גירסה 1.00 – 7.1.2005

Smart Pointers

ניר אדר

Nir Adar

Email: underwar@hotmail.com

Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

Smart Pointers

מבוא

מסמך זה עוסק ב-Smart Pointers בשפת C++. **Smart Pointers** הם אובייקטים המתנהגים כמו מצביעים רגילים, אולם מציעים פונקציונליות נוספת.

הבעיה העיקרית עם שימוש במצביעים בשפת C++ היא שחרור זיכרון. פעמים רבות מתכנתים שוכחים לשחרר זיכרון, או מנסים לשחרר שוב זיכרון שכבר שוחרר. Smart Pointer באים לעזור לנו בנושא זה.

Smart Pointers הם שם כללי לאובייקטים החושפים למשתמש בהם התנהגות הדומה לזו של מצביעים. שפת C++ כוללת מימוש של smart pointers בספריה הסטנדרטית שלה, בשם auto_ptr, ובו יעסוק בעיקר מסמך זה. על מנת להשתמש במימוש זה, נכלול את הספריה <memory>.

```
#include <memory>
using std::auto_ptr;
```

כיצד משתמשים ב-smart pointers? נביט בקוד הבא:

```
void foo()
{
	MyClass* p(new MyClass);
	p->DoSomething();
	delete p;
}
```

אם נכתוב קוד זה שוב, תוך שימוש ב-smart pointers, הוא ייראה כך:

```
void foo()
{
	auto_ptr<MyClass> p(new MyClass);
	p->DoSomething();
}
```

נשים לב: בסוף הקוד הנ"ל האובייקט p ידאג אוטומטית לשחרור הזכרון שהוקצה.

דוגמא לשימוש

הקוד הבא מציג שימוש פשוט ב-smart pointers:

```
#include <iostream>
#include <memory>

using namespace std;

int main()
{
	auto_ptr<int> p;
	int i = 7;
	p.reset(&i);
	cout << "The pointer points to: " << *p << endl;
	*p = 10;
	cout << "Contents of i: " << i << endl;
	return 0;
}
```

אנו יוצרים smart pointer המשמש כמצביע ל-int, ובתחילה לא מאתחלים אותו (במקרה זה, הוא יאותחל אוטומטית להצביע אל NULL).

על ידי שימוש בפונקציה reset() אנחנו גורמים למצביע להצביע אל i. (הערה: אם p היה מצביע קודם לכן אל זיכרון אחר, הוא ישחרר אותו לפני שיעבור להצביע אל i).

לאחר מכן אנחנו מבצעים מניפולציות על ה-smart pointer. כפי שניתן לראות, הוא מתנהג תחבירית בצורה זהה לזו של מצביע רגיל.

בסוף התוכנית, כאמור, משוחרר הזיכרון באופן אוטומטי.

מדוע נרצה להשתמש ב-smart pointers?

1. מניעת באגים

שימוש ב-smart pointers מונע באגים שונים:

שחרור אוטומטי של זיכרון: כמו שראינו בדוגמא, הזכרון עבור המצביע משתחרר אוטומטית בסוף הבלוק, ולכן איננו צריכים לדאוג לגבי שחרור הזכרון.

אתחול אוטומטי של המצביע: כאשר אנחנו יוצרים smart pointer שאינו מצביע לכלום, ה- default constructor שלו דואג לאתחל אותו ל-NULL. שוב, אנחנו חוסכים כאן פעולה שהיינו עלולים לשכוח.

Dangling pointers: Dangling pointers זהו מצב בו מצביע מכיל כתובת של אובייקט שכבר שוחרר, לדוגמא:

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
p->DoSomething();       // p is now dangling! This line is a bug
p = NULL;               // p is no longer dangling
q->DoSomething();       // q is still dangling! ☹
```

auto_ptr מציע התנהגות הפותרת את הבעיה. במידה ונשים תוכן של smart pointer אחד בתוך smart pointer אחר, הראשון יעבור להצביע אל NULL, וכך smart pointers דואגים שיהיה בו זמנית רק מצביע אחד אל אותו הזיכרון.

2. Exception Safety

נביט בקטע הקוד הבא:

```
void func()
{
    Element *elem = new Element;
    elem->DoSomething();
    delete elem;
}
```

Element היא מחלקה שהוגדרה על ידינו קודם.

נשים לב: אם במהלך הפונקציה elem->DoSomething(), אז הזיכרון שהוקצה עבור elem לא ישוחרר אף פעם!

הפתרון: שימוש ב-smart pointers.

התוכנית הבאה מדגימה את התופעה:

```
#include <iostream>
#include <memory>
#include <stdexcept>

using namespace std;

class Element
{
public:
	Element() { cout << "Ctor - Element" << endl; }
	~Element() { cout << "Dtor - Element" << endl; }

	void DoSomething()
	{
		throw runtime_error("Some Error");
	}
};

void func()
{
    Element *elem = new Element;
    elem->DoSomething();
    delete elem;
}

void func2()
{
    auto_ptr<Element> elem(new Element);
    elem->DoSomething();
}

int main()
{
    cout << "Stage 1" << endl;
    try
    {
        func();
    }
    catch(...) {}
```

```
	cout << "Stage 2" << endl;
	try
	{
		func2();
	}
	catch(...) {}

	return 0;
}
```

פלט התוכנית יהיה:

```
Stage 1
Ctor - Element
Stage 2
Ctor - Element
Dtor - Element
```

במקרה הראשון, בו השתמשנו במצביע רגיל, לא נקראה הפונקציה ההורסת של האובייקט.

דוגמאות לשימוש ב-Smart Pointers

בתוכנית הבאה ישנה מתודה המקצה זיכרון ומחזירה מצביע אליו. כאשר נגמר השימוש בו, הזכרון משתחרר מעצמו. שימוש זה ב-smart pointers הינו נפוץ ביותר.

```
#include <iostream>
#include <memory>

using namespace std;

class Element
{
public:
	Element() { cout << "Ctor - Element" << endl; }
	~Element() { cout << "Dtor - Element" << endl; }
};

typedef auto_ptr<Element> PElem;

class Generator
{
public:
	static PElem CreateElement()
	{
		PElem p(new Element());
		return p;
	}
};
```

```
int main()
{
	PElem ref = Generator::CreateElement();
	cout << "Main :: Before Exit" << endl;
	return 0;
}
```

פלט התוכנית:

```
Ctor - Element
Main :: Before Exit
Dtor - Element
```

נשים לב שלכאורה הזכרון יכל להשתחרר בסיום הפונקציה CreateElement, מכיוון ש-p הוא משתנה לוקלי שלה. אם זאת, מכיוון ש-p מועתק אל ref לפני שהוא משוחרר, ומכיוון שהעתקה גורמת ל-p להצביע על NULL, הזכרון לא משתחרר באמת, ובחזרה מהפונקציה ref מצביע על הזיכרון שיצרנו.

מקורות

1. Addison-Wesley (1999), "**Using auto_ptr Effectively**"
2. Andrei Alexandrescu (2003), "**Smart Pointers in C++**"
3. Herb Sutter (2002), "**The New C++: Smart(er) Pointers**"
4. Juha Vihavainen (2004), "**C++ Programming Techniques and Idioms**"
5. Yonat Sharon, (1999), "**Smart Pointers - What, Why, Which?**"