תבניות למתקדמים

וריאציות על תבניות

1. תבניות יכולות לקבל שני פרמטרי-סוג או יותר, למשל:

```
template <typename Key, typename Value> class pair { Key k; Value
v; ... }
```

2. תבניות יכולות לקבל פרמטרים שהם לא סוגים אלא מספרים. למשל, אפשר ליצור מחלקה המייצגת מערך שהגודל שלו ידוע בזמן קומפילציה:

```
template<typename T, int Size> class array{ T m_values[Size]; ... };
```

המספר חייב להיות ידוע בזמן קומפילציה - אי אפשר להשתמש למשל במשתנה שמגיע מהקלט התקני.

למה זה טוב? - כדי לקבל מבנה זהה לחלוטין למערך של C (על המחסנית, עם גודל קבוע, בלי שדה השומר את הגודל), ויחד עם זה, שנוכל להוסיף לו אופרטורים ושיטות שונות (ראו דוגמה בתיקיה 1).

בנוסף, העובדה שהמספר ידוע בזמן קומפילציה מאפשרת לקומפיילר לבצע אופטימיזציות מעניינות, למשל, לפרוש לולאות פנימיות של המחלקה. למשל, אם יוצרים מערך עם פרמטר Size=3, ואחת השיטות שלו היא לולאה מ-0 עד Size, אז ייתכן שהקומפיילר בכלל לא ייצור לולאה אלא שלוש קריאות לגוף הלולאה - יחסוך את משתנה-הלולאה ואת הקפיצות קדימה ואחורה. ראו ב-godbolt.org.

3. אפשר להעביר ברירות-מחדל לפרמטרי-סוג וגם לפרמטרי-מספר, למשל:

```
template<typename T= char, size_t Size= 1024> class array { private: T
m_values[Size]; };
```

במקרה זה, הסוג array<char> והסוג array<char,1024> הם שני שמות של אותו סוג - ניתן לשים עצם מסוג אחד במשתנה מהסוג השני ולהיפך. אבל הסוגים array<char,1023> או array<char,1025> הם סוגים שונים - אי אפשר לשים עצם מסוג אחד במשתנה מהסוג השני, למרות שרק המספר שונה (בניגוד למצב שבו יש לנו וקטור עם שדה פנימי בשם size).

4. אפשר להעביר פרמטר-סוג T, ואז להעביר פרמטר אחר ולקבוע שהוא חייב להיות מסוג T. למשל:

```
template<typename T, T default> class Buffer {
     public: Buffer(int size) { /* initialize all elements to the
default */ } };
```

הגדרת מקרים פרטיים לתבניות (template specialization)

אחרי שהגדרנו תבנית כללית המתאימה לכל הסוגים, אפשר להגדיר מקרים פרטיים שלה עם מימוש שונה ה"תפור" במיוחד לסוג מסויים. הנה כמה דוגמאות.

דוגמה א (תיקיה 2)

הגדרנו פונקציה swap גנרית המבצעת החלפה בעזרת בנאי מעתיק ואופרטור השמה. הפונקציה הזאת עובדת בכל המקרים, אבל, לפעמים היא בזבזנית - למשל כשרוצים להחליף IntBuffer או vector, תתבצע העתקה העמוקה שלוש פעמים. במקרה זה אין צורך בהעתקה עמוקה - אפשר להסתפק בהעתקה שטחית של הפוינטרים והמספרים בלבד. אפשר להגדיר מקרה פרטי של swap שעושה את זה.

דוגמה ב (תיקיה 3)

הגדרנו תבנית-מחלקה בשם Wrapper עם פרמטר-סוג Type; המחלקה עוטפת שדה מסויים data מסוג Type, ויש בה שיטה isBigger הבודקת אם השדה הזה גדול יותר ממשתנה אחר כלשהו.
המימוש הכללי משתמש באופרטור > של Type, אבל המימוש הזה לא מתאים למשל למשתנים מסוג char* כי הוא יגרום להשוואה בין פוינטרים ולא בין המחרוזות שהם מייצגים.

אפשר להגדיר מימוש ספציפי באופן הבא:

```
template <> bool Wrapper<char*>::isBigger(const char*& data)
```

המימוש הזה יוגדר בכל פעם שנפעיל את התבנית Wrapper אם פרמטר-סוג char*.

דוגמה ג (תיקיה 3)

יש לנו וקטור כללי. אנחנו רוצים להגדיר מקרה פרטי - וקטור בוליאני - שבו כל 8 פרטים בוליאניים תופסים רק בית אחד בזיכרון. ראו בתיקיה 3.

דוגמה ד (תיקיה 4)

אנחנו רוצים לבנות שיטה כללית לחלוקת מספר ב-10. בדרך-כלל הסוג המוחזר זהה לסוג המועבר, אלא אם כן הסוג המועבר הוא int, ואז אנחנו רוצים שהסוג המוחזר יהיה double.

דוגמה ה (תיקיה 4)

אנחנו רוצים לבנות שיטה כללית של חילוק, שתבצע חילוק בין עצמים מסוג T רק אם החילוק בטוח - כלומר, רק אם אפשר לחלק עצמים מסוג T, ואין חשש של חלוקה ב-0. אפשר לבנות תבנית-מחלקה המכילה רק enum (ראו במצגת) או משתנים סטטיים (ראו בתיקיה 3), המייצגים את התכונות של חילוק לגבי הסוג T - האם בכלל אפשר לחלק, ואם כן, האם אפשר לחלק באפס. בתבנית הכללית, אי-אפשר לחלק, ואי-אפשר לחלק באפס. במקרה הפרטי עבור int,int, אפשר לחלק, אבל אי-אפשר לחלק באפס. במקרה הפרטי עבור double,double, אפשר לחלק גם באפס (התוצאה היא אינסוף חיובי או שלילי).

תיכנות-על (meta-programming)

מנגנון יצירת מקרים פרטיים של תבניות יכול לשמש אותנו לכתיבת תוכניות שלמות שרצות בזמן הקומפילציה. אפשר אפילו להשתמש ברקורסיה!

במצגת יש דוגמה לתבנית רקורסיבית פשוטה: התבנית Pow3 מקבלת מספר שלם ומעלה אותו בחזקת 3. הרקורסיה מתבצעת בתבנית הכללית - היא מכפילה ב-3 וקוראת לתבנית עם הפרמטר פחות 1. תנאי העצירה מתבצע ע"י יצירת מקרה פרטי שבו הפרמטר שווה 0.

דוגמה קצת יותר שימושית יש בתיקיה 5 - חישוב נומרי של הנגזרת ה-n של פונקציה כללית כלשהי.

זו גם הזדמנות טובה לחזור על ציורים, פונקטורים וביטויי למדא.

ביטויי למדא

ביטוי למדא הוא ביטוי היוצר עצם שיש לו סוגריים עגולים. ניתן להעביר אותו כפרמטר לפונקציות שעובדות על פונקציות.

לדוגמה, נניח שאנחנו רוצים לכתוב פונקציה שמציירת פונקציה. אפשר לכתוב את הכותרת שלה כך:

```
template <typename Function>
plot(Function f,...)
```

בגוף הפונקציה תהיה שורה שתחשב את ערך ה-y המתאים לערך x מסויים, כדי שנדע איפה צריך לשים נקודה:

```
y = f(x)
```

איך מפעילים את הפונקציה plot? יש כמה דרכים. אפשר להעביר לה פונקציה ממש, למשל:

```
double sqr(double x) {return x*x;}
```

ואז בתוכנית הראשית לכתוב:

```
plot(sqr,...);
```

אפשרות שניה היא ליצור מחלקה שיש לה אופרטור סוגריים עגולים, ולהעביר עצם מהמחלקה הזאת.

אפשרות שלישית היא להעביר ביטוי למדא. הביטוי מוגדר ע"י ריבוע שאחריו סוגריים עגולים ואחריהם סוגריים מסולסלים עם גוף הפונקציה:

```
plot([](double x) { return x*x; });
```

ראו דוגמה בתיקיה 5.

מקורות

- מצגות של אופיר פלא.
- Peter Gottschling, "Discovering Modern C++", chapter 3.
- ג'יימס מקנילס "יורד" על תבניות, על הסוגר המסולסל, ועל מצביעים לפונקציות: https://www.youtube.com/watch?v=6eX9gPithBo

סיכם: אראל סגל-הלוי.