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 ג'אבה

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.

שימוש בסיסי במערכת יוניקס

אחת המטרות של הקורס הזה היא לאמן אתכם בשימוש במערכת יוניקס - מערכת נפוצה ביותר בעולם התיכנות. אנחנו לא נלמד את הנושא לעומק, נלמד רק כמה פקודות בסיסיות שיאפשרו לכם לבנות ולהריץ תוכנות ב++C.

לצורך הקורס נשתמש בלינוקס, ובפרט במערכת אובונטו. המערכת אמורה להיות מותקנת על המחשבים במעבדות, אתם יכולים גם להתקין אותה על המחשב הביתי שלכם במקביל ל"חלונות" (יש מדריכים באינטרנט איך לעשות את זה). אחרי שפתחתם את אובונטו, פיתחו חלון-מסוף (terminal) ונסו את הפקודות הבאות:

- ls - הצגת הקבצים בתיקיה הנוכחית.
- ls -latr - אותו דבר, עם יותר פירוט לגבי כל קובץ, וסידור הקבצים לפי זמן העידכון האחרון שלהם (לרוב הפקודות ביוניקס אפשר להוסיף פרמטרים ע"י סימן מינוס ואחריו אות אחת או יותר).
- cd [dirname] - הליכה לתיקיה אחרת.

- mkdir [dirname] - יצירת תיקיה חדשה
- cat [filename] - הדפסת תוכן הקובץ.
- nano [filename] - עריכת הקובץ.
- grep [string] - חיפוש מחרוזת בקלט התקני (למה זה טוב? נראה בהמשך).
- source [filename] - הרצת פקודות מתוך קובץ.

במערכות יוניקס יש מספר קטן של פקודות בסיסיות, שאפשר לשלב ביניהן על-ידי אופרטורי הכוונת קלט ופלט. למשל:

- העברת קובץ כקלט לפקודה - אופרטור הכוונת הקלט (input redirection):
- grep [string] < input.txt
- שמירת התוצאות של פקודה לתוך קובץ - אופרטור הכוונת הפלט (output redirection):
- ls -latr > output.txt
- העברת הפלט של פקודה אחת לפקודה אחרת - אופרטור הצינור (pipe):
- ls -latr | grep bash

ניתן לראות דוגמה בסקריפט showmemory.sh בתיקיה 2, שהשתמשתי בו לצורך הדפסת הזיכרון הפנוי הנוכחי.

התקנת חבילות

במערכות לינוקס מודרניות ישנן פקודות פשוטות המאפשרות להתקין חבילות חדשות. הפקודות משתנות מהפצה להפצה. בלינוקס, הפקודה המשמשת להתקנה היא apt. כדי להשתמש בפקודה הזאת, צריך לקבל "הרשאות משתמש-על" (super-user). איך עושים את זה? שתי דרכים:

- כדי להפוך ל"משתמש על", כותבים את הפקודה su ומכניסים סיסמה. כדי לצאת ממצב "משתמש על", כותבים exit.
- כדי להריץ פקודה אחת כ"משתמש על", כותבים לפני הפקודה sudo. זו השיטה המקובלת.

הנה דוגמה לפקודה שמתקינה משחק נחמד ושימושי לשיפור מהירות ההקלדה בשפות שונות:

```
sudo apt install typespeed
```

(למי שאוהב משחקי טקסט: ראו משחקים נוספים כאן https://askubuntu.com/q/11485/42073).

בניית תוכנת C++ על לינוקס

ישנם כמה חבילות-קומפילציה מקובלות לשפת C++. החבילה הנפוצה ביותר היא כנראה gcc - gnu compiler collection, חבילה הכוללת כמה קומפיילרים שהעיקרי בהם הוא של c, c++. אנחנו נשתמש בחבילה אחרת בשם clang, ובפרט בגירסה 5. היתרון העיקרי של חבילה זו הוא שהודעות-השגיאה שלה קלות יותר להבנה - יתרון חשוב מאד כשלומדים שפה חדשה. כדי להתקין אותה על אובונטו, נכתוב:

```
sudo apt install clang++-5.0
```

עכשיו אנחנו יכולים לבנות תוכנות ב-C++. למשל, בתיקיה 1, נכתוב:

```
clang++-5.0 hello.cpp
```

```
ls -latr
```

ונראה קובץ חדש בשם a.out. זה קובץ שאפשר להריץ. כדי להריץ אותו נכתוב:

```
./a.out
```

בניית תוכנה ב++C (כמו בסי) מורכבת מארבעה שלבים, כדאי להכיר אותם כדי להבין טוב יותר את הודעות השגיאה שאנחנו מקבלים:

- עיבוד מקדים - preprocess - ביצוע פקודות המתחילות ב-#, כגון הכללת קבצים (#include), הגדרת קבועים (#define) וכו'. ראו דוגמה בתיקיה 3.
- הפיכה לאסמבלר - assemble.
- הפיכה לשפת מכונה - compile.
- קישור כל הקבצים בשפת מכונה עם ספריות חיצוניות (במקרה הצורך) והפיכה לקובץ ריצה - link.

ניתן לבצע כל אחד בנפרד ע"פ העברת הפרמטר המתאים ל-clang, אבל בדרך-כלל מריצים את כולם יחד.

אנחנו נצטרך חבילות נוספות לבניית פרוייקטים ובקרת גירסאות:

```
sudo apt install build-essential
sudo apt install git
```

++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).

שינויים נוספים:

- הכללת קבצי-כותרת סטנדרטיים - בלי סיומת h.
- מרחבי-שם - namespaces. למשל, דברים שנמצאים בספריה הסטנדרטית של ++c נמצאים במרחב-שם std וניתן לגשת אליהם, למשל, ע"י std::cout. המטרה של שינוי זה - למנוע מצב ששם מהתוכנה שאנחנו כותבים יסתיר שם מהספריה הסטנדרטית. ניתן לעקוף את זה ע"י הפקודה: using namespace std.

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 מינוס 3 מומר למספר חיובי גדול - ממש לא מה שהתכוונו לעשות.

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

פונקציות inline

בשפת C מקובל להשתמש בקטעי מאקרו כדי לחסוך קוד, למשל:

```
#define SQUARE(x) x*x
```

אבל המנהג הזה מועד לטעויות:

- אין בדיקת טיפוסים. למשל, מה יקרה אם ננסה לעשות `SQUARE("abc")`? תהיה כנראה שגיאת קימפול לא ברורה.

- עלולות להיות תופעות לוואי. למשל, מה יקרה אם נעשה SQUARE(i++)? הערך של i יגדל פעמיים.
- עלולות להיות שגיאות לוגיות. למשל, מה יקרה אם נעשה SQUARE(1+2)? רמז: זה לא יצא 9.

בשפת C++ עדיין אפשר להגדיר קטעי מאקרו, אבל זה מאד מאד לא מומלץ. יש דרך טובה יותר להשיג את אותה מטרה - inline. למשל במקום הקטע הנ"ל אפשר לכתוב:

```
inline int square(int x) {return x*x;}
```

המילה inline נותנת לקומפיילר רמז, שבמקום ליצור פונקציה בשם square, שישים את הקוד שלה בגוף התוכנית. כך שמקבלים אותו אפקט כמו מאקרו.

האם הקומפיילר באמת עושה את זה? לא תמיד.

* אם הפונקציה מסובכת מדי - הוא ישאיר אותה כמו שהיא.

* אם הקומפיילר צריך את הכתובת של הפונקציה - הוא ישאיר אותה כמו שהיא.

* יש חשיבות גם לדגלי האופטימיזציה שמעבירים לקומפיילר. אם לא מעבירים שום דגלים, המילה inline חסרת משמעות - הקומפיילר בכל מקרה יוצר פונקציה מלאה, ראו בקוד האסמבלי שנוצר. זה שימושי בזמן ניפוי שגיאות. אם מעבירים דגל O2 (אופטימיזציה ברמה 2), אז המילה inline משפיעה באופן כלשהו - בלעדיה הקומפיילר יוצר פונקציה בשם squarei, ואיתה הוא לא יוצר פונקציה כזאת. אבל, עם inline או בלי, הקומפיילר מזהה לבד שאפשר להכניס את הקוד x*x לגוף התוכנית!

מסקנה: השימוש ב-inline לא משפיע במיוחד, יש להשתמש בו רק בקטעים שבהם כל ננו-שניה היא קריטית (וגם אז לא בטוח שתהיה השפעה).

מקורות

- מצגות של אופיר פלא ומירי בן-ניסן.
- על ההבדלים בין c++ לבין java, ראו
https://cseducators.stackexchange.com/a/4189/1873
- על inline ומה הוא באמת עושה, ראו
https://stackoverflow.com/q/48751426/827927

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