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.