

משתני רפרנס - reference variables

רפרנסים לעומת פוינטרים

בשפת סי, כשרצינו לקבל כתובת של משתנה (למשל כדי לשלוח לפונקציה שתוכל גם לשנות אותו), הגדרנו **פוינטר** למשתנה, למשל:

```
int num; int* pnum = &num;
```

כדי להשתמש בפוינטר, צריך להקדים לו כוכבית, למשל:

```
(*pnum) = 5;
```

ואם המשתנה הוא עצם או struct, צריך להשתמש בחץ, למשל:

```
Point location; Point* plocation = &location;
```

```
plocation->x = 5; // equivalent to (*plocation).x = 5;
```

בשפת C++ יש דרך נוספת לקבל כתובת של משתנה - להגדיר **רפרנס** (reference):

```
int& rnum = num;
```

זה מאד דומה, רק שהפעם, אפשר לגשת למשתנה בלי כוכבית, למשל:

```
rnum = 5;
```

ובלי חץ, למשל:

```
Point& rlocation = location; rlocation.x = 5;
```

חוץ מצורת הגישה, ישנם שני הבדלים עיקריים בין פוינטר לרפרנס:

- אפשר ליצור פוינטר בלי לאתחל אותו (או לאתחל ל-`nullptr`), אבל כשיוצרים רפרנס - חייבים לאתחל אותו מייד למשתנה. זה אמור לצמצם את הסיכוי לשגיאות - כשיש לנו רפרנס, אנחנו יכולים להיות בטוחים שיש שם עצם ממשי ולא `null`.
- אפשר לשנות פוינטר לאחר יצירתו, למשל להוסיף לו 1 (ואז הוא יצביע למקום אחר בזיכרון). אבל אי אפשר לשנות רפרנס לאחר יצירתו. גם זה אמור לצמצם את הסיכוי לשגיאות - רפרנס תמיד מצביע לאותו מקום בזיכרון ולא יכול "לנדוד" למקומות לא רצויים.

ראו הדגמה בתיקיה 3.

השוואה לג'אבה

בג'אבה, כל המשתנים שהם עצמים (לא פרימיטיביים), הם כמעט כמו רפרנסים. הם מכילים כתובת של משתנה, אפשר לגשת אליהם כמו שניגשים למשתנה עצמו - בלי כוכבית ובלי חץ, ואי-אפשר "להזיז" אותם.

אבל בניגוד לרפרנסים, אפשר לאתחל אותם עם `null`.

שימושים

השימוש העיקרי של רפרנס הוא לצורך החזרת ערכים מתוך פונקציות. לדוגמה, פונקציה כגון `swap` - שמחליפה בין הערכים שהיא מקבלת - צריך לכתוב עם רפרנסים. בסי היינו כותבים את הפונקציה עם פוינטרים, אבל זה פחות נוח כי הפונקציה הקוראת צריכה להפוך את הפרמטרים לפוינטרים, והפונקציה `swap` עצמה צריכה לגשת אליהם בצורה שניגשים לפוינטרים. רפרנסים מאפשרים להשיג אותה מטרה בצורה יותר פשוטה וקריאה.

"ערך שמאלי" ו"ערך ימני" - `lvalue`, `rvalue`

אחד המושגים שמופיעים הרבה בתיעוד של C++ ובהודעות-השגיאה של קומפיילרים הם: `lvalue`, `rvalue`.

- `lvalue` - מציין דבר שיש לו כתובת בזיכרון, דבר שמתקיים גם אחרי שהביטוי מסתיים. למשל: משתנה, רפרנס, כוכבית-פוינטר. הוא נקרא `lvalue` כי יש לו `location` וכי אפשר לשים אותו בצד השמאלי (`left`) של סימן `=`.

- `rvalue` - מציין דבר שלא ממשיך להתקיים אחרי שהביטוי מסתיים. למשל: מספר, מחרוזת קבועה, ביטוי מתמטי `(x+y)`, קריאה לפונקציה `(f(123))`. הוא נקרא `rvalue` כי הוא יכול להופיע רק בצד הימני (`right`) של סימן `=`.

לדוגמה, אם כותבים ביטויים כמו:

```
7 = a;
```

```
x+y = a;
```

```
f() = a;
```

(כאשר `f` היא פונקציה שמחזירה `int`) מקבלים הודעת שגיאה: `expression must be a "modifiable lvalue"`.

טוב, זה די צפוי, אבל הודעת-שגיאה עם `lvalue` יכול להופיע גם במקרים פחות צפויים. למשל:

```
int& refToInt = 5;
```

```
int& refToInt = x+y;
```

```
int& refToInt = f();
```

גם כאן נקבל הודעת שגיאה האומרת לנו שאי-אפשר לאתחל רפרנס עם דבר שהוא לא `lvalue`. למה? כי רפרנס אמור להצביע לעצם שאפשר לשנות אותו - הוא לא יכול להצביע למספר או לביטוי זמני.

יש מקרה אחד שבו אפשר להציב `rvalue` בתוך רפרנס - כאשר מגדירים את הרפרנס כקבוע - `const`. למשל הביטויים הבאים חוקיים:

```
const int& refToInt = 5;
```

```
const int& refToInt = x+y;
```

```
const int& refToInt = f();
```

כיוון שהדבר שמצביעים אליו מוגדר כ"קבוע", הקומפיילר ממילא לא ייתן לנו לשנות אותו, ולכן הרפרנס יכול להצביע גם לדבר שאין לו כתובת בזיכרון.

למה צריך את זה? - השימוש העיקרי של const reference הוא כשרוצים להעביר ארגומנטים לפונקציות. זה שימושי במיוחד כשהארגומנט הוא אובייקט גדול. יש כמה דרכים להעביר אותו לפונקציה:

- אם מעבירים אותו כערך - הוא **מועתק** למשתנה זמני שמועבר לפונקציה. זה יכול להיות מאד בזבזני כשהמשתנה גדול.
- אם מעבירים אותו כרפרנס - הוא לא מועתק, רק הכתובת שלו מועתקת, **אבל**, הפונקציה יכולה לשנות את ערכו, וזה עלול לגרום שגיאות לוגיות.
- אם מעבירים אותו כרפרנס-קבוע (& const), רק הכתובת שלו מועתקת, ובנוסף, הקומפיילר מוודא שאי-אפשר לשנות את ערכו (יתרון נוסף הוא, שאפשר להעביר לפונקציה rvalue).

ראו הדגמה בתיקיה 3.

קבועים - const

המילה const קיימת גם בשפת סי. אם שמים אותה מייד לפני שם של משתנה - המשתנה יהיה קבוע, והקומפיילר לא ייתן לנו לשנות אותו. עד כאן זה פשוט.

כשיש פוינטרים זה קצת יותר מסובך - צריך לשים לב מי הקבוע - הפוינטר או הדבר שהוא מצביע עליו?

```
int *const p1 = &i; // a const pointer to an un-const variable
p1++;             // compiler error
(*p1)++;          // ok
const int* p2 = &b; // an un-const pointer to a const variable
p2++;             // ok
(*p2)++;          // compiler error
const int * const p3 = &b; // a const pointer to a const variable
```

כשיש מחלקות ועצמים, המצב מסתבך עוד יותר. אפשר לשים את המילה const בכותרת של שיטה, אחרי הסוגריים. המשמעות היא, שבתוך השיטה הזאת, המשתנה this יהיה מצביע לעצם קבוע. במילים אחרות: השיטה לא תוכל לשנות את השדות של העצם.

למה זה חשוב?

- זה עוזר לאתר באגים ותקלות. אם שיטה מסויימת מוגדרת כ-const, אפשר להיות בטוחים שהיא לא משנה את העצם, ולכן אם העצם משתנה כנראה התקלה במקום אחר.
- אם בתוכנית הראשית מגדירים עצם כ-const, אפשר לקרוא על העצם הזה רק לשיטות שהוגדרו כ-const.

למה זה קשה? כי כשיטה היא const, הקומפיילר לא ייתן לנו לקרוא מתוכה לשיטות אחרות שהן לא const! לכן, כשמגדירים שיטה כ-const עלולה להיווצר "תגובת שרשרת" שתדרוש מאיתנו הרבה שינויים בקוד. לכן עדיף מלכתחילה להגדיר כ-const כל שיטה שאנחנו יודעים שלא תצטרך לשנות את העצם.

קבועים והעמסה

כשמוסיפים את המילה `const` לשיטה, היא הופכת לחלק מה"חתימה" של השיטה. מכאן שאפשר ליצור שתי שיטות שונות עם אותו שם ואותם פרמטרים - אחת עם `const` ואחת בלי. הקומפיילר ישתמש בשיטה הנכונה לפי ההקשר - אם משתמשים בשיטה כ-`lvalue` הוא יקרא לשיטה בלי ה-`const`, ואם משתמשים בשיטה כ-`rvalue` הוא יקרא לשיטה עם ה-`const`.

מתי זה שימושי? למשל, כשמגדירים מבנה-נתונים של וקטור. למבנה יש שיטה `get(i)` שמחזירה את האלמנט במקום `i`. מקובל ליצור שתי שיטות עם **מימוש זהה**:

- אחת מיועדת לקריאה - היא מוגדרת כ-`const` ומחזירה `&const`.
- השניה מיועדת לכתיבה - היא מוגדרת בלי `const` ומחזירה `&`.

הקומפיילר יחליט לאיזו שיטה לקרוא, לפי סוג המשתנה: אם המשתנה הוא `const` הוא יקרא לשיטה המיועדת לקריאה בלבד; אחרת הוא יקרא לשיטה המיועדת לכתיבה.

שדה mutable

לפעמים רוצים לשמור בתוך עצם, שדה מסויים שהערך שלו לא משקף את המצב של העצם. לדוגמה, זה יכול להיות `cache` של תוצאת-ביניים של חישוב כלשהו. החישוב לא משנה את מצב העצם, אבל הוא צריך לשמור את התוצאה ב-`cache`.

כדי שהקומפיילר יאפשר לנו לעשות זאת, נסמן את ה-`cache` ב-`mutable`. שדה המסומן ב-`mutable` ניתן לשינוי גם אם העצם הוא `const`.

בנאי מעתיק - copy constructor

דיברנו על סוגים מיוחדים של בנאים. אחד מהם הוא בנאי ללא פרמטרים. עוד בנאי מיוחד הוא **בנאי מעתיק**. בנאי מעתיק למחלקה `T` הוא בנאי המקבל פרמטר אחד בלבד, שהסוג שלו הוא:

`const T&`.

הקומפיילר קורא לבנאי הזה אוטומטית בכל פעם שצריך להעתיק עצם מהסוג `T` לעצם חדש, למשל, כשמעבירים פרמטרים לפונקציות.

אם לא מגדירים בנאי מעתיק, ברירת המחדל היא לבצע העתקת סיביות (`bitwise copy`). זה בסדר כשמדובר במחלקות פשוטות (`Point`) אבל לא טוב כשמדובר במבנים מורכבים עם מצביעים.

במקרה שברירת-המחדל לא מתאימה, אנחנו צריכים להגדיר בעצמנו בנאי מעתיק שיבצע "העתקה עמוקה".

חידה: מדוע הפרמטר חייב להיות מסוג `const T&` ולא פשוט מסוג `T`?

תשובה: כי כדי להעביר פרמטר מסוג `T`, הקומפיילר צריך לקרוא לבנאי המעתיק, אבל אנחנו מגדירים את הבנאי הזה עכשיו!

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- <https://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c>

סיכום: אראל סגל-הלוי.