

## **Techniki Efektywnego Programowania – zadanie 5**

### **Własny inteligentny wskaźnik i *Move Semantics* (C++11 i wyżej)**

Inteligentne wskaźniki (ang. *smart pointers*), pozwalają na zautomatyzowane kasowanie zmiennych wymagających dealokacji. Zaletą inteligentnych wskaźników, w porównaniu z mechanizmami typu Garbage Collector jest to, że dokładnie wiadomo, kiedy dana pamięć zostanie zdealokowana. Przykład, kiedy programista musi pamiętać o zdealokowaniu pamięci.

```
void v_analize_sell_data(CDatabase *pcDb)
{
    CSellData *pc_datapack;
    pc_datapack = pcGetSellDataFromDb(pcDb);
    /*do sth with data*/
    delete pc_datapack;
} //void v_analize_sell_data()
```

W powyższym przykładzie, obiekt typu `CSellData` jest zwracany przez funkcję, która ściąga obiekt z jakiegoś repozytorium. Na końcu procedury `v_analize_sell_data` konieczne jest jego skasowanie, o czym programista może zapomnieć.

Jeżeli chcemy, żeby dynamicznie zaalokowana pamięć była automatycznie kasowana przy wyjściu z procedury `v_analize_sell_data`, to należy „opakować” ją w jakiś obiekt, który może być zaalokowany statycznie. Wtedy procedura, może wyglądać na przykład tak:

```
void v_analize_sell_data(CDatabase *pcDb)
{
    CMySmartPointer c_dpack(pcGetSellDataFromDb(pcDb));
    /*do sth with data*/
} //void v_analize_sell_data()
```

Obiekt `CMySmartPointer`, otrzymuje do przechowania wskaźnik na obiekt klasy `CSellData`. Obiekt `c_dpack`, zostanie usunięty ze stosu przy wyjściu z procedury `v_analize_sell_data`. Destruktor obiektu `c_dpack`, powinien kasować przechowywany wskaźnik na obiekt klasy `CSellData`. Klasa `CMySmartPointer` może wyglądać tak, jak poniżej.

**„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”**

```
class CMySmartPointer
{
public:
    CMySmartPointer(CSellData *pcPointer) { pc_pointer = pcPointer; }
    ~CMySmartPointer() { delete pc_pointer; }

    CSellData& operator*() { return(*pc_pointer); }
    CSellData* operator->() { return(pc_pointer); }

private:
    CSellData *pc_pointer;
}; //class CMySmartPointer
```

Dzięki takiej implementacji, jak powyżej, obiekt klasy `CSellData`, zostanie skasowany przy wyjściu z procedury `v_analize_sell_data`. Co więcej, dzięki przeciążeniu operatorów `*` i `->`, łatwo będzie odwoływać się do przechowywanego wskaźnika na obiekt klasy `CSellData`. Na przykład tak:

```
c_dpack->vPrintData();
(*c_dpack).vPrintData();
```

**Uwaga.** Zastanów się czy zamiast wskaźnika na konkretny typ, można przechowywać wskaźnik bardziej ogólny. Jeśli potrafisz wskazać taki typ wskaźnikowy zastanów się, jakie będą wady i zalety takiego rozwiązania.

Może nastąpić sytuacja, w której programista chce mieć więcej inteligentnych wskaźników, które przechowują ten sam wskaźnik. Żeby móc obsłużyć taką sytuację, należy przeciążyć konstruktor kopiujący (co najmniej). Może to wyglądać tak:

```
CMySmartPointer(const CMySmartPointer &pcOther) { pc_pointer = pcOther.pc_pointer; }
```

**Uwaga!** Powyższa implementacja prowadzi do błędu! Jeśli konstruktor kopiujący zostanie wykonany tak jak powyżej, to destruktory dwóch inteligentnych wskaźników będą próbowały skasować tę samą pamięć. Problem ten rozwiązuje się, poprzez wprowadzenie obiektu licznika odwołań.

```
class CRefCounter
{
public:
    CRefCounter() { i_count; }

    int iAdd() { return(++i_count); }
    int iDec() { return(--i_count); };
    int iGet() { return(i_count); }

private:
    int i_count;
}; //class CRefCounter
```

Obiekt licznika jest tworzony w klasie inteligentnego wskaźnika.



**„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”**

```
class CMySmartPointer
{
public:
    CMySmartPointer(CSellData *pcPointer)
    {
        pc_pointer = pcPointer;
        pc_counter = new CRefCount();
        pc_counter->iAdd();
    } // CMySmartPointer(CSellData *pcPointer)

    CMySmartPointer(const CMySmartPointer &pcOther)
    {
        pc_pointer = pcOther.pc_pointer;
        pc_counter = pcOther.pc_counter;
        pc_counter->iAdd();
    } // CMySmartPointer(const CMySmartPointer &pcOther)

    ~CMySmartPointer()
    {
        if (pc_counter->iDec() == 0)
        {
            delete pc_pointer;
            delete pc_counter;
        } // if (pc_counter->iDec())
    } // ~CMySmartPointer()

    CSellData& operator*() { return(*pc_pointer); }
    CSellData* operator->() { return(pc_pointer); }

private:
    CRefCount *pc_counter;
    CSellData *pc_pointer;
}; // class CMySmartPointer
```

W przypadku powyższej implementacji, wiele inteligentnych wskaźników może przechowywać ten sam wskaźnik i będzie mieć ten sam licznik odwołań. Jednak do błędu nie dojdzie. Obiekty wskazywane przez pc\_counter i pc\_pointer zostaną skasowane w momencie, gdy będzie kasowany ostatni przechowujący je inteligentny wskaźnik.

## **„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”**

Istnieją sytuacje, w których w jakieś funkcji dysponujemy stworzonym statycznie obiektem i wartość tego obiektu chcemy przekazać na zewnątrz. Jest to typowa sytuacja dla przeciążania operatorów. Na przykład:

```
CNumber CNumber::operator+(CNumber &cNum)
{
    CNumber c_result;
    /*create result*/
    return(c_result);
} //CNumber CNumber::operator+(CNumber &cNum)
```

Powyższa implementacja operatora dodawania jest wygodna, ponieważ gdyby obiekt `c_result` był dynamicznie alokowany, a operator zwracałby wskaźnik na `CNumber`, a nie wartość `CNumber`, to gdyby na zewnątrz wynik nie został przypisany do żadnej zmiennej doszłoby do wycieku. Jednocześnie, jeśli klasa `CNumber` alokuje sporo pamięci, to usuwanie `c_result` na końcu operatora może być uznane za marnotrawstwo, ponieważ ten obiekt nie zostanie już nigdzie użyty, a przecież pamięć, którą zaalokował mogłaby zostać przekazana do użytkownika do innego obiektu. Zamiast tego pamięć będzie kopiowana, co jest czasochłonne.

Do radzenia sobie w powyższych sytuacjach służy tzw. semantyka przenoszenia (ang. *move semantics* (MS)). Rozważmy przykład z klasą `CTab`.

```
#define DEF_TAB_SIZE 10
class CTab
{
public:
    CTab() { pi_tab = new int[DEF_TAB_SIZE]; i_size = DEF_TAB_SIZE; }
    CTab(const CTab &cOther);
    CTab(CTab &&cOther);
    CTab operator=(const CTab &cOther);
    ~CTab();

    bool bSetSize(int iNewSize);
    int iGetSize() { return(i_size); }
private:
    void v_copy(const CTab &cOther);

    int *pi_tab;
    int i_size;
}; //class CTab
```



**„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”**

Wybrane metody klasy CTab.

```
CTab::CTab(const CTab &cOther)
{
    v_copy(cOther);
    std::cout << "Copy ";
} //CTab::CTab(const CTab &cOther)

CTab::~CTab()
{
    if (pi_tab != NULL) delete pi_tab;
    std::cout << "Destr ";
} //CTab::~CTab()

CTab CTab::operator=(const CTab &cOther)
{
    if (pi_tab != NULL) delete pi_tab;
    v_copy(cOther);

    std::cout << "op= ";

    return(*this);
} //CTab CTab::operator=(const CTab &cOther)

void CTab::v_copy(const CTab &cOther)
{
    pi_tab = new int[cOther.i_size];
    i_size = cOther.i_size;

    for (int ii = 0; ii < cOther.i_size; ii++)
        pi_tab[ii] = cOther.pi_tab[ii];
} //void CTab::v_copy(CTab &cOther)
```

W tradycyjny sposób (z kopiowaniem) klasy możemy użyć tak, jak w poniższym programie.

```
CTab cCreateTab()
{
    CTab c_result;
    c_result.bSetSize(5);
    return(c_result);
} //CTab cCreateTab()

int i_ms_test()
{
    CTab c_tab = cCreateTab();
    /*DO STH WITH c_tab*/
} //int i_ms_test()
```

**Jeżeli** kompilator **nie zoptymalizuje** kodu w trakcie kompilacji, to obiekt c\_tab zostanie stworzony przy pomocy konstruktora kopiującego i dojdzie do kopiowania tablicy. Możemy jednak zdefiniować konstruktor przenoszący o następującej treści.



**„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”**

```
CTab::CTab(CTab &&cOther)
{
    pi_tab = cOther.pi_tab;
    i_size = cOther.i_size;

    cOther.pi_tab = NULL;

    std::cout << "MOVE ";
} //CTab::CTab(CTab &&cOther)
```

Zmiał kopiować pamięć, jak w konstruktorze kopiującym, przepisujemy wskaźnik na już zaalokowaną tablicę do nowego obiektu, a w starym obiekcie ustawiamy wskaźnik na **NULL**. Jest to ważne, ponieważ nie chcemy, żeby destruktor starego obiektu zwolnił pamięć tablicy. Tablica jest *przenoszona* do nowego obiektu.

Samo zadeklarowanie konstruktora przenoszącego nie spowoduje jego użycia. Żeby do niego doszło należy użyć funkcji `std::move`.

```
CTab cCreateTab()
{
    CTab c_result;
    c_result.bSetSize(5);
    return(std::move(c_result));
} //CTab cCreateTab()
```

W powyższym przykładzie obiekt `c_result` jest zwracany przez wartość, ale zamiast konstruktora kopiującego zostanie użyty konstruktor przenoszący. Zauważ, że przy wykonaniu procedury `i_ms_test`, tablica, na którą wskazuje wskaźnik `pi_tab` obiektu `c_result`, nie zostanie skasowana, ale przekazana do obiektu `c_tab`, za pomocą konstruktora przenoszącego. Jednocześnie, gdyby użyć funkcji `cCreateTab` w poniższy sposób, to nie nastąpi wyciek pamięci, ponieważ dla procedury `i_ignore_result` nie zostanie wywołany konstruktor przenoszący, którego argumentem byłby obiekt `c_result` z funkcji `cCreateTab`. Dlatego `c_result` skasuje tablicę, przy wywołaniu swojego destruktora.

```
int i_ignore_result()
{
    cCreateTab();
    /*DO STH WITH */
} //int i_ignore_result()
```

**„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”**

**Zadanie**

**UWAGI:**

1. Pisząc własny program można użyć innego nazewnictwa niż to przedstawione w treści zadania i w przykładach. Należy jednak użyć jakiejś spójnej konwencji kodowania, zgodnie z wymaganiami kursu.
2. Nie wolno używać wyjątków (jest to jedynie przypomnienie, wynika to wprost z zasad kursu).
3. Wolno używać wyłącznie komend ze standardu C++98.
4. Od niniejszego ćwiczenia można korzystać z inteligentnych wskaźników, **ale wyłącznie takich, które zostały napisane samodzielnie.**

1. Korzystając z umiejętności zdobytych na poprzednich laboratoriach zamień klasę `CMySmartPointer` na klasę szablonową.
2. Dodaj obsługę operatora `=`, tak aby jednemu inteligentnemu wskaźnikowi można było przypisać wartość innego. Pamiętaj, że jeśli modyfikowany inteligentny wskaźnik, wskazywał przechowywał przed przypisaniem inny wskaźnik, to należy zdekrementować licznik odwołań i dokonać dealokacji, jeśli zajdzie taka potrzeba.
3. Zastanów się co się stanie, gdy inteligentny wskaźnik będzie przechowywał wskaźnik na pamięć zaalokowaną statycznie.
4. Konstruktor przenoszący dla klasy `CTab` to wygodny mechanizm. Jednak byłoby wygodnie, gdyby można było używać MS, również w przypadku zapisu:

```
CTab c_tab;
CTab c_other;
/*initialize c_tab, c_other*/
c_tab = std::move(c_other);
```

Domyślna treść operatora przeniesienia (`CTab operator=(const CTab &&cOther);`) jest pusta, a więc nie spełni ona oczekiwań użytkownika. Napisz taką treść operatora przeniesienia, która będzie prawidłowa dla klasy `CTab`.

5. Zmodyfikuj klasę `CTable` wykonaną w ramach ćwiczeń nr 2 i 3. Zmień operatory tak, aby zwracały wszystkie wyniki przez wartość (jeżeli tak nie robią), ale użyj do tego move semantics. Sprawdź o ile spadła liczba wykonanych kopii przy użyciu move semantics i bez nich.

**Zalecana literatura**

Jerzy Grębosz „Symfonia C++”, Wydawnictwo Edition, 2000.

Wykład

Materiały możliwe do znalezienia w Internecie

Statsiewicz A., C++11. Nowy standard. Ćwiczenia, Helion, 2012

Stephen Prata, Język C++. Szkoła programowania. Wydanie VI, Helion, 2012

***„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”***

Nicolai M. Josuttis, C++. Biblioteka standardowa. Podręcznik programisty. Wydanie II, Helion, 2014Materiały możliwe do znalezienia w Internecie