



Uniwersytet
Wrocławski

Inteligentne wskaźniki (*smart pointers*)

Zbigniew Koza

Wydział Fizyki i Astronomii

Co jest nie tak ze zwykłymi wskaźnikami?

- Wskazują na obiekt, ale nie rozwiązują problemu, kto jest jego właścicielem
- Typowe błędy:
 - Niezainicjalizowany wskaźnik;
 - Utrata dostępu do zaalokowanej pamięci
 - Podwójne zwolnienie tej samej pamięci
 - Niepoprawne zwolnienie zaalokowanej pamięci
 - Niepoprawne zaalokowanie pamięci
 - Próba zapisu pod `nullptr`
 - Próba zapisu lub odczytu z pamięci, która nie należy do programu / zmiennej / tablicy
 - Utrata ważności przez obiekt wskazywany przez wskaźnik
 - Niepoprawny typ obiektu wskazywanego przez wskaźnik

Co jest nie tak ze zwykłymi wskaźnikami?

- Najprostsze, zalecane rozwiązanie:

Nie używaj „nagich” wskaźników,

- chyba że w ściśle zamkniętych implementacjach bardzo podstawowych klas, bibliotek,
- lub gdy używasz bibliotek, których interfejsy napisano w stylu języka C („nagie wskaźniki”).

Co jest nie tak ze zwykłymi wskaźnikami?

- Wskaźników od biedy można używać w kodzie, w którym nie ma wątpliwości, kto jest właścicielem obiektów wskazywanych przez wskaźniki (zwykle: „nie ja”) i jaki jest ich czas życia (zwykle: dłuższy od czasu życia funkcji)

Inteligentne wskaźniki

Trzy rodzaje

`std::unique_ptr`

`std::shared_ptr`

`std::weak_ptr`

- Każdy z nich to „wrapper” na nagi wskaźnik, służący nieco innym celom
- Przeciążone operatory `*` i `->`.

Trzy rodzaje

`std::unique_ptr`

`std::shared_ptr`

`std::weak_ptr`

- Każdy z nich automatycznie wywoła **destruktor obiektu** wskazywanego przez wskaźnik i to dokładnie wtedy, kiedy będzie to potrzebne
 - Koniec problemów z **delete**

std::unique_ptr

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

int main()
{
    int *p = new int (7); // niebezpiecznie
    std::unique_ptr<int> ptr2 (new int (8)); // niebezpiecznie
    std::unique_ptr<int> ptr = std::make_unique<int>(9); // OK
    const auto & ptr3 = ptr; // można tworzyć referencje
    // ptr = ptr2; // błąd: unique_ptr nie można kopiować
    ptr = std::move(ptr2); // OK, można przesuwać
    std::cout << *p << *ptr << *ptr2 << *ptr3 << "\n";
    delete p;
}
```

- Jeden właściciel wskazywanego obiektu

std::unique_ptr

- Do inicjalizacji używaj `std::make_unique`
 - Bo inaczej twój program nie jest *exception-safe*
- Wskazywany obiekt ma zawsze jednego właściciela, który w destruktorze wywoła operator `delete`
- Nie możesz kopiować
- Możesz przesuwąć (`std::move`)
 - „*ownership transfer*”

std::unique_ptr

- Używaj w „fabrykach” (*factories*) zasobów (zawsze jest jeden właściciel zasobu)
- Funkcja pobierająca / zwracająca **unique_ptr** transferuje nie tylko wskaźnik, ale **relację własności**
- Używaj też tam, gdzie czas życia zasobu ograniczony ma być do
 - zakresu funkcji
 - lub czasu życia obiektu

std::unique_ptr

- A co z tablicami?

```
std::unique_ptr<T[]> ptr =  
std::make_unique<T[]>(3);
```

(tylko po co?)

std::shared_ptr

- Jak nazwa wskazuje: obiekt zarządzany przez `shared_ptr` **może mieć więcej niż jednego właściciela**
- Obiekt zostanie automatycznie zniszczony wraz z końcem życia ostatniego `shared_ptr`, który nim zarządza
- Typowa implementacja: *reference counting*

std::shared_ptr

- Do inicjalizacji używaj make_shared

```
int main()
{
    std::shared_ptr<T> ptr = std::make_shared<T>(3);
    std::shared_ptr<T> ptr2 = ptr;
}
```

- shared_ptr można kopiować

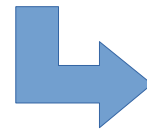
std::shared_ptr

```
struct T {
    int n;
    T(int n = 0): n{n} {std::cout<<"creating T, n = "<<n<<"\n";}
    ~T() {std::cout << "deleting T with n = " << n << "\n";}
    operator int() const {return n;}
    std::shared_ptr<T> other;
};

int main() {
    std::shared_ptr<T> p1 (std::make_shared<T>());
    std::shared_ptr<T> p2 (std::make_shared<T>());
    std::cout << p1.use_count() << "\n";
    p1->other = p2; // p1 references p2
    p2->other = p1; // p2 references p1
    std::cout << p1.use_count() << "\n";
}
```

use_count(): liczba
właścicieli obiektu

Wyciek pamięci (brak destruktatorów)



```
creating T, n = 0
creating T, n = 0
1
2
```

- shared_ptr można niechcący „zapętlić”

std::weak_ptr

- Zasadniczo to jest coś jak `shared_ptr`, ale nie biorące udziału w *reference counting*, czyli wskaźnik-obszernik
 - Można używać gdy nie mamy pewności, czy obiekt zarządzany przez inteligentny wskaźnik nie został zwolniony
 - Można zapobiegać zapętlaniu `shared_ptr`
 - Można promować do `shared_ptr`: jeżeli zarządzany obiekt „zniknął”, to `shared_ptr` będzie zawierał `null_ptr`

Podsumowanie

- Inteligentnych wskaźników używa się do zwiększenia niezawodności oprogramowania
 - Ograniczenie wycieków pamięci i niepoprawnego użycia pamięci
- Są one zgodne z zasadą **RAII**
- Prawie zawsze wystarcza **unique_ptr<T>**