוטומטי גם בכל המחלקות היורשות ממנה. עם זאת, מקובל לסמן גם את השיטות במחלקות היורשות כ-`virtual` כדי שהקוד יהיה ברור יותר.
- אם יוצרים עצם מסוג הבסיס ו**מעתיקים** לתוכו עצם מהסוג היורש, העצם החדש הוא מסוג הבסיס, ולכן גם אם השיטה היא וירטואלית, הקומפיילר יקרא לשיטה של מחלקת הבסיס. כדי ליהנות מהיתרונות של שיטות וירטואליות, צריך להשתמש ברפרנס או בפוינטר (דוגמה בתיקיה 2).

קריאה לשיטות וירטואליות משיטות אחרות

אם שיטה f מוגדרת כוירטואלית במחלקת הבסיס, ושיטה g במחלקת הבסיס קוראת לה - אז הגירסה של שיטה f שתיקרא בפועל תלויה בסוג **העצם** בזמן ריצה. יש לזה שני יוצאי-דופן:

- אם השיטה g היא בנאי של מחלקת-הבסיס.
- אם השיטה g היא מפרק של מחלקת-הבסיס.

בשני המקרים האלה, שיטה f שתיקרא בפועל היא זו של מחלקת **הבסיס**. מדוע? בגלל סדר הבניה והפירוק:

- כשבונים את מחלקת הבסיס - המחלקה היורשת עדיין לא בנויה ולכן מסוכן לקרוא לשיטות שלה - אולי יש שדות חשובים שעדיין לא מאותחלים.
- כשמפרקים את מחלקת הבסיס - המחלקה היורשת כבר מפורקת, ולכן שוב מסוכן לקרוא לשיטות שלה - אולי יש שדות חשובים שכבר נמחקו.

מימוש שיטות וירטואליות - מה קורה מאחרי הקלעים?

כשכותבים שיטה וירטואלית - איך המחשב יודע לאיזו גירסה לקרוא?

- עבור כל מחלקה עם שיטות וירטואליות, מוגדרת **טבלת שיטות וירטואליות**. הטבלה הזאת היא למעשה מערך של מצביעים לפונקציות. עבור כל פונקציה וירטואלית, יש בטבלה הזאת מצביע למימוש שלה בפועל.

- בכל עצם מהמחלקה, יש מצביע לטבלת השיטות הוירטואליות.

- כשאנחנו כותבים קוד שקורא לשיטה וירטואלית, הקומפיילר למעשה כותב קוד שקורא את המצביע לטבלת השיטות הוירטואליות מהעצם שנמצא בזיכרון, הולך לכניסה המתאימה בטבלה (לפי שם הפונקציה שקראנו לה), ומפעיל את המימוש המתאים.

כדי להבין טוב יותר איך זה עובד, מומלץ מאד לקמפל קוד לאסמבלי בעזרת האתר `godbolt.org` ולראות את הקוד שנוצר. ראו גם תרשים במצגת.

שימו לב: בשפת C++ אפשר לבחור איזה שיטות יהיו וירטואליות ואיזה לא. הבחירה תלויה באופן שבו אנחנו רוצים להשתמש בשיטות. שיטה לא וירטואלית היא מהירה יותר וגם חסכונית יותר בזיכרון, אבל שיטה וירטואלית מאפשרת פולימורפיזם.

לעומת זאת, בשפות אחרות כגון Java, Smalltalk, Python, כל השיטות וירטואליות. לכן, בכל עצם יש מצביע לטבלה הוירטואלית, וכל קריאה לפונקציה עוברת דרך הטבלה הוירטואלית. זה מבזבז גם מקום וגם זמן. בשפת C++ הפילוסופיה היא "לא השתמשת - לא שילמת".

מפרקים וירטואליים

מפרק הוא שיטה - בדיוק כמו כל שיטה אחרת. כברירת-מחדל, המפרק הוא **סטטי**. לכן, אם אנחנו מגדירים מצביע מסוג Base ומאתחלים אותו עם עצם מסוג Derived ואז מוחקים - הקומפיילר יקרא למפרק של **Base**.

בדרך-כלל זה לא מה שאנחנו רוצים. ייתכן שב-Derived יש מצביע לגוש זיכרון בערימה שצריך למחוק. לכן חשוב להגדיר את המפרק כוירטואלי במחלקת-הבסיס.

כשמגדירים מפרק וירטואלי במחלקת-בסיס, כל המחלקות היורשות ממנה חייבות להגדיר מפרק.

שיטות וירטואליות טהורות

לפעמים רוצים להגדיר מחלקה מופשטת, שאין עצמים השייכים אליה ישירות. למשל Shape. אנחנו רוצים שהמתכנתים שישתמשו בחבילה שלנו, לא יגדירו עצם מסוג Shape, אלא יגדירו מחלקות שיורשות את Shape (כגון Circle, Square...) וימשו את הפונקציה draw בהתאם.

דרך אפשרית לעשות זאת היא להגדיר את השיטה draw כשיטה וירטואלית טהורה - בלי מימוש:

```
virtual void draw() const = 0;
```

התוספת "=0" אומרת לקומפיילר שהשיטה הזאת היא מופשטת (כמו abstract בג'אבה), אין לה מימוש, חייבים לממש אותה במחלקות יורשות.

אפשר להגדיר מחלקה שיש בה **רק** שיטות וירטואליות טהורות. מחלקה כזאת היא המקבילה ב-C++ לממשק (interface) בג'אבה.

החלפת שיטות פרטיות

כפי שלמדנו למעלה, אם שיטה המוגדרת כפרטית (private) במחלקת הבסיס, המחלקות היורשות לא יכולות להשתמש בה.

אבל, המחלקות היורשות עדיין יכולות להחליף אותה - וזה שימושי במיוחד אם השיטה הזאת היא וירטואלית.

לכן, יש מחלקות-בסיס המוגדרות באופן הבא:

- שיטות ציבוריות לא וירטואליות - מגדירות את הממשק הציבורי של המחלקה.
- שיטות פרטיות וירטואליות-טהורות - מגדירות את פרטי-המימוש של המחלקה.

המחלקות היורשות חייבות לממש את השיטות הוירטואליות-הטהורות הפרטיות, אך אינן יכולות לקרוא להן.

דגם pimpl

אפשר להשתמש בירושה, בפרט של שיטות וירטואליות טהורות, כדי להסתיר את פרטי המימוש של מחלקה.

עקרונית, כל שדה המוגדר כ"פרטי" שייך למימוש של המחלקה. הלקוחות של המחלקה שלנו לא אמורים לדעת על השדות האלה. הבעיה היא, שב C++ , השדות הפרטיים מוגדרים בתוך הגדרת המחלקה שנמצאת בתוך קובץ הכותרת שלה. הלקוחות מקבלים את קובץ הכותרת וכך הם יודעים מה השדות הפרטיים של המחלקה.

דרך מקובלת לעקוף בעיה זו היא להשתמש בדגם-עיצוב הנקרא pimpl - קיצור של pointer to implementation. כל השיטות הציבוריות של המחלקה (בלי השדות הפרטיים) מוגדרות בתוך מחלקה וירטואלית-טהורה, בקובץ h. המימוש והשדות הפרטיים מוגדרים במחלקה יורשת הנמצאת כולה בקובץ cpp ואינה גלויה למשתמש.

לדוגמה, נניח שרוצים להגדיר רשימה מקושרת שאפשר להוסיף לה איברים, אבל לא רוצים שהלקוחות יידעו באיזה שדות אנחנו משתמשים. אז בקובץ list.h נגדיר מחלקה בשם List שתכלול רק שיטות וירטואליות טהורות (כגון Add) ללא כל שדות. בנוסף, תהיה לה שיטה סטטית בשם make.

בקובץ list.cpp נגדיר מחלקה בשם ListImpl , שהיא תירש את List, תוסיף שדות ותממש את השיטות הוירטואליות הטהורות. שם יהיה גם המימוש של השיטה הסטטית make, שייצור עצם חדש מסוג ListImpl ויחזיר פוינטר עבורו. ראו במצגת.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- ירושה של אופרטור השמה: https://stackoverflow.com/q/12009865/827927
- פונקציה וירטואלית פרטית: https://stackoverflow.com/a/3978552/827927
- קריאה לוירטואליות מבנאי ומפרק: https://stackoverflow.com/q/962132/827927

סיכם: אראל סגל-הלוי.