תיכנות בתבניות - template

[כ-60 דקות]

למה בכלל צריך תבניות?

במקרים רבים אנחנו צריכים לכתוב פונקציה כללית המתאימה לטיפוסי משתנים שונים, אבל מתבצעת בצורה שונה לכל טיפוס.

דוגמה פשוטה היא פונקציה להחלפה בין שני משתנים: `swap(a,b)`.

אפשר לכתוב פונקציה שתחליף בין שני משתנים מסוג `int` - בטח כתבתם כזאת בעבר. היא די פשוטה ויש בה 3 שורות לכל היותר.

אפשר גם לכתוב פונקציה שתחליף בין שני משתנים מסוג `string`; היא תהיה זהה לחלוטין פרט לסוגי המשתנים.

האם אפשר לכתוב את ה-`swap` פעם אחת, ולהשתמש בה לכל סוגי-הנתונים?

בשפת סי יכולנו לעשות דבר כזה בעזרת מצביע כללי (`void*`), אבל זה בעייתי מכמה סיבות: (א) אין בדיקה שהטיפוסים אכן תואמים - אפשר למשל לנסות להחליף מספר שלם עם מחרוזת והקומפיילר לא ישים לב. (ב) הקריאה פחות נוחה - צריך להעביר לפונקציה את גודל הסוג שרוצים להחליף. (ג) הביצוע פחות יעיל - צריך להעביר בית בית.

בשפת C++ יש דרך נוחה ובטוחה יותר להגדיר פונקציה כללית: מילת הקסם **`template` - תבנית.** בעזרת המילה הזאת ניתן להגדיר את `swap` כך שיעבוד אוטומטית לכל הסוגים; ראו תיקיה 1.

```
template <typename T> void swap(T& a, T& b) {
  T tmp = a;   a = b;  b = tmp;
}
```

איך זה עובד?

כשמגדירים תבנית, הקומפיילר זוכר את ההגדרה אבל עדיין לא מייצר שום קוד. תבנית היא לא קוד - היא רק מרשם לייצור קוד.

הקוד נוצר רק כשמנסים להפעיל את התבנית. לדוגמה, הקריאה `swap(a,b)` כאשר `a,b` הם מסוג `int`, תגרום ליצירת פונקציה `swap(int&,int&)`. אותה קריאה בדיוק כאשר `a,b` הם מסוג `double`, תגרום ליצירת פונקציה **אחרת** שבה הפרמטרים הם מסוג `double&`.

בכל פעם שהקומפיילר נתקל בקריאה לפונקציה, ולא מוצא פונקציה עם הפרמטרים המתאימים - הוא מחפש תבנית שאפשר להשתמש בה כדי ליצור פונקציה מתאימה. התהליך הזה של יצירת פונקציה מסויימת מתוך תבנית נקרא `instantiation` (מלשון יצירת `instance`).

אפשר לראות את זה יפה ב-`compiler explorer`: כשיש קריאה - הקומפיילר יוצר פונקציה, כשאין קריאה - הוא לא יוצר כלום.

איפה מגדירים תבניות?

הקומפיילר משתמש בתבניות ליצירת קוד תוך כדי קומפילציה, ולכן כל תבנית חייבת להיות מוגדרת כולה (כולל המימוש) במקום שהקומפיילר יכול לראות כאשר הוא בונה את התוכנית הראשית - כלומר בקובץ h.

אם נגדיר בקובץ h כותרת של תבנית בלי מימוש, כמו שעשינו לפונקציות רגילות - הקומפיילר לא יוכל להשתמש בתבנית זו.

הנחות של תבניות

כל תבנית מניחה הנחות מסויימות על הסוגים שהיא מקבלת. ההנחות האלו לא כתובות בפירוש בשום מקום, אבל הן נובעות מהגדרת התבנית.

לדוגמה, הפונקציה swap מניחה ש:

(א) אפשר ליצור עצם חדש מסוג T - יש לו בנאי מעתיק ציבורי.

(ב) אפשר להעתיק עצם מסוג a לתוך עצם מסוג b - יש לו אופרטור השמה ציבורי.

יש כמה מקרים אפשריים:

1. אם לא הגדרנו שום בנאי מעתיק / אופרטור השמה - הקומפיילר ייצור אותם עבורנו כברירת מחדל (העתקה שטחית) והפונקציה swap תשתמש בהם.

2. אם הגדרנו בנאי-מעתיק ואופרטור-השמה ציבוריים משלנו (למשל כדי לבצע העתקה עמוקה), הפונקציה swap תשתמש בהם אוטומטית.

3. אם הגדרנת בנאי-מעתיק פרטי (כי לא רצינו שיוכלו להעתיק את המחלקה), אז הקומפיילר לא יצליח לייצר עבורה את הפונקצייה swap, ויחזיר שגיאת קומפילציה.

שימו לב - הבדיקה הזאת מתבצעת רק כשאנחנו מנסים להפעיל את התבנית swap על סוג כלשהו. כלומר, אנחנו לא נקבל שגיאת קימפול על התבנית swap, אלא רק על ההפעלה של התבנית הזאת על סוג שאין לו בנאי מעתיק ציבורי (ראו תיקיה 1).

דוגמה נוספת: נניח שאנחנו כותבים תבנית של פונקציית סידור (sort) ע"י השוואות והחלפות. ההנחות של התבנית הזאת הן:

(א-ב) אותן הנחות של swap - כי היא קוראת ל-swap;

(ג) אפשר להשוות שני עצמים מהסוג הנתון ע"י אופרטור "קטן מ-".

דוגמה נוספת: תבנית לאופרטור "לא שווה", שקוראת אוטומטית לאופרטור "==" והופכת את התשובה (ראו במצגת).

איך הקומפיילר בוחר לאיזו פונקציה לקרוא?

כשיש כמה פונקציות עם אותו שם ואותו מספר פרמטרים, הקומפיילר בוחר ביניהם באופן הבא:

- קודם-כל, הוא בוחר את כל הפונקציות עם רמת ההתאמה הגבוהה ביותר (הכי פחות המרות סוגים).
- בתוך הקבוצה עם רמת ההתאמה הגבוהה ביותר, הוא בוחר את הפונקציות שהן לא תבניות, ורק אם אין כאלה - הוא מפעיל את התבניות.

הדבר מאפשר לכתוב תבנית כללית, ויחד איתה, פונקציה ספציפית יותר הפועלת באופן שונה על סוגים שונים. הקומפיילר יבחר את הפונקציה הספציפית אם היא מתאימה; אחרת - הוא יבחר את הפונקציה הכללית יותר. יש דוגמאות רבות במצגת. דוגמאות נוספות:

- swap - עבור טיפוסים מספריים - אפשר לבצע בעזרת פעולות חיבור וחיסור ובלי משתנה זמני (זה שימושי רק אם מאד חשוב לנו לחסוך במקום על המחסנית).
- swap - עבור מחלקות עם העתקה עמוקה - אפשר לבצע במהירות רבה יותר ע"י העתקה שטחית.

תבניות של מחלקות

[כ-60 דקות]

עד כאן ראינו תבניות של פונקציות. באותו אופן אפשר להגדיר תבניות של מחלקות.

נניח למשל שאנחנו רוצים להגדיר מחסנית. אפשר להגדיר מחסנית של מספרים, מחרוזות וכו'... הלוגיקה תהיה בדיוק אותו דבר.

לכן עדיף להגדיר את המחסנית כתבנית - template:

```
template <typename T> class Stack { ... }
```

שימו לב - כיוון שזו תבנית, צריך לממש את כל השיטות של המחלקה בקובץ h - אין לה קובץ cpp.

כדי להשתמש בתבנית הזאת, צריך להגיד לקומפיילר איזה סוג בדיוק לשים במקום T, למשל:

```
Stack<int> intList; // T = int
Stack<string> stringList; // T = string
```

ראו דוגמה בתיקיה 2.

איטרטורים

נניח שבנינו מחסנית כללית. עכשיו אנחנו רוצים להכניס לתוכה בבת-אחת מערך שלם של עצמים, למשל לכתוב: stack.push(arr). איך נכתוב את השיטה push?

דרך אחת היא להעביר ל-push שני ארגומנטים: פוינטר למערך (arr), וכן את מספר האיברים במערך (כי הוא לא נמצא במערך -בניגוד לג'אבה).

דרך שניה, מקובלת מאד ב-C וגם ב-++C, היא להעביר שני ארגומנטים: פוינטר לתחילת המערך (arr) ופוינטר לסוף המערך (arr+length). זה מאפשר לבצע לולאה מהירה יותר הסורקת את כל המערך מהתחלה לסוף. בהמשך נראה שהשיטה הזו גמישה יותר.

אבל מה קורה אם המערך שממנו אנחנו מעתיקים הוא לא מערך פרימיטיבי של השפה, אלא מיכל כללי יותר, למשל, IntBuffer, LinkedList, ...? למיכל כזה כבר אין פוינטרים - ייתכן שהמידע בכלל לא נמצא בבלוק רציף בזיכרון!

האם אפשר לכתוב את הפונקציה push פעם אחת, כך שהיא תרוץ גם עם מערכים פרימיטיביים וגם עם מיכלים כלליים יותר?

התשובה היא כן. לשם כך צריך להגדיר **איטרטור**. איטרטור הוא עצם הצמוד למיכל כלשהו, ומתנהג כמו פוינטר, כלומר אפשר לכתוב בעזרתו קוד כמו:

```
for(; begin!=end; ++begin) { // Do something with *begin }
```

לשם כך צריך ליצור עצם שיש לו אופרטורים של פוינטר:

- השמה;
- הגדלה ב-1 (שני אופרטורים);
- שווה / לא שווה;
- אופרטור כוכבית (*) אונרי - גישה לעצם שהאיטרטור מצביע עליו.

במקרים מסויימים נרצה להוסיף אופרטורים נוספים:

- סוגריים מרובעים - גישה לאיברים שלא לפי הסדר;
- הקטנה ב-1 (חזרה אחורה);
- הגדלה / הקטנה במספר כלשהו.

איך יוצרים איטרטור? בדרך-כלל הוא יהיה מחלקה פנימית בתוך המחלקה שעליה הוא רץ. למשל בתוך המחלקה Stack תהיה מחלקה פנימית בשם iterator. אנחנו נוכל לגשת אליה באופן הבא:

```
Stack<int>::iterator i = myStack.begin();
```

עכשיו אנחנו יכולים להוסיף למחסנית שלנו שיטת push גנרית, או בנאי גנרי. הם יהיו template (כן, אפשר לכתוב שיטה שהיא תבנית בתוך מחלקה שהיא עצמה תבנית!). הפרמטר לתבנית יהיה סוג האיטרטור. כך, התבנית תוכל לקבל גם פוינטר רגיל של C, וגם איטרטור של מחסנית, או כל איטרטור אחר התומך באופרטורים של פוינטר.

ראו דוגמה בתיקיה 2.

יש כאן חיסכון גדול בכתיבת קוד. נניח שיש לנו m מבני-נתונים שונים. בלי תבניות, היינו צריכים m^2 בנאים - לבנות כל אחד מכל אחד אחר. עם תבניות, אנחנו צריכים רק m בנאים, ועוד איטרטור לכל מבני-נתונים, סה"כ כמות העבודה שלנו ירדה לבערך 2m.

לולאות עם איטרטורים

כשמחלקה מגדירה איטרטור ופונקציות begin, end, ניתן לרוץ על איברי המחלקה בלולאה באופן הבא:

```
for (Stk<int>::iterator i=mystack.begin(); i!=mystack.end(); ++i)
{
   int val = *i;
   cout << val << endl;
}
```

החל מ C++11, יש צורה קצרה יותר לכתוב את הלולאה הנ"ל:

```
for (int val: mystack) {
   cout << val << endl;
}
```

הפורמט הקצר נקרא foreach loop. הוא שקול לחלוטין לפורמט הארוך - הקומפיילר מתרגם את הפורמט הקצר לפורמט הארוך. לכן, כדי שהפורמט הקצר יעבוד, המחלקה צריכה להגדיר פונקציות begin ו-end המחזירות איטרטור עם אופרטורים מתאימים - ++, * וכו'.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- Peter Gottschling, "Discovering Modern C++", chapter 3.

סיכם: אראל סגל-הלוי.