מהי שפת ++C ולמה צריך אותה?

++C היא הרחבה של שפת סי עם הרבה יסודות שאתם מכירים משפת ג'אבה.

היא כוללת את כל מה שיש בשפת סי עם המון תוספות, למשל:

- תיכנות מונחה עצמים - מחלקות ואובייקטים;
- תיכנות גנרי בתבניות (template);
- העמסת פונקציות ואופרטורים (ראו בתוכנית הדוגמה - תיקיה 1 וגם תיקיה 5);
- בדיקות טיפוסים חזקות בזמן קומפילציה;
- מנגנון לניהול זיכרון ובדיקות תקינות בזמן ריצה.

שפת ++C היא אחת משפות-התיכנות הקשות ביותר. התיעוד של השפה מזכיר לפעמים חוזה משפטי - המון מקרים ותתי-מקרים ומקרי-קצה, המון כפילויות ודרכים רבות ושונות לעשות אותו הדבר, עם הבדלים דקים בתוצאה. התחביר במקרים רבות קשה לקריאה ולהבנה. היתרון הוא, שמי שיצליח להתמודד עם ++C, יצליח כנראה להתמודד עם כל שפת-תיכנות אחרת...

++C לעומת Java

שפת ג'אבה פותחה אחרי ++C. היא דומה לה אבל הרבה יותר פשוטה. מתכנני השפה החליטו לקחת מ++C רק את החלקים הטובים בעיניהם (כגון תיכנות מונחה עצמים), ולזרוק את החלקים הקשים והמבלבלים. שפת ג'אבה אכן הרבה יותר נוחה לתיכנות ולהבנה. אם כך מדוע בכלל כדאי להשתמש ב++C? כמה סיבות.

א. **הבנה**. הרבה שפות-תיכנות כתובות ב-C בשילוב ++C. בפרט, המכונה הוירטואלית של ג'אבה, וכן מערכות הפעלה כגון חלונות, לינוקס, מק ואנדרואיד. גם במערכות תוכנה מודרניות גדולות כגון גוגל ופייסבוק יש שילוב של ++C עם שפות נוספות. גם מי שלא מתכנת ב++C צריך להבין את השפה כדי להבין מה קורה מאחרי הקלעים של השפה שהוא כותב בה.

לתכנת ++C זה כמו להיות נהג מרוצים. נהג כזה לא רק יודע לנהוג. הוא מכיר את מבנה הגוף שלו ויודע איפה מרכז הכובד, הוא יודע מה השפעת זוית הישיבה שלו על הנהיגה, מה ההשפעה של שיפוע הכביש והחיכוך שלו על הנהיגה. הוא מעוניין לנהוג באופן שצורך דלק בצורה מינימלית כדי שיספיק לו למירוץ, וששוחק את הגלגלים בצורה מינימלית, כדי שלא יצטרך לעצור ולהחליף גלגלים באמצע המירוץ. זה מתכנת ++C. לעומת זאת, מתכנת java ומעלה, למד נהיגה על רכב אוטומט. זה נכון שיש סיכוי שהוא יהנה יותר מהנוף, אבל ברור שהוא פחות מבין איך אפשר לנצל בצורה המקסימלית את הרכב שהוא נוהג עליו. הוא ממש לא חונך לזה.

ב. **זיכרון**. בשפת ++C אנחנו יכולים לכתוב תוכנות עם צריכת-זיכרון הדוקה וחסכונית יותר. ניתן לראות הדגמה בתיקיה 2. יש שם שתי תוכנות - אחת ב++C ואחת בג'אבה. שתיהן עושות אותו הדבר בדיוק: יוצרות מערך עם כ-125 מיליון עצמים מסוג "נקודה" (Point), כאשר ב"נקודה" יש שני מספרים שלמים (int). כיוון שמספר שלם דורש 4 בתים, אפשר לשער שצריכת הזיכרון של התוכנה תהיה כמיליארד

בתים (8 כפול 125 מיליון), ואכן זה מה שקורה ב++C. לעומת זאת, בג'אבה צריכת הזיכרון הרבה יותר גבוהה - לפחות 2.5 מיליארד (אם התוכנה לא קורסת בגלל מחסור בזיכרון)! מדוע? כמה סיבות:

- בג'אבה, כל "עצם" הוא למעשה פוינטר לעצם שנמצא בזיכרון. לכן, המערך שלנו כולל 125 מיליון פוינטרים. כל אחד מהם תופס לפחות 4 בתים - כבר הלכו חצי מיליארד, עוד לפני שבכלל יצרנו את הנקודות. לעומת זאת, ב++C פוינטר הוא רק מה שהוגדר בפירוש כפוינטר; כל שאר העצמים תופסים רק את הזיכרון של עצמם בלי פוינטרים מיותרים.
- בג'אבה, לכל עצם יש, בנוסף לשדות הגלויים שלו, גם כמו שדות סמויים. לדוגמה, יש שדה שנקרא "מצביע לטבלת מתודות וירטואליות", שבעזרתו המכונה של ג'אבה בוחרת איזו מתודה להריץ במצב של ירושה. המושג קיים גם בשפת ++C, אלא שב++C הקומפיילר יוצר אותו רק כשיש בו צורך - רק כשיש ירושה עם מתודות וירטואליות (נלמד בהמשך הקורס).

הסיסמה של ++C היא "לא השתמשת - לא שילמת". לכן היא מתאימה במיוחד למערכות שבהן כל טיפת זיכרון חשובה, למשל מערכות מוטמעות (embedded).

ג. **זמן**. בשפת ++C אפשר לכתוב תוכנות זמן-אמת (real-time), כלומר, תוכנות שמגיבות מהר לאירועים. לעומת זאת, בשפת ג'אבה קורים לפעמים דברים שיכולים לגרום לעיכובים לא-צפויים. למשל, **איסוף זבל** (garbage collection). בג'אבה, אנחנו יוצרים עצמים על-ידי new ולא מתעניינים בשאלה מה קורה איתם אחר-כך (כשאנחנו כבר לא צריכים אותם). המכונה של ג'אבה יודעת לזהות מתי אנחנו כבר לא צריכים להשתמש בעצם מסויים, ולשחרר את הזיכרון שהוא תופס כדי שיהיה פנוי לעצמים אחרים. התהליך הזה נקרא "איסוף זבל". זה מאד נוח לנו כמתכנתים, הבעיה היא שהמכונה עלולה להחליט שהגיע הזמן "לאסוף זבל" בדיוק ברגע הלא מתאים. חישבו למשל על תוכנה שמנהלת רכב אוטונומי, שמחליטה "לאסוף זבל" בדיוק כשהנהג מנסה לפנות ימינה... לא נעים. בשפת ++C, איסוף זבל לא קורה באופן אוטומטי, ולכן השפה מתאימה במיוחד למערכות זמן-אמת.

מצד שני, העובדה שאין איסוף-זבל אוטומטי משמעה שאנחנו צריכים להיות אחראים לאסוף את הזבל של עצמנו (כמו בטיול שנתי...) אנחנו צריכים לוודא, שכל עצם שאנחנו יוצרים בזיכרון, משחרר את כל הזיכרון שהוא תופס. בהמשך נלמד איך לעשות את זה.

++C לעומת סי

בשפת ++C נוספו סוגים חדשים שהיו חסרים בסי:

- מחרוזת - string; אין צורך יותר להשתמש ב-char* (אלא אם כן צריך להשתמש בפונקציות ספריה ישנות).
- בוליאני - bool; עדיף מלהשתמש בהשוואה של מספר שלם לאפס.
- מחלקה עם ערכים קבועים - enum class - בטוח יותר מה-enum של סי; הקומפיילר לא מאפשר תרגום אוטומטי מ/למספר שלם; ראו דוגמה בתיקיה 4 (הערה: enum של ג'אבה הוא כמו enum class של ++C).

מרחבי-שם

אחד החידושים החשובים בשפת ++C הוא הוספת מרחבי-שם (namespaces).

בשפת סי, כל התוכנית נמצאת במרחב-שם אחד. זה אומר שאי-אפשר להגדיר שני משתנים או שני קבועים עם אותו שם - זה יגרום לשגיאת קימפול.

מה קורה אם אנחנו משתמשים בשתי ספריות שונות, שקיבלנו משני אנשים שונים, וכל אחד מהם הגדיר משתנה עם אותו שם? במקרה זה לא נוכל להשתמש בשתי הספריות יחד - תהיה התנגשות.

מנגנון מרחבי-השם של ++C נועד למנוע בעיה זו, ולאפשר לבנות פרוייקטים גדולים הכוללים ספריות שונות ממקומות שונים. בשני מרחבי-שם שונים, אפשר להגדיר משתנים עם אותו שם, ולא תהיה שגיאת קומפילציה - הקומפיילר פשוט ייצור שני משתנים שונים (כמו שני אנשים עם אותו "שם פרטי" ועם "שם משפחה" שונה). לדוגמה, הקוד הבא הוא חוקי:

```
namespace abc{
     int x = 123;
      void printx() { std::cout << x << std::endl; }
};

namespace def{
     int x = 456;
      void printx() { std::cout << x << std::endl; }
};
```

כשנמצאים בתוך מרחב-שם מסויים, אפשר לגשת לכל המשתנים שלו כרגיל בלי להזכיר את "שם המשפחה".
אבל כשנמצאים מחוץ למרחב-השם (למשל ב-main), ורוצים לגשת למשתנה או לפונקציה שנמצאים במרחב-שם מסויים, צריך לשים לפניו את "שם המשפחה" עם פעמיים נקודתיים, למשל:

```
int main() {
    abc::printx();
    def::printx();
}
```

אם משתמשים באותו מרחב-שם הרבה פעמים, אפשר לחסוך כתיבה ע"י שימוש במילה השמורה using. למשל, התוכנית הבאה זהה לקודמת:

```
using namespace abc;
int main() {
     printx();
    def::printx();
}
```

חידה: מה יקרה אם נכתוב גם using namespace def באותה תוכנית? (ראו דוגמה בתיקיה 7).

הערה: אין שום תלות בין מרחבי-שם לבין קבצים. אפשר לשים כמה מרחבי-שם בקובץ אחד, או מרחב-שם אחד בכמה קבצים.

מרחב-השם השימושי ביותר בשפת ++C הוא std. הספריה התקנית של ++C, שנלמד עליה בהרחבה בהמשך הקורס, נמצאת כולה במרחב-שם זה.

בפרט, אובייקטי הקלט והפלט, cin ו cout, נמצאים במרחב-שם זה. לכן, כדי לגשת אליהם מהתוכנית הראשית, אחרי שמכלילים את קובץ-הכותרת המתאים (<iostream>), צריך לשים לפני האובייקט std::, למשל std::cout וכו'.

כדי לחסוך כתיבה, אפשר לכתוב בראש התוכנית הראשית שלנו using namespace std.

שימו לב: לא מומלץ להשתמש בפקודה זו בקובץ-כותרת (h) שאנחנו מתכוונים להכליל בקבצים אחרים, כי אז כל המשתנים והפונקציות של הספריה התקנית יעברו למרחב-השם הראשי, ועלולים להתערבב עם המשתנים והפונקציות שמוגדרים בקובץ עצמו. אבל אפשר להשתמש ב using namespace std בתוכנית הראשית.

ספריות גדולות של חברות אחרות נמצאות במרחבי-שם אחרים. לדוגמה, בספריה folly של פייסבוק, כל השמות הם במרחב-שם folly, למשל
https://github.com/facebook/folly/blob/master/folly/stop_watch.h

אחד הכללים החשובים בבניית ספריות מורכבות הוא "לא לזהם את מרחב השם הגלובלי" - don't pollute the global namespace - לשים את כל הפונקציות שלכם במרחב-שם מיוחד כך שלא תהיה התנגשות עם פונקציות של ספריות אחרות.

טיפול בשגיאות

השגיאות העלולות לקרות בתוכנית מתחלקות לשני סוגים עיקריים:

א. שגיאות שעלולות לקרות בזמן ריצה תקינה של התוכנית, למשל כתוצאה מקלט לא תקין של המשתמש;

ב. שגיאות הנובעות מטעות של המתכנתים (כלומר שלנו).

שגיאות מסוג א נקראות "חריגות", וכדי לטפל בהם, זורקים חריגה. למדתם על זה בJava והמנגנון קיים גם ב++C. בשפת ++C אפשר לזרוק מה שרוצים - לא דווקא אובייקט מסוג "חריגה". למשל, אפשר לכתוב (ראו תיקיה 5):

```
if (x<0)  throw string("x should be at least 0");
```

אפשר לתפוס את השגיאה ולהדפיס אותה, למשל כך:

```
try {
     func(-5);
} catch (string message) {
     cout << "   caught exception: " << message << endl;
}
```

מקובל יותר לזרוק עצמים המוגדרים בספריה התקנית stdexcept:

```
#include <stdexcept>

...

if (x<0)  throw std::out_of_range("x should be at least 0");
```

והתפיסה נראית כך:

```
     try {
          func(-5);

     } catch (const std::exception& ex) {

          cout << "   caught exception: " << ex.what() << endl;

     }
```

שגיאות מסוג ב נקראות כידוע "באגים". הן לא אמורות להתקיים בתוכנה תקינה, אבל תוך כדי פיתוח הן עלולות להופיע. כדי לתפוס אותן בצורה נוחה, משתמשים ב-assert (ראו תיקיה 6). למשל, נניח שיש לנו פונקציה שאמורה להחזיר ערך חיובי, אבל משום-מה היא מחזירה לפעמים ערך שלילי. כדי לתפוס את השגיאה ברגע שהיא קורה, אפשר לשים פקודת assert בסוף הפונקציה, למשל:

```
     assert (result>=0);
```

אם בנקודה זו התנאי לא יתקיים - הביצוע ייפסק עם הודעת-שגיאה מתאימה.

בניגוד לחריגות, באגים לא אמורים להתקיים בתוכנה הסופית, ולכן אנחנו לא רוצים לבזבז זמן על הבדיקות הללו לאחר שסיימנו לתקן את הבאגים. אפשר בבת-אחת לבטל את כל הבדיקות מסוג assert ע"י הגדרת משתנה-קומפילציה בשם NDEBUG. הדרך הנוחה ביותר להגדיר אותו היא ע"י פרמטר לקומפיילר, למשל:

```
clang++-5.0 -DNDEBUG ...
```

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- על ההבדלים בין c++ לבין java, ראו https://cseducators.stackexchange.com/a/4189/1873
- על יסודות הסגנון של C++, ראו https://www.youtube.com/watch?v=xnqTKD8uD64

סיכם: אראל סגל-הלוי. הערות תוספות והשלמות: מירי בן-ניסן.

העמסה בשפת C++

העמסת פונקציות

העמסה (overloading) היא מצב שבו יש כמה פונקציות עם אותו שם וארגומנטים שונים. הבחירה לאיזה פונקציה לקרוא מתבצעת ע"י הקומפיילר. למה זה חשוב? כי זה מאפשר לנו לקרוא לפונקציות בשמות קצרים וברורים, ולסמוך על הקומפיילר שיבחר את הפונקציה הנכונה לפי הצורך. הרבה מהתכונות המתקדמות של שפת C++ מסתמכות על העמסה, ולכן כדאי להכיר היטב את המנגנון.

איך הקומפיילר יודע איזה פונקציה לבחור? הוא עובד בכמה שלבים:
1. מציאת כל הפונקציות עם השם המתאים.
2. מתוך קבוצה 1, מציאת כל הפונקציות עם מספר הפרמטרים המתאים.
3. מתוך קבוצה 2, מציאת הפונקציות עם סוג הפרמטרים המתאים ביותר.

שלב 3 הוא הכי מסובך, כי הקריטריון להתאמה בסוג הפרמטרים לא תמיד ברור. לדוגמה ראו בתיקיה 5. הגדרנו שם פונקציה בשם power המחשבת חזקה של שני מספרים. כשהחזקה היא מספר שלם חיובי אפשר להשתמש ברקורסיה:

```
int power(int a, unsigned int b) {
      cout << " power of ints" << endl;
      return b==0? 1: a*power(a,b-1);
}
```

אבל כשהחזקה היא מספר ממשי צריך להשתמש בלוג:

```
double power(double a, double b) {
      cout << " power of reals" << endl;
      return exp(b*log(a));
}
```

הקריאה power(2,3) מגיעה לפונקציה הראשונה והקריאה power(2.0,3.5) מגיעה לפונקציה השניה. לאן תגיע הקריאה power(2,3.5)? היינו מצפים שהיא תגיע לפונקציה השניה, אבל אצלי זו שגיאת קומפילציה - הקומפיילר לא בטוח לאיזו פונקציה התכוונו, רמת ההתאמה היא זהה כי בשני המקרים צריך לבצע המרה (להמיר מספר שלם לממשי או להיפך).

לאן תגיע הקריאה power(2,-3)? היינו מצפים שהיא תגיע לפונקציה השניה כי החזקה שלילית, אבל אצלי היא מגיעה לפונקציה הראשונה ונכנסת לרקורסיה ארוכה מאד - המספר מינוס3 מומר למספר חיובי גדול - ממש לא מה שהתכוונו לעשות.

המסקנה: צריך מאד להיזהר בהעמסת פונקציות, במיוחד כשהן עם אותו מספר פרמטרים ועם פרמטרים מספריים.
ראו דוגמה בתיקיה 7.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.

סיכם: אראל סגל-הלוי.

מחלקות

המחלקות בשפת ++C הן שילוב והרחבה של המבנים הקיימים ב-C וב-Java.

בשפת C, הדרך המקובלת ליצור מבנים מורכבים היא להגדיר struct. המשמעות של struct היא פשוט אוסף של משתנים שנמצאים ברצף בזיכרון. אפשר גם להגדיר משתנים מורכבים יותר ע"י struct בתוך struct וכו'. אבל ב-struct אין שיטות - אם רוצים לעשות איתו משהו צריך לכתוב שיטות חיצוניות.

בשפת Java יש מחלקות (class) שאפשר לשים בהן גם משתנים וגם שיטות - זה אמור להפוך את הקוד לקריא יותר כי כל המידע והפעולות הדרושות למימוש העצם נמצאים במקום אחד. אבל, בניגוד למבנים של סי, בג'אבה המשתנים לא תמיד נמצאים ברצף בזיכרון. לדוגמה, נניח שאנחנו מגדירים מבנה של "נקודה" ובו שני מספרים שלמים, ואז מגדירים מבנה של "משולש" ובו שלוש נקודות. בסי, כל משולש כולל פשוט שישה מספרים הנמצאים ברצף בזיכרון, ותופס 24 בתים בדיוק. בג'אבה, כל משולש כולל שלושה **מצביעים** (pointers), כל אחד מהם מצביע לנקודה שבה יש שני מספרים. כלומר המספרים לא נמצאים ברצף בזיכרון. למה זה משנה?

- בלי מצביעים, התוכנה תופסת פחות זיכרון ודורשת פחות זמן כשניגשים למשתנים.
- כשכל המבנה נמצא באותו מקום בזיכרון, ניהול הזיכרון מהיר יותר. בפרט, כשמשתמשים בזכרון מטמון (cache), זה מאד יעיל שכל המבנה נטען בבת-אחת לאותו בלוק בזיכרון.

שפת ++C משלבת את היתרונות של שתי השפות:

- יש בה struct כמו ב-C, אבל אפשר לשים בו גם שיטות.
- יש בה class כמו ב-Java, אבל המשתנים נשמרים ברצף בזיכרון ולא ע"י פוינטרים (נראה בהמשך).

למעשה, ב-++C, ההבדל היחיד בין struct לבין class הוא בהרשאות הגישה: בראשון, כברירת מחדל, הרשאת הגישה היא public; בשני, כברירת מחדל, הרשאת הגישה היא private. מכאן שההגדרות הבאות שקולות לחלוטין (ראו דוגמה בתיקיה 1):

```
struct X {
    private:
          ...
}
===
class X {
          ...
}
```

וכן ההגדרות הבאות שקולות לחלוטין:

```
struct X {
          ...
```

```
}
===
class X {
     public:
          ...
}
```

מימוש שיטות – בפנים או בחוץ

יש שתי דרכים לממש שיטות של מחלקות ב-++C.

- דרך אחת היא לממש אותן ישירות בתוך המחלקה (בדומה לJava). זה נקרא מימוש "inline" (לא צריך לכתוב את המילה inline).
- דרך שניה היא לשים בתוך המחלקה רק **הצהרה** של השיטה, בלי גוף. את המימוש עצמו שמים מחוץ למחלקה. זה נקרא מימוש "out-of-line".

עקרונית, גם את המימוש החיצוני (out-of-line) אפשר לשים באותו קובץ (ראו תיקיה 0). אפשר גם לשים את כל המימוש בתוך המחלקה (inline), כמו בג'אבה. אבל, מקובל ורצוי מאד לשים את המימוש בקובץ נפרד. מקובל להגדיר כל מחלקה בשני קבצים (ראו תיקיה 1):

- קובץ הכותרת - בדרך-כלל עם סיומת hpp או h - כולל את הגדרת המחלקה, השדות והשיטות שלה - אבל בלי מימוש השיטות.
- קובץ התוכן - בדרך-כלל עם סיומת cpp - כולל את המימוש של השיטות.

למה זה עדיף? שתי סיבות:

1. הנדסת תוכנה. אם ניתן את המחלקה שלנו למישהו אחר, הוא ירצה לראות איזה שיטות יש בה, אבל לא יעניין אותו לדעת איך בדיוק מימשנו אותן. לכן הוא יסתכל בקובץ הכותרת, ועדיף שהקובץ הזה יהיה קטן ופשוט ככל האפשר.
2. זמן קומפילציה. במערכות תוכנה מורכבות, קומפילציה לוקחת הרבה זמן. בכל פעם שמשנים קובץ, צריך לקמפל מחדש את כל הקבצים שתלויים בו. כששמים את המימוש בקובץ נפרד, שינוי במימוש לא דורש קימפול מחדש של קוד שמשתמש בקובץ הכותרת.

בניית תוכנות עם כמה קבצי מקור

בתיקיה 1 אפשר לראות דוגמה פשוטה הכוללת מחלקה אחת (Complex). בתיקיה יש שלושה קבצי-מקור: Complex.hpp – הצהרת המחלקה, Complex.cpp – מימוש המחלקה, ו-main.cpp – התוכנית הראשית המשתמשת במחלקה.

כדי לבנות את המערכת, יש לתרגם כל קובץ cpp בנפרד לקובץ בינארי, ואז לקשר את הקבצים הבינאריים (כמו שלמדתם בקורס שפת C):

```
clang++-5.0 -std=c++17 Complex.cpp -o Complex.o
clang++-5.0 -std=c++17 main.cpp -o main.o
clang++-5.0 -std=c++17 Complex.o main.o
```

```
./a.out
```

שימו לב: התוכנית main.cpp משתמשת בקובץ הכותרת Complex.hpp, אבל היא לא צריכה להכיר את הקובץ Complex.cpp. תחשבו שהמחלקה Complex נכתבה ע"י מתכנת א, והתוכנית הראשית שמשתמשת בה main נכתבה ע"י מתכנת ב. מתכנת ב לא צריך לדעת איך מתכנת א מימש את כל השיטות של Complex – הוא צריך רק להכיר את הגדרת המחלקה, הנמצאת בקובץ Complex.hpp.
מתכנת א, שייצר את המחלקה, צריך לתת למתכנת ב רק את הקובץ Complex.hpp, וכן את הקובץ Complex.o (הבינארי) – הוא לא צריך לחשוף את פרטי המימוש הנמצאים בקובץ Complex.cpp.
זה יוצר מִידוּר – כל מתכנת יודע רק מה שהוא צריך לדעת ולא יותר.

מה קורה אם מעדכנים את הקובץ Complex.cpp? אז צריך לבנות מחדש את Complex.o:

```
clang++-5.0 -std=c++17 Complex.cpp -o Complex.o
```

ואז לקשר מחדש:

```
clang++-5.0 -std=c++17 Complex.o main.o
```

אבל, לא צריך לבנות מחדש את main.cpp – כי הוא לא משתמש ישירות ב Complex.cpp.
באותו אופן, אם משנים את main.cpp, צריך לבנות מחדש רק את main.o ולקשר – לא צריך לבנות מחדש את Complex.cpp.
אבל, אם משנים את Complex.hpp, צריך לבנות מחדש גם אם main.o וגם את Complex.o – כי שניהם משתמשים ב-Complex.hpp.
ההבחנה הזאת היא חשובה, כי זמן הקומפילציה של מערכות בשפת C++ יכול להיות מאד ארוך, במיוחד כשיש הרבה קבצים ובכל קובץ הרבה קוד. ראו: https://xkcd.com/303/
לכן כדאי שקובץ הכותרת יהיה קטן ויכלול כמה שפחות מימושים, כך שלא נצטרך לעדכן אותו בתדירות רבה מדי.

בניית תוכנות בעזרת "מייק" (make)

ראינו למעלה, שכאשר יש לנו תוכנה עם כמה קבצי-מקור, וחלק מהקבצים משתנים, אנחנו צריכים לקמפל מחדש את קבצי-המטרה שהמקורות שלהם התעדכנו. כשמעדכנים כמה קבצים בו זמנית, זה קשה לזכור מה בדיוק התעדכן ומה בדיוק צריך לבנות מחדש. התוכנה make היא תוכנה המבצעת משימה זו עבורנו.
דוגמה לבניית תוכנה בעזרת "מייק" נמצאת בתיקיה 2. יש שם מחלקה בשם Point (המורכבת משני קבצים – cpp, hpp כפי שלמדנו), וכן מחלקות נוספות המשתמשות בה (Rectangle, Triangle וכו'), וגם תוכנית ראשית (main).

כדי לעבוד עם מייק, צריך ליצור בתיקיה שלנו קובץ טקסט אחד בשם Makefile (בדיוק בשם זה).
בקובץ זה שמים כללים ליצירת "מטרות" מ"מקורות". לדוגמה, אחת ה"מטרות" שצריך ליצור היא Point.o. הכלל המתאים הוא:

```
Point.o: Point.cpp Point.hpp
    clang++-5.0 -std=c++17 --compile Point.cpp -o Point.o
```

הכלל חייב להתחיל בתחילת שורה (בלי רווחים מקדימים).
המילה שלפני הנקודתיים (במקרה זה Point.o) היא שם קובץ ה"מטרה".
המילים שאחרי הנקודתיים (במקרה זה Point.cpp Point.hpp) הם שמות קבצי ה"מקור" – הקבצים הדרושים על-מנת ליצור את המטרה.

השורות הבאות חייבות להתחיל בטאב (tab), וכל שורה כזאת מייצגת פקודה שצריך לבצע על-מנת ליצור את המטרה מהמקורות.
משמעות הכלל היא: "אם הקובץ Point.o לא קיים, או שהוא ישן יותר מאחד המקורות שלו, אז בצע את הפקודה בשורה למטה".
כדי להפעיל את הכלל, נמחק את הקובץ Point.o ונריץ את הפקודה הבאה (במסוף לינוקס):

```
make Point.o
```

הפקודה מריצה את הקומפיילר ובונה את הקובץ Point.o.
אם נריץ שוב make Point.o, נקבל הודעה האומרת ש-Point.o כבר מעודכן – לא צריך ליצור אותו שוב. "מייק" בודק את זמן העידכון האחרון של קובץ המטרה, ואם הוא מאוחר יותר מזמן העדכון האחרון של קבצי המקור – הוא חוסך את פקודת הקימפול – וכך חוסך לנו זמן.
אם נשנה את הקובץ Point.cpp (גם שינוי מינימלי כגון תוספת/מחיקת רווח) ונריץ שוב make Point.o, הפקודה שוב תריץ את הקומפיילר וכך תעדכן את המטרה Point.o.

ב-Makefile אפשר גם לשים מטרות התלויות במטרות אחרות. לדוגמה, הכלל:

```
a.out: Point.o main.o Rectangle.o Triangle.o Circle.o
    clang++-5.0 -std=c++17 Point.o main.o Rectangle.o Triangle.o Circle.o
```

אומר שהמטרה a.out (- קובץ ההרצה) תלויה בחמש מקורות – חמישה קבצים בינאריים. כל אחד מהמקורות הללו, הוא בעצמו מטרה באחד הכללים האחרים באותו Makefile. כאשר מריצים:

```
make a.out
```

ה"מייק" עובד באופן רקורסיבי: קודם-כל, הוא בונה כל אחד ואחד מהמקורות לפי הכלל המתאים (אם יש), ואחר-כך בונה את a.out.
כך למשל, אם נמחק את Point.o, או נעדכן את Point.cpp, ונריץ שוב make a.out, יתבצעו רק שתי פקודות – קומפילציה (כדי ליצור מחדש את Point.o) וקישור (כדי לעדכן את a.out):

```
clang++-5.0 -std=c++17 --compile Point.cpp -o Point.o
clang++-5.0 -std=c++17 Point.o main.o Rectangle.o Triangle.o Circle.o
```

אבל מה יקרה אם נעדכן את Point.hpp? - כיוון שכל המטרות בקובץ תלויות בו, "מייק" יבנה מחדש את כל המטרות – יתבצעו 5 פקודות קימפול ועוד פקודת קישור אחת. כפי שהסברנו למעלה, זו אחת הסיבות שכדאי לשים מימושים של שיטות בקובץ cpp ולא בקובץ הכותרת.

בתחילת קובץ ה-Makefile, יש כלל לבניית מטרה בשם all:

```
all: a.out
    ./a.out
```

משמעות הכלל הזה היא בדיוק כמו משמעות הכללים הקודמים שלמדנו: "בנה את a.out; בדוק אם קיים קובץ בשם all ואם הוא מעודכן; אם הוא לא מעודכן – הרץ את הפקודה בשורה הבאה". אבל יש הבדל אחד קטן – הקובץ all אף פעם לא קיים כי אנחנו לא יוצרים אותו. לכן, אם נכתוב:

```
make all
```

אז "מייק" תמיד יבנה את a.out ואז יריץ את a.out.

ב-Makefile יש עוד כלל המתייחס למטרה שאף פעם לא נוצרת:

```
clean:
    rm -f *.o a.out
```

משמעות הכלל הזה היא כמו כל הכללים הקודמים: "בדוק אם קיים קובץ בשם clean ואם הוא מעודכן; אם לא – הרץ את הפקודה בשורה הבאה". במקרה זה אין מקורות. וכיוון שהקובץ clean אף פעם לא נוצר, הפקודה make clean פשוט תריץ את הפקודה שמתחתיה ותמחק את כל הקבצים הזמניים בתיקיה.

פרט קטן נוסף לגבי make – כשמריצים make בלי פרמטרים, הוא תמיד מנסה לבנות את המטרה הראשונה בקובץ (במקרה שלנו, זו המטרה all). לכן, כשכותבים רק make, "מייק" יעדכן ויריץ את התוכנה שלנו עם כל העדכונים האחרונים – בדיוק מה שאנחנו רוצים.

להרחבה: בתיקיה הראשית של המאגר ariel-cpp-5779 יש קובץ בשם Makefile המשמש לבניה אוטומטית של קבצי pdf מתוך קבצי odt,odp במאגר זה. הקובץ חוסך לי המון זמן – אני מעדכן הרבה מסמכים ומצגות בו-זמנית, ואז בפקודה אחת make בונה את כל קבצי ה-pdf המתאימים כך שיישארו תמיד מעודכנים. אתם מוזמנים להיכנס לקובץ ולנסות להבין איך הוא עובד.

המצביע this

במימוש של שיטה, המילה השמורה this מכילה מצביע לעצם הנוכחי. ניתן להשתמש בה כמו שמשתמשים במצביעים, למשל:

```
Point::Point(int x, int y) {
      this->x = x;
      this->y = y;
}
```

(זה דומה לג'אבה פרט לכך שמשתמשים בחץ במקום בנקודה).

שדות סטטיים

ניתן להגדיר שדות ושיטות סטטיים (שייכים לכל המחלקה ולא לעצם מסויים) בעזרת המילה השמורה static. כדי לגשת אליהם מבחוץ, מקדימים להם את שם המחלקה ופעמיים נקודתיים, למשל Point::MAXX. אם מגדירים שדה סטטי שהוא גם קבוע (const), ניתן לאתחל אותו בהגדרת המחלקה; אחרת, יש לאתחל אותו מבחוץ (ראו דוגמה בתיקיה 3).

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- עוד על קבצי מייק: https://stackoverflow.com/a/2481326/827927

סיכם: אראל סגל-הלוי.

מחלקות - בניה ופירוק

בשפת ++C יש עקרון חשוב:

- כל עצם חייב לעבור תהליך של בניה (construction) ברגע שהוא נוצר; הבניה מתבצעת ע"י בנאי.
- כל עצם חייב לעבור תהליך של פירוק (destruction) ברגע שהוא מפסיק להתקיים; הפירוק מתבצע ע"י מפרק.

בנאים - constructors

בנאי הוא שיטה ששמה כשם המחלקה, ואין לה ערך-חזרה. כמו כל שיטה אחרת, ניתן לממש אותה בקובץ הכותרת או בקובץ המימוש. ראו דוגמאות בתיקיה 4.

כמו כל פונקציה, גם בנאים אפשר להעמיס. כלומר, אפשר להגדיר כמה בנאים עם פרמטרים שונים, והקומפיילר יקרא לבנאי הנכון לפי השימוש.

איך קוראים לבנאי? תלוי:

- אם זה בנאי עם פרמטרים, פשוט שמים את הפרמטרים אחרי שם העצם, למשל Point; p(10,20) קורא לבנאי של Point המקבל שני מספרים שלמים.
- אם זה בנאי בלי פרמטרים, לא שמים שום דבר אחרי שם העצם, למשל Point p; (לא לשים סוגריים ריקים - זה יגרום לשגיאת קימפול כי הקומפיילר עלול לחשוב שאתם מנסים להגדיר פונקציה...)

שימו לב - בניגוד לג'אבה - לא צריך להשתמש ב-new! המשתנה נוצר "על המחסנית" ולא "על הערימה" (אם נשתמש ב-new, המשתנה ייווצר על הערימה).

בנאי ללא פרמטרים

יש כמה סוגי בנאים שיש להם תפקיד מיוחד ב-++C. אחד מהם הוא **בנאי ללא פרמטרים** (parameterless constructor). אם (ורק אם) לא מגדירים בנאי למחלקה - הקומפיילר אוטומטית יוצר עבורה בנאי כזה; הוא נקרא default parameterless constructor.
בהתאם לגישה של ++C "לא השתמשת - לא שילמת", בנאי ברירת-מחדל של מחלקה פשוטה לא עושה כלום - הוא **לא מאתחל את הזיכרון** (בניגוד לג'אבה). לכן, הערכים לא מוגדרים - ייתכן שיהיו שם ערכים מוזרים שבמקרה היו בזיכרון באותו זמן.

בנאים נוספים נראה בהמשך.

מפרקים - destructors

אנחנו מגיעים עכשיו לאחד ההבדלים העיקריים בין ++C לג'אבה. כזכור, ב-++C ניהול הזיכרון הוא באחריות המתכנת. בפרט, אם אנחנו יוצרים משתנים חדשים על הערימה, אנחנו חייבים לוודא שהם משוחררים כשאנחנו כבר לא צריכים אותם יותר. לשם כך, בכל מחלקה שמקצה זיכרון (או מבצעת פעולות אחרות הדורשות משאבי מערכת), חייבים לשים **מפרק - destructor.**

מפרק הוא שיטה בלי ערך מוחזר, ששמה מתחיל בגל (טילדה) ואחריו שם המחלקה; ראו דוגמה בתיקיה 5 (למה גל? כי זה האופרטור המציין "not" של סיביות. למשל: 0~=1-).

חידה: האם אפשר להגדיר מפרק עם פרמטריס? האם אפשר לבצע העמסה (*overload*) למפרק? מדוע?

המתכנת אף-פעם לא צריך לקרוא למפרק באופן ידני; זוהי האחריות של הקומפיילר להכניס קריאה למפרק ברגע שהעצם מפסיק להתקיים. מתי זה קורה?

- אם העצם נוצר על המחסנית - העצם מפסיק להתקיים כשהוא יוצא מה-scope (יוצאים מהבלוק המוקף בסוגריים מסולסלים המכיל את העצם).
- אם העצם נוצר על הערימה בעזרת new - העצם מפסיק להתקיים כשמוחקים אותו בעזרת delete.

מה קורה כששוכחים לשים מפרק? המשאבים לא ישתחררו, וכתוצאה מכך תהיה דליפת זיכרון; ראו הדגמה בתיקיה 5.

האופרטורים new, delete

האופרטור new מבצע, כברירת מחדל, את הדברים הבאים:

- הקצאת זיכרון עבור עצם חדש מהמחלקה;
- קריאה לבנאי המתאים של המחלקה (בהתאם לפרמטרים שהועברו).

האופרטור delete מבצע, כברירת מחדל, את הדברים הבאים:

- קריאה למפרק של המחלקה (יש רק אחד - אין פרמטרים);
- שיחרור הזיכרון שהוקצה עבור העצם.

האופרטור []new מבצע, כברירת מחדל, את הדברים הבאים:

- הקצאת זיכרון עבור מערך של עצמים מהמחלקה;
- קריאה לבנאי ברירת-המחדל של כל אחד מהעצמים במערך.

האופרטור []delete מבצע, כברירת מחדל, את הדברים הבאים:

- קריאה למפרק של כל אחד מהעצמים במערך.
- שיחרור הזיכרון שהוקצה עבור המערך.

כשמאתחלים מערך ע"י []new, חייבים לשחרר אותו ע"י []delete. כאן הקומפיילר לא יציל אתכם משגיאה - אם תשתמשו ב-delete במקום []delete, הקוד יתקמפל, אבל הקומפיילר יקרא רק למפרק אחד (של האיבר הראשון במערך). כתוצאה מכך תהיה לכם דליפת זיכרון, או אפילו שגיאת ריצה (ראו תיקיה 6).

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.

סיכם: אראל סגל-הלוי.

הרכבה ואיתחול של מחלקות

הרכבה - composition

לעצם ב-++C יכולים להיות שדות שהם עצמם עצמים. דוגמה קלאסית: מחלקה Line שיש לה שני שדות מסוג Point גם בג'אבה זה אפשרי, אבל יש הבדל - ב-++C העצמים מסוג "נקודה" פשוט מונחים אחד ליד השני בעצם מסוג "קו", בעוד שבג'אבה העצם מסוג "קו" מכיל רק מצביעים לעצמים מסוג "נקודה". לכן ב-++C כשיוצרים עצם מסוג "קו", עוד לפני שמאתחלים אותו, כבר יש בו "נקודות". בעוד שבג'אבה הנקודות עדיין לא קיימות - יש רק מצביע המאותחל ל-null. ראו הדגמה בתיקיה 0.

למה זה משנה?

- בלי מצביעים, התוכנה תופסת פחות זיכרון ודורשת פחות זמן כשניגשים למשתנים.
- כשכל המבנה נמצא באותו מקום בזיכרון, ניהול הזיכרון מהיר יותר. בפרט, כשמשתמשים בזכרון מטמון (cache), זה מאד יעיל שכל המבנה נטען בבת-אחת לאותו בלוק בזיכרון.

מה קורה כשלעצמים המוכלים יש בנאים ומפרקים משל עצמם? הקומפיילר קורא להם לפי סדר פשוט והגיוני - כמו למשל בבנייה ופירוק של מכונית:

- בבנייה הולכים מהקטן אל הגדול - קודם-כל יוצרים את כל הרכיבים (במקרה שלנו: שתי נקודות), ואז יוצרים את העצם הגדול (במקרה שלנו: קו).
- בפירוק הולכים מהגדול אל הקטן - קודם-כל מפרקים את העצם הגדול (קו) ואז מפרקים את כל הרכיבים (שתי נקודות).

רשימת איתחול - initialization list

כשהקומפיילר בונה את הרכיבים, הוא משתמש ב**בנאי-ברירת-המחדל** שלהם (default constructor) - בנאי בלי פרמטרים - אם יש להם.

בנאי-ברירת-מחדל של מחלקה פשוטה (בלי רכיבים) - לא עושה כלום.

בנאי-ברירת-מחדל של מחלקה מורכבת - קורא לבנאי-ברירת-המחדל של הרכיבים.

אם לרכיבים אין בנאי-ברירת-מחדל - אז כברירת-מחדל, תהיה שגיאת קומפילציה.

שאלה: באיזה מקרה למחלקה אין בנאי-ברירת-מחדל? (התשובה בשיעור הקודם).

מה עושים אם רוצים לקרוא לבנאי אחר במקום בנאי ברירת-המחדל?

--- משתמשים ב**רשימת איתחול**. יש לשים את הרשימה **אחרי** הכותרת של הבנאי אבל **לפני** הסוגריים המסולסלים של קוד הבנאי. ראו דוגמה בתיקיה 0.

שימוש ברשימת-איתחול הוא **מהיר יותר** ו**בטוח יותר** מאיתחול השדות בתוך הבנאי. מדוע?

- מהיר יותר - כי הוא חוסך את האתחול ע"י בנאי ברירת-המחדל.
- בטוח יותר - כי הוא מבטיח שבתוך הבנאי, כל הרכיבים כבר בנויים עם הפרמטרים הנכונים.

רשימת איתחול לא משפיעה על סדר האיתחול של הרכיבים! סדר האיתחול של הרכיבים הוא תמיד לפי הסדר שבו הם מוגדרים במחלקה.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.

סיכם: אראל סגל-הלוי.

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 1.

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 כשרוצים לכתוב פונקציות שמשנות את הארגומנטים שלהם.

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 **l**value כי יש לו **l**ocation וכי אפשר לשים אותו בצד השמאלי (**l**eft) 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 **r**value כי הוא יכול להופיע רק בצד הימני (**r**ight) 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.

במקרה זה, הקומפיילר מאריך את אורך-החיים של המשתנה הזמני, כך שיישאר בזיכרון (בד"כ על המחסנית) למשך כל אורך החיים של הרפרנס.

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` - למשל ערך זמני המגיע מקריאה לפונקציה קודמת.

החזרת רפרנסים מתוך פונקציות

ב-`C++` אפשר לא רק להעביר רפרנסים כפרמטרים לפונקציות, אלא גם להחזיר רפרנסים מתוך פונקציות. ראו הדגמה בתיקיה 2.

למה זה טוב?

1. שיטה של מחלקה יכולה להחזיר רפרנס לשדה של המחלקה, כדי לאפשר לקוראים לשנות את המחלקה. לדוגמה, במחלקה המייצגת מערך של מספרים, יכולה להיות שיטה בשם "`get`" המקבלת אינדקס ומחזירה את המספר הנמצא באינדקס זה. בג'אבה, כדי לשנות את המספר הזה, היינו צריכים שיטה אחרת בשם "`put`". ב-`C++`, אפשר לשנות את השיטה "`get`" כך שתחזיר `int&` במקום `int`, כך נוכל להשתמש באותה שיטה לקריאה ולכתיבה.
2. כשכותבים שיטה שמשנה מחלקה, מקובל להחזיר רפרנס למחלקה עצמה, כדי שיהיה אפשר לקרוא לכמה שיטות-עדכון בשרשרת. ראו דוגמה בתיקיה 2.

מקורות

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

סיכם: אראל סגל-הלוי.

שיטות קבועות – const methods

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.
- זה מאפשר לנו לקבל אזהרות על תופעות-לוואי לא רצויות. למשל, נניח שאנחנו כותבים פונקציה print שמדפיסה את אחד השדות של העצם שלנו. אבל תוך כדי ההדפסה, אנחנו קוראים לפונקציית-ספריה שמשנה את העצם (יש פונקציות כאלו בספריה התקנית! למשל אופרטור סוגריים מרובעים של map עלול לשנות את העצם!). אם נגדיר את השיטה כ-const, הקומפיילר יזהה את הבעיה ויזהיר אותנו.

למה זה קשה? כי כששיטה 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.

תרגיל בית: קחו את התרגילים הקודמים שלכם, הוסיפו להם "const" במקומות הנכונים, וראו מה קורה..

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.

ראו דוגמה בתיקיה 7.

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, או מונה הסופר את מספר הגישות לפונקציה מסויימת. החישוב לא משנה את מצב העצם מבחינה לוגית, אבל הוא צריך לשמור את התוצאה בתוך העצם.

כדי שהקומפיילר יאפשר לנו לעשות זאת, נסמן את ה-`cache` ב-`mutable`.

שדה המסומן ב-`mutable` ניתן לשינוי גם אם העצם הוא `const`.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- למה חשוב שהשיטות יהיו `const` - מפי מפתח בכיר בפייסבוק
  `https://youtu.be/lkgszkPnV8g`

סיכם: אראל סגל-הלוי.

חברים

לפעמים יש פונקציה שהיא קשורה באופן הדוק למחלקה, אבל אי אפשר להגדיר בתוך המחלקה. לדוגמה, אופרטורי קלט ופלט (<< >>) שנלמד בשיעור הבא.

יש פונקציה שאפשר להגדיר בתוך המחלקה אבל נוח יותר להגדיר בחוץ. לדוגמה, אופרטור חיבור (+) אפשר להגדיר בתוך המחלקה:

```
Complex Complex::operator+ (Complex other)
```

אבל יותר טבעי להגדיר אותו מחוץ למחלקה:

```
Complex operator+ (Complex a, Complex b)
```

בדרך-כלל, כשפונקציות מוגדרות מחוץ למחלקה, אין להן גישה לשדות הפרטיים של המחלקה. אבל בשפת C++ אפשר לחרוג מכלל זה ע"י הצהרה שהפונקציה המדוברת היא **חברה** (friend) של המחלקה. למשל, כדי להגדיר את אופרטור החיבור מחוץ למחלקה ולאפשר לו להשתמש בשדות של המחלקה, צריך לכתוב בתוך המחלקה (בקובץ h):

```
friend Complex operator+ (Complex a, Complex b);
```

את המימוש אפשר לכתוב כרגיל בקובץ cpp; המימוש יוכל לגשת לשדות הפרטיים של המחלקה.

ראו דוגמה בתיקיה 8.

באותו אופן, אפשר להגדיר **מחלקה חברה**: מחלקה אחת יכולה להגדיר מחלקה אחרת כ"חברה" ובכך לאפשר לה לגשת לשדות הפרטיים שלה.

זה שימושי כשיש שתי מחלקות הקשורות באופן הדוק זו לזו, למשל: מחלקה של וקטור, ומחלקה של איטרטור על הוקטור הזה.

חידה: האם הקשר הוא הדדי? אם מחלקה A חברה של מחלקה B, האם בהכרח מחלקה B גם חברה של מחלקה A? חישבו קודם מה התשובה ההגיונית, ואז כיתבו תוכנית דוגמה ובידקו אם זה אכן כך!

מקורות

- מצגת של אופיר פלא.

סיכם: אראל סגל-הלוי.

העמסת אופרטורים

העמסה (overloading) היא מצב של כמה פונקציות עם אותו שם וארגומנטים מסוגים שונים.

בשפת ++C, גם אופרטור הוא פונקציה, ולכן אפשר **להעמיס אופרטורים**: להגדיר אופרטורים המבצעים פעולות שונות לפי סוג הארגומנטים המועברים אליהם. לדוגמה:

- **אופרטורים חשבוניים** (חיבור, חיסור, כפל, השוואה, וכו') מוגדרים על מספרים שלמים וממשיים; אנחנו יכולים להעמיס אותם גם במחלקות שאנחנו בונים, המייצגים עצמים מתמטיים מורכבים יותר. למשל: מספרים מרוכבים (ראו תיקיה 1), מטריצות, פולינומים וכו'. בספריה התקנית הם הורחבו גם למחרוזות.
- **אופרטורי זרימה** (<< >>) - מוגדרים במקור (בשפת סי) על מספרים שלמים, אבל בשפת ++C העמיסו אותם לזרמי קלט ופלט כפי שכבר ראינו. גם אנחנו יכולים להרחיב אותם כדי לכתוב ולקרוא מחלקות שאנחנו בונים (ראו תיקיה 1).
- **אופרטור סוגריים מרובעים []** - מוגדר לגבי מערכים בסיסיים; אפשר להרחיב אותו גם למחלקות שאנחנו בונים, כשאנחנו רוצים לגשת לדברים לפי אינדקס (ראו תיקיה 2).
- **אופרטור סוגריים עגולים ()** - יכול לשמש להגדרת אובייקטים המתפקדים כמו פונקציות - "פונקטורים" (functors). ראו תיקיה 4.

כשמעמיסים אופרטורים, חשוב לשים לב שהערך המוחזר תואם למשמעות של האופרטור. למשל:

- אופרטור + מחזיר את הסכום; אופרטור += מגדיל את הארגומנט השמאלי שלו אבל **גם** מחזיר את הסכום (שהוא הארגומנט השמאלי אחרי ההשמה).
- אופרטור = (השמה) מחזיר *this - זה מאפשר לבצע השמות בשרשרת.
- האופרטורים >> << מחזירים את זרם הקלט/פלט שהם מקבלים - שוב כדי לאפשר שרשרת.
- לאופרטור הגדלה באחד (++) והקטנה באחד (--) יש שתי גירסאות: כששמים אותו לפני המספר (prefix), הוא מחזיר את המספר אחרי ההגדלה; כששמים אותו אחרי המספר (postfix), הוא מחזיר את המספר לפני ההגדלה. כדי להבדיל בין האפשרויות, יש להגדיר את הגירסה השניה עם פרמטר מדומה מסוג int, למשל:
  - `T& T::operator++();  // prefix operator`
  - `T& T::operator++(int);  // postfix operator`

אופרטור גרשיים

אופרטור גרשיים נקרא גם אופרטור הסיומת (suffix operator). מגדירים אותו בעזרת גרשיים, אבל משתמשים בו כסיומת - בלי גרשיים. לדוגמה, מגדירים כך:

```
Complex operator"" _i(long double x) {
```

```
    return Complex(0,(double)x);
}
```

ומשתמשים פשוט כך:

```
7.0_i
```

ראו דוגמה בתיקיה 8.

אופרטור פסיק

אופרטור פסיק מוגדר בשפת סי, והמשמעות שלו היא "התעלם מהפרמטר הראשון והחזר את הפרמטר האחרון". זה שימושי כשלפרמטר הראשון יש תוצאת-לוואי, למשל, הפרמטר הראשון מייצר ערך מסויים, והפרמטר השני בודק אותו.

בשפות-תיכנות מודרניות יותר, כגון פייתון, אופרטור פסיק משמש להגדרת `tuple` - זוגות, שלשות וכו'. אנחנו יכולים לקבל את המשמעות הזאת ע"י העמסה - אבל רק למחלקות או ל-`enum` (זה לא עובד עם מספרים).

ראו דוגמה בתיקיה 9.

מקורות

- מצגת של אופיר פלא.

סיכם: אראל סגל-הלוי.

העתקה, השמה והמרה

בנאי מעתיק

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!

ראו הדגמה של בנאי מעתיק בתיקיה 1.

אופרטור השמה

אופרטור השמה (=) מגדיר איך להעתיק עצמים. חשוב במיוחד להגדיר אותו נכון כשהעצמים הם "עמוקים" (כוללים הקצאת זיכרון דינמית). בדרך כלל, במקרה זה נרצה להגדיר שלוש שיטות: בנאי מעתיק (copy constructor), מפרק (destructor), ואופרטור השמה (assignment operator). ראו תיקיה 1.

בנאי מעתיק לעומת אופרטור השמה

העתקת עצמים מתבצעת בארבעה מקרים:

1. כשמגדירים עצם חדש מתוך עצם קיים (למשל Complex a=b או Complex a(b)).
2. כשמעבירים פרמטר לפונקציה by value.
3. כשמחזירים ערך מפונקציה by value.
4. בפעולת השמה של עצם קיים לעצם קיים אחר (למשל Complex a; a=b).

במקרים 1, 2, 3 הקומפיילר קורא לבנאי מעתיק; במקרה 4 הקומפיילר קורא לאופרטור השמה.

שימו לב: כשכותבים Complex a=b נקרא בנאי מעתיק ולא אופרטור השמה, כי נבנה כאן עצם חדש.

אופרטורי המרה

סוג מיוחד של אופרטורים הוא **אופרטורי המרה**. הם הולכים ביחד עם **בנאי המרה**:

- בנאי המרה ממיר ממחלקה אחרת למחלקה שלנו. לדוגמה ראו במחלקה Fraction בתיקיה 5, עבורה הגדרנו בנאי המרה ממספר שלם.
- אופרטור המרה ממיר מהמחלקה שלנו למחלקה אחרת. לדוגמה ראו במחלקה Fraction בתיקיה 5, עבורה הגדרנו אופרטור המרה למספר ממשי.

האם אפשר לשים גם בנאי המרה וגם אופרטור המרה? -- אפשר, אבל זה עלול לבלבל את הקומפיילר, כי במקרים מסוימים יהיו לו שתי דרכים לבצע המרה והוא לא יידע באיזו מהן להשתמש. פתרון אפשרי הוא להשתמש במילה השמורה explicit. מילה זו אומרת לו להפעיל המרה רק כשמבקשים אותה באופן מפורש; ראו דוגמה בתיקיה 5.

המילה explicit נועדה גם למנוע שגיאות לוגיות ולהפוך אותן לשגיאות קומפילציה; ראו דוגמה בתיקיה 4.

מקורות

- מצגת של אופיר פלא.
- https://stackoverflow.com/q/49092801/827927
- איך לכתוב אופרטור השמה עמוק בשתי שורות https://chara.cs.illinois.edu/sites/cgeigle/blog/2014/08/27/copy-and-swap/

סיכם: אראל סגל-הלוי.

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.

הדפסת מספרים ממשיים ב++C

כידוע, מספרים ממשיים עלולים להיות אינסופיים, ואנחנו לא רוצים להדפיס מספרים אינסופיים. לכן כשמדפיסים מספרים ממשיים, המחשב מעגל אותם בהתאם ל**רמת הדיוק** שבחרנו. המשמעות של "רמת הדיוק" (precision) היא מספר הספרות המשמעותיות שמודפסות – מספר הספרות בין הספרה השמאלית ביותר שאינה 0 לספרה הימנית ביותר שאינה 0. לדוגמה:

- במספר 7600 יש שתי ספרות משמעותיות – 7, 6.
- גם במספר 0.0076 יש שתי ספרות משמעותיות – 7, 6.
- במספר 7600.0076 יש שמונה ספרות משמעותיות.

ברירת-המחדל של רמת-הדיוק בהדפסה ל-cout היא **6**. אם מדפיסים מספר עם יותר מ-6 ספרות משמעותיות – נראה רק 6. למשל:

- `cout << 1234.5678;`

מדפיס **1234.57**: במספר המקורי יש 8 ספרות משמעותיות, המחשב חותך 2 ומעגל (במקרה זה מעגל למעלה)

- `cout << 12345678.;`

מדפיס **1.23457e+07**: שוב המחשב חותך שתי ספרות ומעגל, אבל במקום לכתוב 12345700, כדי לחסוך באפסים, הוא כותב רק את 6 הספרות המשמעותיות, ומוסיף את האקספוננט e+07 (שמשמעותו: "כפול 10 בחזקת 7").

אפשר לשנות את רמת-הדיוק ע"י הפקודה setprecision שנמצאת בכותרת <iomanip> - ראו בתיקיה 8. למשל, אם נשנה את רמת הדיוק ל-4, נקבל **1235** ו **1.235e+07**; אבל אם נשנה ל-10, נקבל רק 8 ספרות משמעותיות – **1234.5678** ו **12345678**.

אם רמת-דיוק היא גבוהה מאד, אנחנו עלולים לקבל תוצאות לא צפויות. למשל, ברמת דיוק 100 מקבלים:
1234.5678000000000338332029059529304504394531 25
מדוע? - כי יש אינסוף מספרים ורק מספר סופי של ייצוגים של מספרים בסיביות. לכן, לרוב המספרים אין ייצוג בסיביות, והמחשב מעגל אותם לייצוג הקרוב ביותר. למשל, למספר 1234.5678 אין ייצוג בסיביות, והמספר הקרוב ביותר שיש לו ייצוג בסיביות (לפחות על המחשב שלי) הוא המספר הנ"ל.
כשרמת הדיוק היא נמוכה, אנחנו לא שמים לב לזה, כי בתצוגה המספר מתעגל ל 1234.5678. אבל כשרמת הדיוק גבוהה, אנחנו רואים את המספר כפי שהוא באמת.

מקורות

- 1) Floating-Point Determinism ,
- 2) What Every Computer Scientist Should Know About Floating-Point Arithmetic ,
- 3) Is floating point math broken? .
- https://en.cppreference.com/w/cpp/io/ios_base/precision

סיכם: אראל סגל-הלוי

זרמי קלט ופלט

נלמד על זרמי קלט ופלט (iostreams) בספריה התקנית של ++C. זה גם נושא חשוב ושימושי בפני עצמו, וגם דוגמה מעולה לירושה רגילה וכפולה והעמסת אופרטורים.

היררכיית הזרמים

הספריה התקנית של ++C מגדירה היררכיה שלמה של זרמי קלט ופלט (ראו תרשים במצגת). לצורך הדיון נתמקד במחלקות העיקריות:

- בשורש של עץ הירושה נמצאת המחלקה ios_base והיורשת שלה – ios. המחלקות האלו כוללות משתנים המשותפים לכל הזרמים, כגון: הפורמט ורמת הדיוק של מספרים ממשיים.
- יורשות ממנה המחלקות istream – זרם קלט, ו-ostream – זרם פלט.
- המחלקה iostream מייצגת זרם אשר יכול לשמש גם לקלט וגם לפלט. היא יורשת גם מ istream וגם מ ostream – זו דוגמה לירושה כפולה (אחת הדוגמאות היחידות לירושה כפולה שהיא גם שימושית – בדרך-כלל ירושה כפולה נחשבת לבעייתית ולא שימושית במיוחד).
- מהמחלקה istream יורשות כמה מחלקות שונות של זרמי קלט, במיוחד ifstream (קלט מקובץ) ו istringstream (קלט ממחרוזת). כמו כן, המשתנה cin המשמש לקלט מהמקלדת הוא מסוג זה.
- מהמחלקה ostream יורשות כמה מחלקות שונות של זרמי פלט, במיוחד ofstream (פלט לקובץ) ו ostringstream (פלט למחרוזת). כמו כן, המשתנה cout המשמש לפלט למסך הוא מסוג זה. גם המשתנה cerr משמש לקלט למסך.
  - מה ההבדל? – ההבדל הוא ברמת מערכת ההפעלה: מערכת ההפעלה מקצה לכל תוכנית זרם-קלט אחד (נקרא גם "קלט תקני" או stdin), ושני זרמי-קלט (נקראים גם "פלט תקני" או stdout, ו"פלט-שגיאה תקני" או stderr). כשמריצים תוכנית בלינוקס, ניתן להפנות את הפלט התקני לקובץ אחד ואת פלט-השגיאה התקני לקובץ אחר. למשל, בפקודות הבאות:
    - `./a.out`
    - `./a.out > out.txt`
    - `./a.out 2> err.txt`
    - `./a.out > out.txt 2> out.txt`
- הפקודה הראשונה כותבת גם את הפלט התקני וגם את פלט-השגיאה התקני למסך. הפקודה השניה כותבת את הפלט התקני לקובץ, ואת פלט-השגיאה התקני למסך (שימושי כשיש הרבה פלט שלא רוצים לראות על המסך אלא לשמור בקובץ לעיון מאוחר יותר, אבל עדיין רוצים שהודעות-שגיאה יגיעו למסך באופן מיידי). הפקודה השלישית כותבת את פלט-השגיאה התקני לקובץ ואת הפלט הרגיל למסך, והפקודה הרביעית כותבת את שני סוגי הפלטים לשני קבצים שונים.

- מהמחלקה iostream יורשות כמה מחלקות שונות של זרמים המשמשים גם לקלט וגם לפלט, למשל fstream, stringstream.

דוגמאות לשימוש בזרמים השונים ניתן למצוא בתיקיה 2.

מניפולטורים

אפשר "לכתוב" לזרמים גם עצמים מיוחדים שנקראים "מניפולטורים" – io manipulators. כשכותבים עצם כזה, למשל, ל-cout, לא מודפס שום דבר למסך, אלא המצב של הזרם cout משתנה. למשל, אפשר "להדפיס" לשם עצם שנקרא setprecision, המשנה את רמת-הדיוק בכתיבת מספרים ממשיים (ראו דוגמה בתיקיה 3).

איך זה עובד? פשוט, ע"י העמסת אופרטורים! ראו את קוד המקור כאן:
http://cs.brown.edu/~jwicks/libstdc++/html_user/iomanip-source.html .

קבצים בינריים

עד עכשיו עבדנו עם קבצי טקסט, שבהם כל בית מייצג (בדרך-כלל) אות קריאה ע"י אדם. יש גם קבצים בינריים, המשמשים למשל לייצוג תמונות. ישנן דרכים רבות לייצג תמונה בקובץ. ברמה הבסיסית ביותר, כדי לייצג תמונה צריך לייצג את עוצמת האור בפיקסלים של התמונה. נחשוב לדוגמה על תמונה בגווני אפור. לכל פיקסל יש, נניח, 256 גוונים אפשריים של אפור – מ-0 (שחור) ל-255 (לבן). יכולנו לייצג את התמונה כקובץ טקסט ובו רשימה של מספרים מופרדים בפסיקים, אבל זה מאד בזבזני. מקובל יותר לייצג תמונה כקובץ בינרי ובו כל בית מייצג פיקסל אחד, באופן דחוס.

אם רוצים לייצג תמונה צבעונית, אפשר לייצג כל פיקסל בעזרת שלושה בתים, שכל אחד מהם מייצג את עוצמת האור בערוץ-צבע אחר (למשל: אדום, ירוק, כחול). ישנן דרכים רבות לעשות זאת, והן מעבר להיקף של קורס זה. אנחנו לומדים על הנושא רק כדוגמה לשימוש בקובץ בינארי. בתיקיה 5 ניתן למצוא דוגמה לקובץ-תמונה בפורמט פשוט ביותר – פורמט ppm. שימו לב איך התמונה נוצרת באופן אוטומטי ע"י נוסחה בתוך מערך בזיכרון, ואיך היא נשמרת לקובץ בפעולה אחת write.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.

סיכם: אראל סגל-הלוי.

המרות סוגים ומידע על סוגים בזמן ריצה

בשפת סי, כשרוצים להמיר סוג, משתמשים בסוגריים. יש הרבה סוגים של המרות וכולן מתבצעות באותו אופן. למשל:

```
double d = 3.0;  int i = (int) d;   // המרת מספרים

const int* cp = &i;   int *ncp = (int*)cp;  // המרה עוקפת קונסט

double* dP = (double*)ncp;  // המרת פוינטר מסוג אחד לפוינטר מסוג אחר

Base *baseP1 = new Derived;  Derived *derP1 = (derP*)baseP1; // המרת
פוינטר בסיס שמצביע למחלקה יורשת, לפוינטר מהסוג הנכון.

Base *baseP2 = new Base;  Derived *derP2 = (derP*)baseP2; // המרת פוינטר
בסיס שמצביע למחלקת בסיס, לפוינטר מהסוג הלא-נכון.
```

בשפת ++C, המרות מתבצעות ע"י אופרטורים מיוחדים - לכל סוג של המרה יש אופרטור עם שם משמעותי, שעוזר לקורא להבין איזו המרה בדיוק מתבצעת כאן. אנחנו נציג אותם מהנדיר לנפוץ.

---

const_cast

האופרטור const_cast משמש להמרת מצביע או רפרנס קבוע (const) למצביע/רפרנס רגיל (לא const). הוא למעשה אומר לקומפיילר להתעלם מהבדיקה של const.

אופרטור זה משמש במקרים נדירים מאד. בדרך-כלל, שימוש באופרטור זה מראה על שגיאה בתיכנון המחלקה. צריך לתכנן את המחלקה נכון כך שכל מה שצריך להיות const אכן יהיה const.

מתי בכל-זאת משתמשים בו? כשאנחנו מקבלים קוד ישן של מתכנת שעשה שגיאה ולא סימן פונקציה מסויימת כ-const למרות שבפועל היא כן const (לא משנה את העצם שהיא מקבלת כפרמטר). אם יש לנו גישה לקוד המקור - נשנה אותו ונסמן את הפונקציה כ-const. אבל לפעמים אין לנו גישה לקוד המקור אלא רק לקובץ הבינארי המקומפל. במקרה זה, אנחנו משתמשים בconst_cast כדי להגיד לקומפיילר "סמוך עלינו, בדקנו ואנחנו יודעים שהפונקציה לא משנה את הארגומנט שלה, יהיה בסדר".

כמו במקרים רבים אחרים בחיים, כשמישהו אומר "סמוך עליי" זה עלול להיות פתח לצרות... לכן בדרך-כלל לא נשתמש בהמרה זו.

reinterpret_cast

האופרטור reinterpret_cast משמש להמרת פוינטר או רפרנס מסוג אחד לסוג אחר. האופרטור לא מבצע שום בדיקה - לא בזמן קימפול ולא בזמן ריצה. כמו const_cast, הוא למעשה אומר לקומפיילר להתעלם מבדיקות הטיפוסים הרגילות שהוא מבצע, ולסמוך עלינו שאנחנו יודעים מה אנחנו עושים. זה מסוכן ועלול לגרום לשגיאות לוגיות וערכים לא מוגדרים. ראו תיקיה 4.

אם reinterpret_cast כל-כך מסוכן, למה בכל-זאת משתמשים בו? שימוש לגיטימי הוא כשרוצים לתרגם מחלקה כלשהי לבינארית לצורך כתיבה לקובץ בינארי (למשל קובץ תמונה). ראו דוגמה בתיקיה 5.

static_cast

האופרטור static_cast משמש להמרה המתבצעת ע"י פונקציה ידועה בזמן קומפילציה. לדוגמה:

```
double d = 12.45;
int i = static_cast<int>(d);
```

זו המרה של מספר ממשי למספר שלם, המתבצעת ע"י פעולה ידועה - לקיחת החלק השלם בלבד. באותו אופן אפשר להמיר מספר שלם למספר ממשי.

למה זה עדיף על המרה בעזרת סוגריים כמו בסי -

```
int i = (int)d;
```

? משתי סיבות:

א. קל יותר למצוא המרות בקוד - פשוט מחפשים את המחרוזת static_cast.

ב. ההמרה בעזרת סוגריים עלולה להיות שגויה. למשל, בעזרת סוגריים אפשר להמיר מצביע למספר ממשי למצביע למספר שלם:

```
int* ip = (int*)dp;
```

ההמרה נראית בדיוק כמו קודם אבל היא שגויה - התוצאה תהיה זבל. לעומת זאת, אם נכתוב:

```
int* ip = static_cast<int*>(dp);
```

הקומפיילר יציל אותנו מנפילה לפח הזבל בכך שיכריז על שגיאת קומפילציה - אין המרה סטטית המאפשרת להמיר מצביע לממשי למצביע לשלם (ראו תיקיה 4).

איך זה עובד עם מחלקות?

- static_cast חוסם המרות בין מצביעי-מחלקות שאין ביניהן קשר (אף אחת לא יורשת מהשניה).
- static_cast מאפשר המרה של מצביע למחלקה יורשת אל מצביע למחלקת הבסיס - והמרה כזאת היא בטוחה (אבל היא מתבצעת אוטומטית גם בלי static_cast).
- static_cast מאפשר גם המרה של מצביע למחלקת בסיס אל מצביע למחלקה יורשת - והמרה כזאת היא **מסוכנת** - היא נכונה רק אם המצביע המומר אכן הצביע לעצם מהסוג של המחלקה היורשת. **לא מתבצעת** כל בדיקה לנושא זה ולכן זו סכנה - עדיף במקרה זה להשתמש ב-dynamic_cast שנלמד בהמשך.

שימו לב: כשמשתמשים ב-static_cast, בדרך-כלל הקומפיילר מכניס פקודה כלשהי שתתבצע בזמן ריצה. כשמשתמשים ב-const_cast או reinterpret_cast, הקומפיילר לא מכניס שום פקודה לביצוע בזמן ריצה - ההוראה משפיעה על הקומפילציה בלבד. לא מאמינים? בדקו ב-godbolt.org.

dynamic_cast

האופרטור dynamic_cast משמש להמרה של פוינטר/רפרנס למחלקת-בסיס עם שיטות וירטואליות, אל פוינטר/רפרנס למחלקה יורשת שלה. כשמשתמשים באופרטור זה, הקומפיילר מכניס בדיקה, המתבצעת בזמן ריצה, אם העצם שמצביעים אליו אכן מתאים לסוג שאליו רוצים להמיר.

לדוגמה, אם יש לנו מצביע מסוג Shape לעצם מסוג Circle, אפשר להמיר אותו ע"י המרה דינמית למצביע מסוג Circle. אבל, אם ננסה להמיר אותו למצביע מסוג Square, האופרטור יחזיר null.

אם ננסה להמיר רפרנס במקום מצביע - לא נקבל null (כי אין רפרנס כזה) אלא נקבל חריגה - bad_cast.

הבדיקה מתבצעת לפי סוג העצם בזמן ריצה. לכן, אפשר לבצע בדיקה זו רק אם העצם הוא ממחלקה שיש לה מידע על סוגים בזמן ריצה - כלומר מחלקה שיש לה טבלת שיטות וירטואליות. אם למחלקת הבסיס אין שיטות וירטואליות - הקומפיילר לא ייתן לנו להשתמש ב-dynamic_cast.

אפשר להשתמש ב-dynamic_cast כדי לדמות את אופרטור instanceof. ראו דוגמה בתיקיה 6.

שימו לב: dynamic_cast היא פעולה "יקרה" בזמן ריצה - היא צריכה לעבור על כל עץ הירושה כדי לקבוע אם ההמרה תקינה או לא.

מידע על סוגים בזמן ריצה - RTTI

האופרטור dynamic_cast משתמש במידע על סוג העצם בזמן ריצה.

אנחנו יכולים לגשת למידע הזה ישירות בעצמנו, בעזרת האופרטור typeid.

כשמריצים typeid על עצם, מקבלים מצביע למבנה המכיל מידע על סוג העצם, כגון השם שלו (השיטה name), האינדקס שלו, ועוד.

אם העצם הוא מסוג סטטי - המידע נקבע בזמן קומפילציה. אם העצם הוא מסוג פולימורפי (- צאצא של עצם עם פונקציות וירטואליות), אז הקריאה ל-typeid יוצרת פעולה בזמן ריצה הפונה לטבלה הוירטואלית ושולפת משם את סוג העצם. ראו דוגמה בתיקיה 6.

זמן הריצה - O(1) - לא צריך לעבור על כל עץ הירושה אלא רק ללכת לטבלה הוירטואלית של המחלקה הנוכחית.

שימו לב: אפשר להגיד לקומפיילר שלא ישמור בכלל מידע על סוגים בזמן ריצה, כך שלא יהיה לנו dynamic_cast ולא typeid. בקומפיילר של ויז'ואל סטודיו זו ברירת המחדל - כדי לחסוך זמן ומקום בזיכרון. אם רוצים להשתמש באפשרויות האלו צריך לשנות את ההגדרות של הקומפיילר.

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- http://en.cppreference.com/w/cpp/language/reinterpret_cast
- https://stackoverflow.com/q/103512/827927 - למה להשתמש ב static_cast?
- http://en.cppreference.com/w/cpp/language/typeid

סיכם: אראל סגל-הלוי.

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-".

אופרטורים לעומת ממשקים

שימו לב להבדל בין ++C לבין Java בנושא זה. בשפת Java, אם אנחנו רוצים להגדיר פונקציית סידור כללית, אנחנו צריכים קודם-כל להגדיר ממשק "Comparable" המגדיר את העובדה שהעצם ניתן להשוואה (נניח עם שיטה מופשטת בשם compare), ואז לבנות פונקציה sort המקבלת עצם מסוג Comparable, ולממש אותה בעזרת השיטה compare.

בשפת ++C בדרך-כלל לא עובדים עם ממשקים, כי זה בזבזני – זה דורש שכל עצם יכיל מצביע לטבלה הוירטואלית. כשכותבים תבנית sort, אפשר לקרוא לה עם עצם מכל סוג שהוא, כולל סוג פרימיטיבי כמו int, בתנאי שיש לו אופרטור "קטן מ-". כדי לאפשר את זה, הקומפיילר עובד בשני שלבים:

שלב א – כשהקומפיילר נתקל בקריאה ל-sort עם סוג כלשהו T, הוא מחולל קוד-מקור חדש של פונקציה sort<T>. בשלב זה לא מתבצעת כל בדיקה על הסוג T.
שלב ב – קימפול הקוד החדש שנוצר בשלב א. בשלב זה, אם חסר אופרטור, תתקבל שגיאת קומפילציה. השגיאה תופיע בשני חלקים: יהיה **error** בשורה שגורמת לשגיאה בתוך התבנית, ובנוסף יהיה **note** המפנה למקום שבו קוראים לתבנית.

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.

מקורות

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

סיכם: אראל סגל-הלוי.

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.

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.

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.

מתאמים

מתאם (adaptor) הוא דגם-עיצוב שנועד להתאים מחלקה נתונה לממשק רצוי. בספריה התקנית יש כמה מתאמים, למשל:

- stack - מתאם ההופך כל מיכל סדרתי למחסנית ע"י הוספת שיטות push, pop (השיטה pop שולפת את האיבר הכי חדש בתור).
- queue - מתאם ההופך כל מיכל סדרתי לתור חד-כיווני ע"י הוספת שיטות push, pop (השיטה pop שולפת את האיבר הכי ישן בתור).

אפשר לבנות מחסנית/תור על-בסיס וקטור, deque, רשימה מקושרת, או כל מיכל סדרתי אחר.

מחרוזות

מחרוזת ממומשת כמיכל של תוים (char) בתוספת פונקציות שימושיות למחרוזות, כמו חיפוש תת-מחרוזת, שירשור ועוד.

כדי להפוך ערך כלשהו למחרוזת, אפשר להשתמש בשיטה הגלובלית to_string, או ב-ostringstream, או בסיומת s, למשל "abc" עם סיומת s מציין אובייקט string המכיל "abc".

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.

ספריית התבניות התקנית – איטרטורים

איטרטורים הם מושג מרכזי ב-STL. כל האלגוריתמים (שנלמד עליהם בהמשך) עובדים על איטרטורים. לדוגמה, אלגוריתם של חיפוש לא מבצע חיפוש על וקטור נתון, אלא חיפוש על תחום בין שני איטרטורים נתונים. לכן כדי להבין את הספריה חשוב להבין את סוגי האיטרטורים המוגדרים בה:

- איטרטור טריביאלי - משמש לקריאה בלבד, אי אפשר להזיז אותו.
- איטרטור קלט - משמש לקריאה בלבד, אפשר להזיז אותו קדימה (++).
- איטרטור פלט - משמש לכתיבה בלבד, אפשר להזיז אותו קדימה (++).
- איטרטור קדימה (`forward iterator`) - שילוב של איטרטור קלט ופלט, יכול לשמש לקריאה ולכתיבה.
- איטרטור דו-כיווני (`bidirectional iterator`) - כמו הקודם, רק שהוא יכול גם לזוז אחורה (--).
- איטרטור גישה אקראית (`random access iterator`) - כמו הקודם, רק שאפשר גם לבצע עליו אריתמטיקה כמו עם פוינטרים של סי.

המיכלים השונים מציעים איטרטורים ברמות שונות, למשל:

- **זרמים** מציעים איטרטורי קלט, פלט ואיטרטור "קדימה". למה צריך אותם? ראו דוגמה בתיקיה 1.
- **קבוצה, מפה ורשימה** מציעות איטרטור דו-כיווני.
- **וקטור** מציע איטרטור גישה אקראית.

לכל מיכל יש שיטה `begin` המחזירה איטרטור לתחילת המיכל ושיטה `end` המחזירה איטרטור ל**אחרי** סוף המיכל.

למיכלים המאפשרים הליכה אחורה (כמעט כולם, חוץ מזרמים ו-`forward_list`) יש גם שיטה `rbegin` המחזירה איטרטור לסוף המיכל ושיטה `rend` המחזירה איטרטור ל**לפני** תחילת המיכל (עבור איטרציה בסדר הפוך).

לכל איטרטור יש גם גירסה שהיא `const` - גירסה המאפשרת לקרוא את הפריטים במיכל אבל לא לשנות אותם. אפשר לגשת אליה ע"י `cbegin, cend, crbegin, crend`. איך מגדירים אותם? - ראו דוגמה בתיקיה 2.

יש גם איטרטורי הכנסה. למשל, לוקטור יש איטרטור בשם `back_insert_iterator`, שהוא איטרטור מסוג "פלט" (לכתיבה בלבד), ומשמש להכנסת פרטים חדשים בסוף הוקטור. לקבוצה יש איטרטור בשם `insert_iterator`, המשמש להכנסת פרטים לקבוצה (בהתאם לסדר המוגדר על הקבוצה). ראו דוגמה בתיקיה 1.

הכנסת פרטים בעזרת איטרטורים

כשרוצים להכניס פרט באמצע מיכל כלשהו (לא בסוף או בהתחלה), משתמשים בדרך-כלל באיטרטור שאומר איפה בדיוק להכניס.

- כדי להכניס פריט לפני המקום שהאיטרטור i מצביע עליו - **c.insert(i,x)**
- כדי להכניס פריטים בקטע החצי-פתוח מ-first ל-- last
  c.insert(i,first,last)
- אותו הדבר עם מחיקה - **erase**.
- אותו הדבר עם הכנסה במקום - **emplace**.

הפונקציות **insert, erase, emplace** עובדות בצורה דומה בכל המיכלים התומכים בהם (ראו טבלה ב **http://www.cplusplus.com/reference/stl**), אלא שהן שומרות על השמורה של המיכל. כך למשל, אם מנסים להכניס איבר למיכל מסודר, והאיטרטור שנותנים ל-**insert** מצביע למקום הלא נכון מבחינת הסדר - האיטרטור הזה ישמש רק כרמז (**hint**), החיפוש יתחיל משם אבל בסופו של דבר הפריט יוכנס למקום הנכון לפי הסדר.

תקינות איטרטורים

כשעובדים עם איטרטורים, חשוב לוודא שהם **תקינים**. מתי איטרטור עלול להיות לא-תקין? למשל, כשתוך כדי לולאה, אנחנו מוחקים את האיבר שהאיטרטור מצביע עליו:

- ברשימה, מפה וקבוצה - האיטרטור מכיל פוינטר המצביע למקום לא מאותחל בזיכרון - שגיאה חמורה.
- בוקטור - האיטרטור מצביע לאיבר הבא אחרי האיבר שמחקנו - לא שגיאה כל-כך חמורה, אבל עדיין לא מה שרצינו.

אז מה עושים? החל מ- C++11, השיטה erase מחזירה איטרטור מעודכן ותקין לאחרי המחיקה. צריך פשוט לשים את האיטרטור הזה באיטרטור שלנו. ראו דוגמה בתיקיה 3.

איטרטורים על מפה

כשמשתמשים באיטרטור i על מפה (map), הסוג של *i הוא זוג (pair) של מפתח+ערך. ראו תיקיה 4.

איטרטורים על מיכלים אסוציאטיביים

למיכלים אסוציאטיביים, כמו מפה או קבוצה, יש שיטות מיוחדות שמחזירות איטרטורים:

- find - מקבל מפתח, מחזיר איטרטור לערך המתאים או end() אם הערך לא נמצא.
- lower_bound - מקבל מפתח, מחזיר איטרטור לערך הכי קטן שהוא שווה או גדול מהמפתח.

- upper_bound - מקבל מפתח, מחזיר איטרטור לערך הכי קטן שהוא גדול ממש מהמפתח.

ראו תיקיה 4.

מקורות

- מצגות של אופיר פלא.
- Peter Gottschling, "Discovering Modern C++", chapter 4.
- תיעוד הספריה התקנית: http://www.cplusplus.com/reference/stl.

סיכם: אראל סגל-הלוי.

הספריה התקנית – אלגוריתמים

בספריה התקנית של C++ יש 105 אלגוריתמים, נכון לשנת 2017. אנחנו צריכים להכיר את כולם.

למה? - כי האלגוריתמים פותרים בעיות מאד נפוצות בתיכנות. אם לא נכיר את כולם, אנחנו עלולים לנסות לממש אותם בעצמנו תוך כדי פרוייקט אחר. ואז, אנחנו כנראה נממש אותם בחיפזון, בלי בדיקות, בצורה לא יעילה ועם באגים. בנוסף, האלגוריתמים נכנסו לתקן של השפה, ולכן מתכנתים בכל העולם משתמשים בהם ומבינים אותם, ומצפים שגם מתכנתים חדשים יכירו אותם וישתמשו באותה שפה.

איך אפשר לזכור 105 אלגוריתמים? - זה לא קל, אבל אנחנו נחלק אותם לקבוצות הגיוניות (ע"פ המצגת המעולה של יונתן בוקארה – ראו קישור למטה).

לפני שנתחיל, נדגיש שהאלגוריתמים בספריה התקנית מקבלים כקלט **זוג איטרטורים** ולא מיכל. זאת בניגוד לשפות אחרות כגון ג'אבה. בג'אבה יש אלגוריתם סידור נפרד עבור List, עבור מערך של תוים, מערך של מספרים וכו'...; בספריה התקנית יש אלגוריתם **אחד** לסידור טווח, והטווח נתון ע"י שני איטרטורים – התחלה (begin) ואחרי-הסוף (end). אותו אלגוריתם יכול לסדר גם וקטור, גם מערך על המחסנית, גם deque, וגם כל מבנה אחר שיש לו איטרטור-התחלה ואיטרטור-סוף.

שאילתות – queries

בקבוצת השאילתות נמצאים אלגוריתמים רבים המקבלים כקלט טווח (איטרטור התחלה ואיטרטור סוף), ומחזירים ערך כלשהו על הפרטים הנמצאים בטווח. למשל, האלגוריתם count מחזיר את מספר המופעים של פרט מסויים בטווח, האלגוריתם accumulate מחזיר את סכום הפרטים בטווח, וכו'. ראו תיקיה 10.

אלגוריתמים על קבוצות – set algorithms

בקבוצה זו נמצאים אלגוריתמים המקבלים כקלט שתי קבוצות ומחזירים קבוצה שלישית. כל אחת מקבוצות-הקלט מועברת ע"י שני איטרטורים – התחלה וסוף. תנאי מקדים לאלגוריתמים האלה הוא, שהפרטים בכל אחת מקבוצות-הקלט מסודרים לפי אופרטור "קטן מ-" (<) או לפי פונקטור אחר המועבר כקלט. תנאי זה מתקיים עבור std::set, אבל גם עבור וקטור שסידרנו אותו בעזרת sort (ראו למטה). הפלט מיוצר בעזרת איטרטור נוסף – איטרטור-פלט. למשל, האלגוריתם set_union מקבל חמישה איטרטורים – התחלה וסוף של קבוצה א, התחלה וסוף של קבוצה ב, ואיטרטור-פלט עבור התוצאה. האלגוריתם מחשב את האיחוד של קבוצה א וקבוצה ב (אוסף הפרטים הנמצאים לפחות באחת משתי הקבוצות, ללא כפילויות), ומכניס את התוצאה לאיטרטור-הפלט. ראו תיקיה 11.

פרמוטציות – permutations

בקבוצת הפרמוטציות נמצאים אלגוריתמים המקבלים כקלט טווח (איטרטור התחלה ואיטרטור סוף), ומשנים את סדר הפרטים בסדרה הנמצאת בין האיטרטורים, אך אינם משנים את הפרטים עצמם. למשל, האלגוריתם sort מסדר את הסדרה לפי האופרטור "קטן מ-"', או לפי פונקטור אחר כלשהו המועבר כפרמטר לאלגוריתם. ראו תיקיה 12.

מעבירים - movers

בקבוצה זו נמצאים אלגוריתמים המעתיקים או מעבירים פרטים בין טווח אחד לטווח אחר. הפשוט ביותר ביניהם הוא copy: הוא מקבל טווח קלט (ע"י שני איטרטורים), ואיטרטור פלט, ומעתיק את כל הפרטים מטווח הקלט לאיטרטור הפלט. ראו תיקיה 13.

משני-ערך – value modifiers

בקבוצה זו נמצאים אלגוריתמים המשנים את הערכים בטווח נתון. הפשוט ביותר הוא fill – הוא מקבל טווח (= איטרטור התחלה ואיטרטור סוף), וממלא אותו בערך נתון. ראו תיקיה 14.

מיליות – "Runes"

לרוב האלגוריתמים שראינו עד כה יש כמה גירסאות, שאפשר ליצור ע"י הוספות "מיליות" לשם האלגוריתם. לדוגמה, המילית is מציינת שאילתה: is_sorted = האם הטווח מסודר, is_sorted_until = עד איפה בדיוק הטווח מסודר, וכו'. ראו תיקיה 15.

משני-מבנה – structure changers

בקבוצה זו נמצאים שני אלגוריתמים שמטרתם למחוק פרטים מתוך טווחים. למשל remove מקבל טווח (= שני איטרטורים) וערך, ו"מוחק" את כל הפרטים בטווח השווים לערך הנתון. למה "מוחק" בגרשיים? כי האלגוריתם לא באמת יכול למחוק אותם – הוא הרי לא מקבל רפרנס לאוסף, אלא רק שני איטרטורים! אז מה הוא עושה? - הוא מעביר כל הפרטים שאמורים להישאר בטווח (= שאינם שווים לערך הנמחק) לתחילת הטווח, ואז מחזיר איטרטור (נניח iter) לאחרי-הסוף של הפרטים הנשארים. לאחר מכן, אפשר להשתמש בשיטה אחרת של האוסף (למשל vector::erase) כדי לבצע את המחיקה עצמה. ראו תיקיה 16.

אלגוריתמים נוספים

שני האלגוריתמים האחרונים הם transform – ביצוע טרנספורמציה כלשהי על פרטים בטווח (נתון ע"י שני איטרטורים) ושפיכת התוצאה לטווח אחר (הנתון ע"י איטרטור פלט), ו for_each – ביצוע פעולה כלשהי על כל פרט בטווח (נתון ע"י שני איטרטורים). ראו תיקיה 17.

סוגי איטרטורים

כל אלגוריתם דורש איטרטור ברמה מסויימת (ראו טקסט קודם על איטרטורים).

לדוגמה, האלגוריתם sort עובד על איטרטורים מסוג RandomAccess. לכן אפשר להשתמש בו לסידור וקטור וגם מערך פרימיטיבי.

אבל, אי אפשר להשתמש באלגוריתם sort לסידור list, כי האיטרטור שלה הוא מסוג Bidirectional (לא תומך למשל בפעולת חיסור).

(מה עושים? משתמשים בשיטת sort המיוחדת של list; ראו תיקיה 6).

הודעות שגיאה

אחד הקשיים העיקריים בעבודה עם STL הוא הודעות השגיאה. למשל, אם ננסה להריץ את אלגוריתם sort על list, לא נקבל הודעה פשוטה שאומרת "אי אפשר להריץ sort על list", אלא הודעה ארוכה ומסובכת הנכנסת לפרטי התבניות בספריה התקנית (ראו דוגמה בתיקיה 6).

כדי לפענח את הודעת השגיאה, צריך לחפש את ה-note המפנה לשורה בקוד שלנו, ומשם לנסות להבין מה הבעיה.

ישנן ספריות המנסות לתת הודעות שגיאה משמעותיות יותר, למשל boost, STLFilt - לא ניכנס לזה בקורס הנוכחי.

ספריות נוספות

בנוסף לספריה התקנית, יש ספריות נוספות. המקובלת ביותר היא boost היא "כמעט" תקנית - הרבה מהדברים בספריה התקנית התחילו את דרכם ב-boost, השתכללו והשתפרו, עד שבסוף נכנסו לספריה התקנית. לכן, אם חסר לכם משהו בספריה התקנית - נסו לחפש ב-boost.

מקורות

- Jonathan Boccara, "105 algorithms in less than an hour", CPPCON 2018, https://youtu.be/2olsGf6JIkU
- Sean Parent, "C++ Seasoning", GoingNative 2013, https://www.youtube.com/watch?v=qH6sSOr-yk8,
- Jonathan Boccara, "Is for_each obsolete? (no)" https://www.fluentcpp.com/2018/03/30/is-stdfor_each-obsolete/
- מצגות של אופיר פלא.
- Peter Gottschling, "Discovering Modern C++", chapter 4.
- Marius Bancila, "Modern C++ Programming Cookbook", chapter 5
- תיעוד הספריה התקנית: http://www.cplusplus.com/reference/stl.

סיכם: אראל סגל-הלוי.

פוינטרים חכמים

אחד הגורמים העיקריים לשגיאות ולדליפות-זיכרון בשפת C++ הוא שימוש לא נכון בפוינטרים. גם כשנראה שהשתמשנו בפוינטרים בצורה נכונה, מחקנו עם delete את כל מה שאתחלנו בעזרת new, עלולים להיות לנו באגים נסתרים. לדוגמה, ראו את הקוד בתיקיה 1. יש שם מחלקה בשם "מוסיקאי". ניתן להגיד למוסיקאי לנגן (play), אבל אם יש יותר מדי מוסיקאים שמנגנים בו-זמנית – נזרקת חריגה.

הבעיה היא, שכאשר נזרקת חריגה, המחשב לא מגיע לשורות שמבצעות delete, ויש דליפת זיכרון; ראו בפונקציה playMusic1.

ב-Java פותרים את הבעיה בעזרת המילה השמורה finally: שמים את כל הקוד שעלול לזרוק חריגות בבלוק try, ואת הקוד המשחרר את הזיכרות בבלוק finally מתחתיו. השפה מתחייבת, שהקוד בבלוק finally יתבצע בכל מקרה שבו יוצאים מהבלוק – בין אם באופן רגיל או בחריגה. ראו בפונקציה playMusic2.

ב-++C לא קיימת המילה finally, כי אין בה צורך – יש פתרון טוב יותר. זה דגם-עיצוב הנקרא **RAII** – קיצור של: **Resource Acquisition Is Initialization**. בדגם-עיצוב זה, אנחנו בונים מחלקה שהבנאי שלה אחראי לאיתחול, והמפרק שלה אחראי לשחרור. השפה מתחייבת, שהקוד שנמצא במפרק יתבצע בכל מקרה שבו יוצאים מהבלוק – בין אם באופן רגיל או בחריגה. היתרון של זה על-פני finally הוא שצריך לכתוב את הקוד המשחרר רק פעם אחת, כשכותבים את המחלקה, ולא בכל פעם שמשתמשים בה. כתוצאה מכך הקוד קצר ונקי יותר.

מחלקה האחראית לניהול פוינטר בשיטת RAII נקראת פוינטר חכם. מה צריך להיות במחלקה המייצגת פוינטר חכם? קודם-כל בנאי, מפרק, ואופרטור השמה ("כלל השלושה"):

- **בנאי** המקבל פוינטר רגיל, ושומר אותו בשדה פרטי של המחלקה.
- **מפרק** המוחק את הפוינטר שנשמר בשדה הפרטי (אם הוא לא null).
- **אופרטור=** המוחק את הפוינטר הקיים (אם יש) ושם במקומו פוינטר חדש.

דרושים גם אופרטורים שיאפשרו לנו להשתמש בפוינטר החכם שלנו כמו פוינטר רגיל:

- אופרטור → שמחזיק את הפוינטר השמור;
- אופרטור * שמחזיר רפרנס לעצם שהפוינטר מצביע עליו.

עכשיו אנחנו יכולים להחליף את הפוינטר הרגיל בפוינטר החכם שכתבנו, ולמחוק את הפקודה delete; הפוינטר החכם ידאג לכך שהעצמים יימחקו אוטומטית בין אם יש או אין חריגה. ראו בפונקציה playMusic3.

בתוכנית הפשוטה הזאת הפוינטר החכם שלנו מבצע את תפקידו היטב, אבל התוכנית מורכבת יותר עלולה להיות בו בעיה: אם אנחנו מעתיקים אותו למשתנה אחר מאותו סוג, ושניהם יוצאים מחוץ לתחום, אז המפרק של שניהם יפעל, והפוינטר יימחק פעמיים – נקבל שגיאת זמן ריצה מסוג double free. כבר ראינו את השגיאה הזאת כשדיברנו על העתקה שטחית של וקטורים, ושם הפתרון היה לבצע העתקה עמוקה. אבל כאן אנחנו לא רוצים העתקה עמוקה – כל הרעיון של פוינטר הוא שאפשר להעתיק אותו במהירות בלי להעתיק את כל העצם שהוא מצביע עליו! אז מה עושים? - יש כמה פתרונות.

פתרון א: פוינטר ייחודי –unique pointer

פתרון אחד הוא פשוט לא לאפשר העתקה – להחליט שלכל פוינטר בסיסי יש רק פוינטר חכם יחיד שמנהל אותו. כדי למנוע העתקה, צריך למחוק את הבנאי המעתיק ואת אופרטור ההשמה – זה מה שאנחנו עושים במחלקה AutoPointer.

אבל במקרים מסוימים אנחנו עדיין רוצים לאפשר העברה של הפוינטר הבסיסי ממנהל אחד למנהל אחר. לדוגמה, נניח שיש לנו פונקציה musician_with_a_random_name – פונקציה שתפקידה לייצר מוסיקאי חדש עם שם אקראי (פונקציה כזאת נקראת "פונקציית מפעל" – factory function – עוד דגם-עיצוב שכדאי להכיר). אנחנו רוצים שהפונקציה הזאת תחזיר פוינטר חכם, ושיהיה אפשר לשים את הערך שהיא מחזירה בפוינטר חכם אחר, למשל כך:

```
SmartPointer mp =  musician_with_a_random_name();
```

במקרה זה, פעולת ההשמה אינה פוגעת ביחידות, כי ברגע שהפונקציה הסתיימה, אנחנו כבר לא צריכים את הפוינטר החכם שהיא החזירה – אנחנו צריכים רק את הפוינטר הבסיסי שהוא מנהל, ורוצים להעביר אותו למשתנה החדש mp.

אנחנו למעשה צריכים בנאי-מעתיק חדש, שיבצע את הפעולות הבאות:

- יעתיק את הפוינטר הבסיסי מהמשתנה שבצד ימין (המשתנה המוחזר מהפונקציה) למשתנה שבצד שמאל (המשתנה mp);
- ישים nullptr במשתנה שבצד ימין – כך שכאשר המשתנה הזה יוצא מחוץ לתחום, המפרק שלו לא ימחק את הפוינטר הבסיסי.

בקוד זה נראה כך:

```
ptr = other.ptr;
other.ptr = nullptr;
```

עכשיו צריך להיזהר: אם נשתמש בבנאי-המעתיק החדש על עצם רגיל (lvalue) מסוג SmartPointer, אנחנו עלולים לקלקל אותו. אנחנו רוצים להשתמש בבנאי-המעתיק החדש רק על *rvalue* – רק על משתנה זמני, שאי-אפשר לקלקל (כמו ערך מוחזר מפונקציה). איך אפשר לכתוב בנאי המבחין בין lvalue לבין rvalue?

כדי לפתור את הבעיה הזאת, המציאו סוג חדש של פרמטר שנקרא **rvalue reference**. הוא מסומן כמו רפרנס כפול: &&. הקומפיילר קורא לפונקציה המקבלת rvalue reference, כשהפרמטר הוא rvalue; ראו דוגמאות בתיקיה 0.

הדבר מאפשר לנו להגדיר, עבור כל מחלקה, שני בנאים-מעתיקים שונים:

- אחד מקבל const reference (מסומן &const) ומשמש להעתקה;
- השני מקבל rvalue reference (מסומן &&), ומשמש להעברה (נקרא גם "בנאי מעביר").

במקרה של פוינטר חכם ייחודי, אנחנו צריכים רק את השני. במקרים אחרים, ייתכן שנרצה להשתמש בשניהם. לדוגמה, כשמממשים מחרוזת (string), אם כותבים למשל:

```
string x = "abc";
```

```
string y = "def";
string s = x + y;
```

אז בשורה השלישית, אילו היה לנו רק בנאי מעתיק, המחשב היה עושה עבודה מיותרת: יוצר מחרוזת זמנית ששווה ל abcdef, ואז מעתיק אותה (העתקה עמוקה) לתוך המשתנה s, ומוחק אותה.

במקום זה, הגדירו למחלקה string בנאי מעביר, שאינו מבצע העתקה עמוקה אלא העתקה שטחית בלבד, ובו-זמנית, מציב nullptr בפוינטר של הארגומנט שלו, כדי שלא יימחק ע"י המפרק כשיצא מהתחום.

כמו שיש שני סוגי בנאים, יש גם שני סוגי אופרטורי השמה:

- אחד מקבל const reference (מסומן &const) ומשמש להעתקה;
- השני מקבל rvalue reference (מסומן &&), ומשמש להעברה (נקרא גם "אופרטור העברה").

הקומפיילר בוחר ביניהם לפי סוג הפרמטר המועבר: אם מועבר lavlue אז נבחר האופרטור המעתיק, ואם מועבר rvalue אז נבחר האופרטור המעביר.

ומה אם אנחנו בכל-זאת רוצים להעביר משתנה מסוג lvalue? - אנחנו יכולים לעשות לו cast מסוג lvalue לסוג rvalue reference. זה מתבצע ע"י הפונקציה התקנית std::move; ראו דוגמה בתיקיה 0.

עכשיו אנחנו יכולים להבין את המימוש של UniquePointer בתיקיה 1.

פתרון ב: פוינטר משותף – shared pointer

לפעמים אנחנו דווקא כן רוצים לאפשר שיהיו כמה עותקים של אותו פוינטר. זה אחד היתרונות של פוינטר – הוא קטן, וקל לשתף אותו. אנחנו רוצים לוודא, שהוא יימחק אס ורק אס אין בו צורך יותר – אם ורק אם כל המנהלים שלו יצאו מהתחום.

לשם כך אנחנו צריכים לשמור מונה גישה (reference counter), הסופר כמה פוינטרים חכמים מחזיקים את אותו פוינטר בסיסי.

אבל מונה פשוט מסוג int לא יעבוד, כי המספר פשוט יועתק בכל פעם שנעתיק את הפוינטר החכם. אנחנו צריכים מונה מסוג int*, כדי לוודא שכל הפוינטרים החכמים המשתפים את אותו פוינטר בסיסי, רואים את אותו מספר ומשנים את אותו מספר.

מימוש אפשרי נמצא במחלקה SharedPointer בתיקיה 1.

שימו לב: התוצאה של שימוש ב SharedPointer דומה לאיסוף זבל בשפת Java, אבל המנגנון שונה: SharedPointer מבטיח שכל עצם יימחק מהזיכרון מייד כשאין בו צורך – ולא ע"י תהליך נפרד שרץ בזמן לא ידוע כשהמכונה הוירטואלית מחליטה.

פוינטרים חכמים בספריה התקנית

עכשיו, אחרי שהבננו איך לממש פוינטרים חכמים בעצמנו, אפשר לגלות שאין צורך לממש אותם בעצמנו – הם כבר נמצאים בספריה התקנית. בקובץ הכותרת <memory> יש תבנית בשם unique_ptr המציינת פוינטר יחיד, ותבנית בשם shared_ptr המציינת פוינטר משותף; המימוש והתפקוד שלהם דומה לאלה שתיארנו למעלה.

סוג שלישי של פוינטר חכם בספריה התקנית הוא weak_ptr. הוא דומה ל shared_ptr בכך שהוא מאפשר שיתוף של פוינטרים המצביעים לאותו מקום, אבל שונה ממנו בכך שאינו מגדיל/מקטין את מספר הגישות לאותו פוינטר. למה צריך אותו? - כי לפעמים יש מבנים המצביעים אחד על השני. למשל, עץ בינארי שבו כל בן מצביע לאביו וכל אב מצביע לבניו. אם נשתמש רק ב shared_ptr, אז אף אחד מהצמתים בעץ לא יימחק, כי לכל אחד מהם יש מספר-גישות גדול מאפס. לכן צריך שאחד מהכיוונים יהיה weak_ptr כך שלא ימנע את המחיקה.

יש עוד המון דברים ללמוד על פוינטרים חכמים – זה נושא מורכב ביותר שלא נספיק לכסות בקורס זה. אנחנו מקווים שתלמדו עליו בהרחבה בקורס הבא.

מקורות

- מצגת ודוגמאות קוד של ערן קאופמן
- Why doesn't c++ provide a "finally" construct? / Bjarne Stroustrup.
- What is Move Semantics? / FredOverflow

סיכם: אראל סגל-הלוי.

שילוב C++ עם פייתון

שפת C++ היא שפה מאד יעילה ומהירה, אבל גם לא-כל-כך נוחה למתכנת (כמו שודאי שמתם לב אחרי 5 מטלות..)

לעומת זאת, שפת פייתון (**Python**) היא שפה חדשה, מאד קלה ונוחה לתיכנות (תיכוניסטים לומדים אותה לבד תוך שבוע), אבל מאד לא יעילה – נראה בהמשך השיעור דוגמה לתוכנית שרצה בפייתון פי 40 (ארבעים!) מתוכנית זהה בשפת C++.

במקרים רבים נרצה לשלב את היעילות של C++ עם הנוחות של פייתון. לדוגמה, ביישומים של למידת מכונה, אנחנו רוצים מצד אחד שהחישובים יתבצעו מאד מהר – אחרת הם ייקחו שנים רבות (במיוחד בלמידה עמוקה). מצד שני, אנחנו רוצים מערכת נוחה לתיכנות, כך שגם אנשים מחוץ למדעי המחשב (כגון פרופסורים לסוציולוגיה, ספרות וכד') יוכלו להשתמש בה לצרכיהם.

הפתרון הוא לשלב את C++ עם פייתון. ישנן דרכים רבות לשלב, אנחנו נלמד דרך אחת חדשה ונוחה במיוחד – cppyy.

היכרות עם פייתון

השלב הראשון הוא להתקין על המחשב שלכם את שפת פייתון (גירסה 3 ומעלה), וללמוד לעבוד איתה. יש כמה דוגמאות בסיסיות ביותר בתיקיה 1. שימו לב איך מגדירים פונקציות ומשפטי תנאי, ו איך מייבאים חבילות (import).

היכרות עם cppyy

השלב השני הוא להתקין את cppyy. ניתן להתקין אותה דרך מנהל-החבילות של פייתון, למשל:

```
pip3 install cppyy
```

לפרטים נוספים, וכן אם אתם נתקלים בבעיות בהתקנה, אנא קיראו בתיעוד של cppyy.

אחרי שהתקנתם את החבילה, היכנסו לדוגמאות הקוד בתיקיה 2. הקובץ hello.py הוא תוכנית פייתון שגם מגדירה פונקציה ב-C++ בעזרת הפקודה: cppyy.cppdef. לאחר שהגדרנו פונקציה בשם say_hello, היא הופכת להיות שיטה של העצם הגלובלי cppyy.gbl, ואפשר פשוט לקרוא לה כמו לכל שיטה אחרת בפייתון: cppyy.gbl.say_hello().

דרך נוספת לייבא קוד C++ לתוך פייתון היא הפקודה cppyy.include – ראו בקובץ include.py. התוצאה זהה – הקוד המוכלל (מתוך hello.hpp) הופך להיות שיטה של העצם הגלובלי cppyy.gbl.

פרמוטציות

עכשיו נראה דוגמה שבה המעבר ל-C++ חוסך הרבה מאד זמן. נניח שאנחנו רוצים לפתור את בעיית הסוכן הנוסע – הסוכן שצריך לעבור בכל הערים, כך שהמרחק הכולל שהוא עובר יהיה הקצר ביותר. כידוע לכם, זו בעיה קשה מאד. הפתרון הכי פשוט (והכי לא יעיל) לבעיה, הוא פשוט לעבור על כל

הפרמוטציות של כל הערים, לחשב את המרחק הכולל עבור כל פרמוטציה, ולבחור את הפרמוטציה עם המרחק הכולל הקצר ביותר.

בתור הקדמה לבעיה, נפתור קודם בעיה אחרת – לספור את כל הפרמוטציות (כמובן, אנחנו יודעים שמספר הפרמוטציות על n איברים הוא n!, אבל לצורך הדוגמה נניח שאנחנו רוצים לעבור עליהן אחת אחת ולספור אותן).

בקובץ count_permutations.py יש קוד שעושה את זה בשפת פייתון – מאד פשוט קצר וקריא.

בקובץ count_permutations.cpp יש קוד שעושה את זה בשפת ++C - תוך שימוש באלגוריתמים של הספריה התקנית שלמדנו בשיעור קודם.

בקובץ count_compare.py יש קוד שמייבא את count_permutations.cpp בעזרת cppyy.include, ומריץ את שתי הפונקציות – זאת שכתובה בפייתון וזאת שכתובה ב ++C – על אותו קלט.

התוצאות מדהימות – הקוד ב ++C רץ פי 40 יותר מהר, ואפשר לקרוא לו מתוך פייתון באותה רמת נוחות כמו לקוד פייתון מקורי.

תופעה דומה קורה בבעיית הסוכן-הנוסע עצמה – ראו בקבצים traveling_salesman.py, traveling_compare.py, traveling_salesman.cpp. שימו לב לאופן שבו אנחנו מעבירים קלט ל-++C: אנחנו מעבירים וקטור של וקטורים בפייתון, והוא מתורגם אוטומטית לוקטור של וקטורים ב-++C.

השגנו את המטרה – שילבנו נוחות עם מהירות.

סיכום

בדרך-כלל, כשמתחילים לכתוב קוד לצורך הדגמה או בדיקת רעיון חדש, הכי נוח לכתוב אותו בפייתון – התיכנות מאד קל ומהיר. בהמשך, כשהמערכת גדלה והופכת להיות "מבצעית", בודקים את המקומות הקריטיים שבהם מתבזבז זמן רב במיוחד, מתרגמים אותם לשפת ++C, ומשלבים בין השפות בעזרת cppyy או כלי אחר כלשהו.

היתרון שלכם כבוגרי מדעי-המחשב (לעומת "סתם" קורס תיכנות) הוא, שאתם מכירים הרבה שפות והרבה טכנולוגיות שונות, יודעים מה היתרונות והחסרונות של כל שפה, יודעים לבחור את השפה המתאימה למשימה המתאימה, ויודעים גם לשלב בין שפות לפי הצורך.

מקורות

- התיעוד של cppyy: https://cppyy.readthedocs.io/en/latest/

סיכם: אראל סגל-הלוי.