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.