# C++ - תחביר בסיסי

## הקדמה

שפת C++‎ הומצאה כהרחבה לשפת C, באמצעות הוספת מחלקות (כמו string ו-bool), תכנות מונחה עצמים, תכנות גנרי, בדיקת שגיאות וחריגות, העמסת פונקציות ואופרטורים, מנגנון לניהול זיכרון, ועוד מספר תכונות ההופכות אותה לשפת High Level. כמעט כל תוכנית C תוכל לעבור הידור במהדר C++‎, לכן ניתן להשתמש ולשלב קטעי קוד אשר נכתבו ב-C לקוד אשר נכתב בC++‎.

C++ מצד אחד נחשבת לשפת תוכנה ברמת נמוכה מכיוון שצריך להתעסק עם הקצאה ושחרור של הזיכרון, אמנם מצד שני היא נחשבת לשפת תוכנה ברמה גבוהה מכיוון שמותאמת לתכנות מונחה עצמים. לשילוב זה יש יתרונות גדולים שנפרט בהמשך. שפה זו מתעדכנת כל שלוש שנים.

### יתרונות

* **הבנה** - יש המון מערכות הפעלה שכתובות ב-C++, כמו חלונות, לינוקס, מק ואנדרואיד. וכן מערכות תוכנה מודרניות כמו פייסבוק, גוגל וביטקוין כתובות ב-C++ ביחד עם שפות אחרות. כדי להבין מערכות אלו צריך ללמוד C++.

בנוסף, מכיוון ששפה זו מורכבת ושורשית מאוד, מי שילמד שפה זו טוב יוכל להבין כיצד עובדות שפות תכנות באופן כללי, וככה ילמד אותן טוב ומהר יותר.

* **זיכרון** - בשפת ++C אנחנו יכולים לכתוב תוכנות עם צריכת זיכרון הדוקה וחסכונית יותר. הסיסמה של ++C היא "לא השתמשת - לא שילמת". לכן היא מתאימה במיוחד למערכות שבהן כל טיפת זיכרון חשובה, למשל מערכות מוטמעות (embedded).
* **זמן** - בשפת ++C אפשר לכתוב תוכנות זמן אמת (real-time), כלומר, תוכנות שמגיבות מהר לאירועים.

### חסרונות

* **מורכבות** - שפת C++ היא אחת משפות התכנות 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.
* **פינוי זיכרון** - ב-C++ אין איסוף זבל אוטומטי, לכן אנחנו צריכים להיות אחראים לאסוף את הזבל של עצמנו. אנחנו צריכים לוודא שכל עצם שאנחנו יוצרים בזיכרון, משחרר את כל הזיכרון שהוא תופס. בהמשך נלמד איך לעשות את זה.

## מרחבי-שם - namespace

כאשר מייבאים שתי ספריות שונות אל פרויקט, יכול להיווצר בעיה אם בשתי הספריות האלו יש משתנה, פונקציה או מחלקה מאותו שם, כך שאם נשתמש באחד מאלו נקבל שגיאת ambiguous (רב משמעות). הפתרון הוא לתת לכל משתנה מידע נוסף לזיהוי, כך שגם אם יש להם אותו שם יהיה ניתן להבדיל ביניהם. כמו שני אנשים עם אותו "שם פרטי" אך עם "שם משפחה" שונה.

זה בדיוק מה שעושה מנגנון מרחבי-השם בC++. כל ספרייה שנכתוב שנרצה לייבא אותה לתוכניות אחרות נגדיר אותה תחת מרחב-שם ספציפי, כך שכל פונקציה שניצור בה תהיה שייכת למרחב-שם זה. באמצעות מנגנון זה נוכל בשני מרחבי-שם שונים להגדיר פונקציות עם אותו שם, כך שאם נייבא אותם לתוכנית לא תהיה שגיאת קומפילציה מפני שהקומפיילר ידע להבדיל ביניהם. מגדירים מרחב-שם באמצעות המילה השמורה namespace, ולאחריה שם מרחב-השם וסוגריים מסולסלות. לדוגמא:

namespace abc{

    int x = 123;

    void printx() { std::cout << x << std::endl; }

};

namespace def{

    int x = 456;

    void printx() { std::cout << x << std::endl; }

};

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

אחד הכללים החשובים בבניית ספריות מורכבות הוא "לא לזהם את מרחב השם הגלובלי". כלומר, להשתמש רק במרחבי שם מיוחדים כדי שלא תהיה התנגשות עם פונקציות של ספריות אחרות.

### גישה למשתנים במרחבי-שם

כשנמצאים בתוך מרחב-שם מסוים, אפשר לגשת לכל המשתנים והפונקציות שלו כרגיל בלי להזכיר את "שם המשפחה". אבל כשנמצאים מחוץ למרחב-השם (למשל ב-main), ורוצים לגשת לפונקציה שנמצאת במרחב-שם מסוים, צריך לשים לפניה את שמו של מרחב-השם עם פעמיים נקודתיים "::". אם נשתמש בפונקציה שלא מוגדרת במרחב שם נקבל שגיאת undeclared.

int main() {

    abc::printx(); // will print 123

אם יש שני פונקציות זהות, אחת בתוך מרחב שם והשנייה במרחב הגלובלי, כלומר הוגדרה בקובץ שאני נמצא בו, אזי נוכל לפנות אל הפונקציה במרחב הגלובלי באמצעות '::' ללא מרחב שם לפניה. הקומפיילר ידע שזהו המרחב שם הגלובלי.

::printx();

### using namespace

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

using namespace abc;

int main() {

printx(); // will print 123

מרחב-השם השימושי ביותר בשפת ++C הוא std. הספרייה התקנית של ++C נמצאת כולה במרחב-שם זה. לכן נוכל להשתמש ב- using namespace stdבתחילת תוכנית main כדי שלא נצטרך כל פעם שנרצה להשתמש בפונקציה מהספרייה התקנית לקרוא לה דרך מרחב השם. אמנם לא מומלץ להשתמש בפקודה זו בקובץ-כותרת (hpp) שאנחנו מתכוונים להכליל בקבצים אחרים, כי אז כל המשתנים והפונקציות של הספרייה התקנית יעברו למרחב-השם הראשי, ועלולים להתערבב עם המשתנים והפונקציות שמוגדרים בקובץ עצמו.

אם נרשום using באותו הקובץ על שני מרחבי שמות שונים שבשניהם יש פונקציה מאותו שם, כאשר נקרא לה נקבל שגיאת ambiguous (רב משמעות).

שימוש נוסף במילה השמורה using, הוא כאשר איננו רוצים להשתמש בכל מרחב-השם אלא רק במספר פונקציות מעטות ממנו. לדוגמא, אם איננו רוצים לכלול את כל מרחב-השם std, אך משתמשים הרבה בפונקציה cout כך שאיננו רוצים לקרוא לה כל פעם דרך מרחב-השם, נוכל לרשום מעל פונקציית main:

using std::cout;

### מרחבי-שם מקוננים

ניתן לכתוב מרחבי-שם מקוננים באופן הבא:

namespace name1 {

namespace name2 {

כדי להשתמש בדרך מקוצרת במרחב השם השני נשתמש בפקודה:

using namespace name1::name2;

האופרטור >> משמעו תוזיז את ספרות ימינה, לדוגמא 1<<5 משמעו תוזיז את 1 חמש ספרות ימינה, כל הזזה היא הכפלה ב-2.

שני קבצים מיוחדים בלינוקס

Dev/null – רק פלט משמעותי. צריך בדיקה.

Zero/null – קובץ עם פלט אינסופי של אפסים.