הספריה הסטנדרטית

לפני כ-20 שנה החליטו להוסיף לשפת ++C הבסיסית, ספריה הבנויה מעליה ומשתמשת בעיקר בתבניות (template). הספריה נקראת STL - Standard Template Library, והיא כיום באה כחלק בלתי נפרד מהשפה.

מושגים

לפני שניכנס לפרטי הספריה, ניזכר בעיקרון חשוב הקשור לתבניות. כל תבנית שאנחנו יוצרים, מגיעה לתהליך של קומפילציה רק כאשר אנחנו משתמשים בה עם סוגים מסויימים. כדי שהתבנית תתקמפל, הסוגים צריכים לקיים דרישות מסויימות. אוסף של דרישות על סוג נקרא "מושג" - "concept".

לדוגמה, נניח שיש לנו תבנית-פונקציה המחשבת מינימום. הפונקציה עובדת רק על סוגים שיש להם אופרטור "קטן מ-". המושג " LessThanComparable" מציין כל סוג שיש לו אופרטור ":קטן מ-". לכן, אם אנחנו כותבים תבנית של פונקציית מינימום עם פרמטר-סוג T, אנחנו יכולים לכתוב בתיעוד שלה שהסוג T צריך להיות "LessThanComparable".

"מושג" בשפת ++C דומה ל- "ממשק" ב-Java: גם בשפת Java יכולנו להגדיר ממשק בשם LessThanComparable עם שיטה מופשטת בשם lessthan ולהגדיר פונקציית מינימום המקבלת פרמטרים מסוג LessThanComparable. אבל יש שני הבדלים:

1. ב-Java, רק מחלקות יכולות לממש ממשקים. לכן, אם כתבנו פונקציית מינימום על עצמים מסוג LessThanComparable, אז היא לא תעבוד על מספרים שלמים. בנוסף, אם כתבנו פונקציית מינימום על עצמים מסוג LessThanComparable, אז היא לא תעבוד על עצמים אחרים מספריה אחרת שלא מכירים את הממשק LessThanComparable. לעומת זאת, ב ++C המושג " LessThanComparable" משמש לתיעוד בלבד - גם מספר טבעי משתייך למושג הזה כי יש לו אופרטור "קטן מ-". גם מי שאינו מכיר כלל את המושג LessThanComparable יכול להשתייך למושג הזה אם יש לו אופרטור "קטן מ-".

2 החיסרון הוא, שב-++C הודעות השגיאה קשות יותר להבנה. ב-Java הקומפיילר מזהיר אותנו בפירוש כשאנחנו מנסים להפעיל פונקציה עם פרמטר שאינו מממש את הממשק LessThanComparable; ב-++C, הקומפיילר יצעק רק כשיראה שאנחנו מנסים לגשת לאופרטור "קטן מ-" שלא קיים. כתוצאה מכך הודעת השגיאה עלולה להיות קשה יותר להבנה.

בתיעוד של הספריה התקנית, מוגדרים כמה מושגים:

- LessThanComparable - מכילים אופרטור "קטן מ-".
- EqualityComparable - מכילים אופרטור "=="
- Assignable - מכילים בנאי מעתיק ואופרטור השמה (כברירת מחדל כל הסוגים הם כאלה, אלא-אם-כן מחקנו להם את הבנאי המעתיק ו/או את אופרטור ההשמה, או שהפכנו אותם לפרטיים).

רכיבי הספריה התקנית

הרכיבים העיקריים של הספריה התקנית הם: מיכלים, איטרטורים, אלגוריתמים, פונקטורים, מתאמים, זרמים ומחרוזות.

מיכלים, איטרטורים ואלגוריתמים קשורים זה לזה באופן הבא:

- כל **מיכל** מגדיר **איטרטורים** המאפשרים לעבור על כל הפריטים במיכל.
- כל **אלגוריתם** מקבל כקלט **איטרטורים** המגדירים את התחום שבו האלגוריתם צריך לעבוד.

שימו לב - אין קשר ישיר בין אלגוריתמים למיכלים. לכאורה, היינו חושבים שהקלט של אלגוריתם (למשל לסידור) צריך להיות מיכל. אבל, אילו היינו מגדירים כך, היינו צריכים לכתוב את האלגוריתם מחדש לכל סוג של מיכל. עם `n` אלגוריתמים ו-`m` מיכלים, זה יוצא `O(mn)` עבודה.

לעומת זאת, בשיטת האיטרטורים אנחנו צריכים לממש כל אלגוריתם פעם אחת, ולממש איטרטורים לכל מיכל, סה"כ `O(m+n)` עבודה.

מיכלים

ההגדרה של הספריה התקנית קובעת שהמיכלים מכילים **עותקים** של עצמים - ולא **קישורים** לעצמים (בניגוד לג'אבה). המשמעות:

- אפשר להכניס למיכל רק עצם המשתייך למושג `Assignable` - כלומר יש לו אופרטור השמה ובנאי מעתיק.
- בכל פעם שמכניסים עצם למיכל, נבנה עצם חדש; בכל פעם שמפרקים מיכל, מתפרקים כל העצמים הנמצאים בו.

יש שני סוגים עיקריים של מיכלים:

- סדרתיים - וקטור, רשימה... - שומרים פריטים לפי סדר ההכנסה שלהם
- אסוציאטיביים - קבוצה, מפה... - שומרים פריטים לפי הסדר הטבעי שלהם (המוגדר ע"י אופרטור קטן מ-).

ניתן לראות טבלת השוואה מפורטת בין כל המיכלים באתר
`http://www.cplusplus.com/reference/stl/`

מיכלים סדרתיים

המיכלים הסדרתיים נבדלים בסיבוכיות הזמן הנדרשת לביצוע פעולות שונות:

- `list` - רשימה מקושרת - זמן הכנסה בהתחלה/אמצע/סוף הוא קבוע (אם יש לנו איטרטור מתאים), אבל זמן הגישה לאיבר באמצע הרשימה הוא ליניארי.
- `vector` - וקטור - זמן הכנסה בהתחלה/אמצע הוא ליניארי, זמן הכנסה בסוף קבוע בממוצע, וזמן הגישה לאיבר באמצע הוא קבוע.

- deque - תור דו-כיווני - זמן הכנסה בהתחלה/סוף קבוע, וגם זמן הגישה לאיבר באמצע הוא קבוע, אבל פחות יעיל מוקטור.

וקטור

וקטור - vector<T> - ממומש כבלוק רציף של עצמים מסוג T. הבלוק גדל בצורה דינמית כשמוסיפים לו עצמים. יש שתי שיטות להוסיף עצם לסוף של וקטור:

- push_back - מקבלת עצם מסוג T ומעתיקה אותו לתא חדש בסוף הוקטור, ע"י שימוש בבנאי מעתיק.
- emplace_back - מקבלת פרמטרי-איתחול לעצם מסוג T, ומשתמשת בהם כדי לבנות עצם חדש בסוף הוקטור, ע"י שימוש בבנאי המתאים. שיטה זו יעילה יותר מהראשונה כי היא חוסכת את הצורך ליצור עצם זמני - אנחנו יוצרים את העצם ישירות במקום שלו (ראו הדגמה בתיקיה 1).

כדי לייעל את פעולת ההכנסה, הוקטור מקצה מקום בזיכרון מעבר למספר העצמים שיש בו. מספר העצמים שיש בוקטור נקרא **גודל הוקטור** - size. מספר העצמים שיש להם מקום בוקטור נקרא **קיבולת הוקטור** - capacity. הגודל תמיד שווה או קטן מהקיבולת. כשמוסיפים עצם בסוף הוקטור, יש שתי אפשרויות -

- האפשרות הקלה היא שהגודל לאחר ההוספה עדיין שווה או קטן מהקיבולת. במקרה זה צריך רק לבנות/להעתיק את העצם החדש למקום הפנוי בסוף הוקטור.
- האפשרות הקשה היא שהגודל לאחר ההוספה גדול יותר מהקיבולת. במקרה זה צריך להגדיל את הקיבולת: לאתחל בלוק עם קיבולת גדולה יותר ולהעתיק את הבלוק הישן לבלוק החדש ולשחרר את הבלוק הישן.

מקובל להגדיל את הקיבולת פי 2 בכל פעם; אפשר להוכיח שבמצב זה, הזמן הדרוש להכניס n עצמים הוא בערך 2n, כלומר הזמן הממוצע להכנסת עצם אחד הוא קבוע.

שימו לב: העצמים מ-0 עד (גודל-1) הם מאותחלים, אבל העצמים מ(גודל) עד (קיבולת-1) הם לא מאותחלים. אמנם הם שמורים עבור הוקטור, אבל הערך שלהם לא מוגדר.

כדי לגשת לעצמים בוקטור, אפשר באופרטור [] או בשיטה at. אופרטור [] לא בודק שהאינדקס קטן מהגודל; השיטה at כן בודקת.

אתחול וקטור: וקטור בלי פרמטרים מאותחל לוקטור בגודל 0 (אבל הקיבולת יכולה להיות גדולה מאפס - תלוי במימוש). וקטור עם פרמטר אחד מאותחל לוקטור בגודל הנתון; כל האיברים מ-0 עד (גודל-1) מאותחלים ע"י הפעלת הבנאי בלי פרמטרים.

אם מעבירים פרמטר שני, הוא משמש לאיתחול כל העצמים בין 0 לבין (גודל-1). אפשר גם לאתחל כל עצם בוקטור עם פרמטרים אחרים, ע"י שימוש בסוגריים מסולסלים.

סוגים קשורים לוקטור: לכל וקטור יש כמה טיפוסים הקשורים אליו, ואפשר לגשת אליהם בעזרת "ארבע נקודות" - ::

- value_type - הסוג של כל אחד מהעצמים בוקטור (שווה לסוג T המועבר כפרמטר לוקטור.).
- reference - רפרנס לעצם בוקטור (שווה ל T&).

- `const_reference` - רפרנס לעצם קבוע (שווה ל `const T&`).
- `iterator` - איטרטור על הוקטור.

הכנסת איברים באמצע הוקטור - השיטה `insert` מקבלת עצם בנוי מסוג T, ואיטרטור לתוך הוקטור, ומכניסה את העצם הבנוי במקום שעליו מצביע האיטרטור. השיטה `emplace` מקבלת פרמטרי איתחול לבנאי של T, ובונה בעזרתם עצם חדש במקום שעליו מצביע האיטרטור. השניה יעילה יותר כי היא חוסכת את יצירת העצם הזמני. אבל שתיהן צריכות להזיז חלק גדול מהאיברים בוקטור ולכן הן לוקחות זמן ליניארי בגודל הוקטור.

תור דו-כיווני

תור דו-כיווני - `deque` - מאפשר להכניס עצמים גם בהתחלה וגם בסוף בצורה יחסית יעילה. איך הוא עושה את זה? יש כמה מימושים, אחד המימושים הוא: וקטור של וקטורים. הוקטור הראשי מכיל פוינטרים לוקטורים המשניים, ושומר מקום פנוי גם בהתחלה וגם בסוף.

הגישה היא בזמן קבוע - הולכים לוקטור הראשי, משם לוקטור המשני המתאים, ושם מוצאים את הפריט בזמן קבוע.

הוספה בהתחלה או בסוף - בזמן קבוע אם יש מקום בוקטור המשני הראשון או האחרון. אם אין מקום - אז צריך ליצור בלוק ראשון/אחרון חדש, ולהוסיף פוינטר לוקטור הזה בהתחלה/בסוף של הוקטור הראשי. זה עלול לדרוש מאיתנו להעתיק את הוקטור הראשי, אבל אין צורך להעתיק את הוקטורים המשניים - כך אפשר ליצור `deque` גם של פריטים בלי בנאי מעתיק או אופרטור השמה.

מיכלים אסוציאטיביים

מיכל אסוציאטיבי הוא מיכל שבו ניתן לגשת לנתונים לפי מפתחות. המימושים המקובלים למיכלים אסוציאטיביים הם: עצי חיפוש מאוזנים (למשל עץ אדום-שחור), או טבלאות עירבול. סוגים של מיכלים אסוציאטיביים הם:

- `set` - קבוצה - מכילה רק מפתחות; כל מפתח פעם אחת בלבד.
- `map` - מפה - מתאימה מפתחות לערכים; כל מפתח פעם אחת בלבד (עם ערך אחד בלבד).
- `multiset, multimap` - כנ"ל, רק שכל מפתח יכול להופיע כמה פעמים.

(חידה: נניח שיש לכם *set,map*. איך תממשו *multiset,multimap*?).

מיכלים אסוציאטיביים מסודרים

ניתן להגדיר **סדר** על המפתחות במיכל אסוציאטיבי. כברירת מחדל, מיכל אסוציאטיבי מסודר משתמש באופרטור "קטן מ-".

ניתן להגדיר סדר שונה. לשם כך צריך להשתמש באובייקטים המציינים פונקציות - בהרצאות קודמות קראנו להם "פונקטורים".

"פונקטור" הוא כל עצם שאפשר להשתמש כמו כמו שמשתמשים בפונקציה. בפרט: מצביע לפונקציה, עצם ממחלקה עם אופרטור סוגריים (), או ביטוי למדא.

כדי ליצור מיכל אסוציאטיבי עם סדר שונה מהרגיל, מעבירים את המחלקה של הפונקטור המתאים כפרמטר לתבנית. למשל, עבור סידור מספרים בסדר יורד אפשר להגדיר את המחלקה:

```
struct SederYored {
    bool operator()(int x, int y) {return x>y;}
};
```

ואז בתוכנית הראשית לכתוב:

```
set<int, SederYored> s1;
```

המספרים שנכניס ל-s1 יהיו מסודרים לפי אופרטור-סוגריים של המחלקה SederYored.

בספריה התקנית כבר הגדירו מחלקות עם אופרטור-סוגריים מתאים, המתאימות לכל מחלקה שיש לה אופרטור-קטן-מ או אופרטור-גדול-מ. למשל, השורה:

```
set<int,less<int>> s1;
```

יוצרת קבוצה שבה הפריטים מסודרים מהקטן לגדול (לפי אופרטור קטן-מ שלהם), והשורה:

```
set<int,greater<int>> s1;
```

יוצרת קבוצה שבה הפריטים מסודרים מהגדול לקטן (לפי אופרטור גדול-מ שלהם).

ברירת המחדל היא היא less<T>, למשל אם כותבים:

```
set<int> s1;
```

זה כמו לכתוב:

```
set<int,less<int>>
```

הכנסת פריטים למפה

למפה יש אופרטור סוגריים מרובעים המשמש לקריאה וכתיבה של נתונים המתאימים למפתחות. במפה האופרטור הזה משמש גם כדי להוסיף מפתחות חדשים. למשל, אם כותבים m["a"] והמפתח "a" עדיין לא קיים - הוא ייווצר (זה בניגוד לוקטור שם גישה לאינדקס שאינו קיים לא יוצרת שום דבר חדש).

ראו בתיעוד של map באתר cplusplus.com.

למדנו על מיכלים; בשבוע הבא נלמד על שני הרכיבים המשלימים - איטרטורים ואלגוריתמים.

מקורות

- מצגות של אופיר פלא.
- Peter Gottschling, "Discovering Modern C++", chapter 4.
- תיעוד הספריה התקנית: http://www.cplusplus.com/reference/stl.
- השוואת ביצועים בין deque לבין וקטור: https://www.codeproject.com/Articles/5425/An-In-Depth-Study-of-the-STL-Deque-Container

- על טעויות נפוצות במפות ומבני-נתונים נוספים, ולמה חשוב לכתוב שיטות const (מפי מהנדס בכיר בפייסבוק): https://www.youtube.com/watch?v=lkgszkPnV8g

סיכם: אראל סגל-הלוי.