גירסה 1.00 – 24.12.2004

קוד יעיל בשפת C++

ניר אדר

מסמך זה הורד מהאתר http://underwar.livedns.co.il.
אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.
מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כל הזכויות שמורות ל**ניר אדר**

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

מסמך זה יציג בקצרה שיטות לכתיבת קוד יעיל יותר בשפת C++.
החשיבות של אופטימיזציה של הקוד עצומה. רק לדוגמא, עבור תוכנית ממוצעת, רק השימוש ב- inline functions יכול להגדיל את מהירות התוכנית פי 25 (!) - ללא שום שינוי אחר בקוד.

המסמך מציג דוגמאות ומראה איך נתכנת בצורה יעילה, ומה עלול לפגוע בביצועי התוכנית.

שימוש ב-inline functions

פונקציות קצרות, המכילות מספר שורות, נגדיר כ-inline. מילה זו אומרת למהדר להחליף בקוד כל קריאה לפונקציה בקוד שלה, ואז אנחנו חוסכים את זמן הקריאה לפונקציה ואת זמן החזרה ממנה.

דוגמא:

```
inline void func()
{
      // code here
}
```

כאמור, שימוש נכון ב-inline-functions הוא הדרך כמעט הכי יעילה לשפר את ביצועי התוכנית שלנו.

הערה: פונקציות אותן אנו מממשים בתוך ההצהרה על המחלקה הינן באופן אוטומטי inline.

נשים לב שפונקציות שהן יותר משורות בודדות לא נרצה לכתוב כ-inline מכיוון שגם הגדלת הקוד בסופו של דבר תביא לפגיעה בביצועים.

השיפור העצום בביצועים הנגרם על ידי inline נובע עקב העובדה כי בשפת C++ כל גישה למבני הנתונים צריכה להתבצע דרך getters ו-setters, פונקציות הנותנות גישה את השדות של המחלקה. פונקציות אלו הן לרוב בנות שורה אחת שאינה לוקחת הרבה זמן, ואז ה-overhead של הקריאה אל הפונקציה והחזרה ממנה – הוא רוב זמן הקריאה לפונקציה.

נשאל לגבי טיפ זה, ולגבי האחרים - עד כמה שווה לנו לחסוך את המספר הקטן של פקודות המכונה שאנחנו חוסכים על ידי השימוש בו.
התשובה: מאוד שווה. נניח כי הפונקציה הנ"ל היא בקטע הקריטי של התוכנית שלנו – אז הפונקציה הזו תרוץ מליונים ואולי מיליארדים של פעמים. כל פקודה שנחסוך זה חסכון של מיליארדים של פקודות בשפת מכונה. שיפור של מספר קטן של פונקציות בקטע הקריטי של התוכנית – מזניק את ביצועיה.

שימוש ברשימת אתחול ב-constructor

אתחול משתנים צריך לעשות כמה שיותר בתוך רשימת האתחול של ה-Constructor ולא בגוף ה-Constructor עצמו. הקוד הבא:

```
MyClass::MyClass(const CData &data) : m_Data(data) { }
```

עדיף על כתיבה כזו:

```
MyClass::MyClass(const CData &data)
{
	m_Data = data;
}
```

הסיבה: משתנים שאינם מאותחלים ברשימת האתחול מאותחלים באופן דיפולטי לפני הכניסה אל גוף ה-constructor. במקרה השני בעצם בוצע אתחול כפול – פעם לפני הכניסה לגוף ה-constructor ופעם בתוך ה-constructor על ידי הקוד. משתנים המאותחלים ברשימת האתחול מאותחלים פעם אחת בלבד.

אתחול מיד עם הגדרת המשתנה

באופן דומה לטיפ הקודם, נעדיף לאתחל משתנה מיד עם יצירתו, ולא ליצור משתנה ולאחר מכן לשים בתוכו ערך.

נעדיף את הכתיבה:

```
MyClass myClass = data;
```

על פני:

```
MyClass myClass;
myClass = data;
```

במקרה הראשון יקרא ה-copy constructor. במקרה השני ייקרא constructor ללא פרמטרים, ולאחריו ייקרא operator=.

העברת משתני התייחסות בכל מקום בו ניתן

נעדיף להעביר לפונקציות משתני התייחסות בכל מקום שניתן, ולא להעביר משתנים by value.
לדוגמא, נעדיף לכתוב:

```
void myFunc(CData &myVar)
{
      ...
}
```

על פני:

```
void myFunc(CData myVar)
{
      ...
}
```

הסיבה: במקרה השני הפרמטר עובר by value, כלומר בתחילת הפונקציה נקרא copy constructor על מנת ליצור העתק לוקלי של המשתנה עבור הפונקציה, ובסוף הפונקציה נקרא ה-destructor שלו.
זאת לעומת המקרה הראשון, שאף אחד משניהם לא נקרא.

הצהרה על המשתנים ברגע האחרון

שפת C++ מאפשרת לנו להגדיר משתנים לא רק בראש כל בלוק, כמו שפת C, אלא בכל קטע בקוד שלנו.
ננצל עובדה זו כדי להגדיר משתנים מאוחר ככל האפשר.

ראשית – דרך זו עוזרת בקיום הטיפ הקודם – אתחול כל משתנה – נגדיר את המשתנים רק כאשר אנחנו יודעים במה לאתחל אותם.

שנית, אם נגדיר משתנים מאוחר ככל שניתן, יתכן בכלל שנחסוך את היצירה שלהם, אם בחלק מהמסלולים של הפונקציה נצא לפני קטע הקוד בו הם נוצרו.

סידור מבני Switch ו-If

כאשר התוכנית רצה, היא עוברת על מבני ה-if וה-switch אחד אחרי השני, לפי הסדר. לפיכך אסטרטגיה טובה הינה למקם ראשונים את האירועים שקוראים לרוב, על מנת לחסוך בדיקות מיותרות ברוב המקרים.

שמירת ערכי פונקציה יקרה בתוך משתנים

לעתים רבות קוראים בטעות לפונקציה יקרה בתוך לולאה שוב ושוב, למרות שערכה אינו משתנה, נרצה להימנע ממקרה כזה. נעדיף את הקוד הבא:

```
int len = strlen(myStr);

for (int i = 0; i < len; i++)
{
	// code here
}
```

על פני:

```
for (int i = 0; i < strlen(myStr); i++)
{
	// code here
}
```

אופרטור קידום עצמי

נעדיף את האופרטור pre-increment על האופרטור post-increment במידת האפשר.
לדוגמא, במקום הקוד הבא:

```
for (int myVar = 1; myVar < size; myVar++)
{
	// Code here
}
```

נעדיף:

```
for (int myVar = 1; myVar < size; ++myVar)
{
	// Code here
}
```

הסיבה – בשימוש באופרטור post-increment נוצר העתק של האיבר המוחזר לקורא (כי הרי השינוי צריך לקרוא רק אחרי שהחישוב על האובייקט מבוצע). פעולת יצירה זו לוקחת זמן נוסף.

EOF