




C++17 i STL

Inteligentne wskaźniki



Dlaczego nie lubimy surowych wskaźników?

- ▶ Na podstawie deklaracji wskaźnika nie można stwierdzić, czy wskazywany jest pojedynczy obiekt czy tablica.
- ▶ Deklaracja wskaźnika nie mówi nic o tym, czy po zakończeniu używania wskazywanego obiektu należy go usunąć.
- ▶ Jeżeli nawet dowiemy się, że powinniśmy usunąć wskazywany obiekt, to nie wiadomo jak to zrobić (czy należy użyć operatora delete, czy istnieje inny mechanizm destrukcyjny).
- ▶ Jeżeli uda się nam stwierdzić, że delete jest dobrym sposobem do usunięcia obiektu, to dalej nie wiadomo, czy użyć formy dla pojedynczego obiektu delete czy dla tablicy delete[].
- ▶ Zakładając, że ustalimy, że wskaźnik posiada to, co wskazuje, i odkryjemy, jak to zniszczyć, nadal będzie trudno zapewnić, że wykonamy destrukcję dokładnie raz na każdej ścieżce kodu.
- ▶ Zwykle nie ma sposobu stwierdzenia, że wskaźnik jest wiszący, czyli że wskazuje pamięć, która już nie przechowuje wskazywanego elementu.



Inteligentne wskaźniki

- ▶ Wskaźniki inteligentne są sposobem rozwiązania tych problemów.
- ▶ Wskaźniki inteligentne są zbudowane na podstawie wskaźników surowych i działają podobnie do nich, ale pozwalają uniknąć wielu związanych z nimi pułapek.
- ▶ Zawsze powinniśmy wybierać wskaźniki inteligentne zamiast wskaźników surowych.
- ▶ Wskaźniki inteligentne mogą wykonać prawie wszystko, co wskaźniki surowe, ale zmniejszają liczbę możliwości popełnienia błędów.
- ▶ Istnieją trzy rodzaje wskaźników inteligentnych: `std::unique_ptr`, `std::shared_ptr` i `std::weak_ptr`.



Inteligentne wskaźniki

- ▶ Jednym z zadań wskaźników jest udostępnienie semantyki referencji, obejmującej więcej niż bieżący zasięg widoczności.
- ▶ Właściwe zarządzanie czasem życia wskaźników i obiektów przez nie wskazywanych jest trudne, zwłaszcza jeśli na ten sam obiekt wskazuje większa liczba wskaźników.
- ▶ Rola inteligentnych wskaźników:
 - ▶ mają zadbać o to, aby obiekt istniał co najmniej tak długo, jak długo istnieją wskazujące do niego wskaźniki;
 - ▶ obiekt powinien zostać zwolniony, jeśli zostanie usunięty ostatni wskazujący na niego wskaźnik.



Inteligentne wskaźniki w C++98

- ▶ Klasa `auto_ptr<>` w bibliotece standardowej C++98 została zaprojektowaną do zadań związanych z RAII (pozyskiwanie zasobów poprzez inicjalizację).
- ▶ Z uwagi na niedostatek elementów języka C++98 w rodzaju semantyki przeniesienia w konstruktorach i operatorach przypisania klasa ta była często używana niepoprawnie.
- ▶ Obecnie rolę `auto_ptr<>` realizuje klasa `unique_ptr<>` a klasa `auto_ptr<>` została uznana za przeznaczoną do wycofania z nastaniem C++11 i wycofana ze standardu C++17.


Inteligentne wskaźniki w C++11

- ▶ Wszystkie klasy inteligentnych wskaźników w C++ są zdefiniowane w pliku nagłówkowym `<memory>`.
- ▶ Klasa `shared_ptr<>` dla wskaźników implementuje pojęcie współdzielonej własności obiektu. Klasa ta pozwala na współistnienie wielu wskaźników do tego samego obiektu wskazywanego i implementuje zwalnianie obiektu wskazywanego w momencie zwalniania ostatniego wskaźnika.
 - ▶ Do realizacji tego zadania w mniej typowych przypadkach powołano klasę pomocniczą `weak_ptr<>`.
- ▶ Klasa `unique_ptr<>` dla wskaźników implementuje pojęcie wyłącznej własności obiektu. Klasa ta gwarantuje istnienie dokładnie jednego wskaźnika do obiektu wskazywanego w danym momencie. Pozwala jednak na przenoszenie własności.
 - ▶ Klasa ta zabezpiecza przed wyciekami zasobów w kontekście występowania wyjątków.



Klasa `shared_ptr`

- ▶ Każdy nietrywialny program operuje obiektami widocznymi z wielu miejsc i w różnych momentach wykonania – potrzebne są więc „odwołania” do obiektów, używane w rozmaitych miejscach programu.
- ▶ Sam język udostępnia wskaźniki i referencje, ale nie są one wystarczająco bezpieczne – niejednokrotnie zachodzi potrzeba zagwarantowania usunięcia obiektu wskazywanego wraz z usunięciem ostatniego „odwołania”, co ma zapewniać wykonanie operacji porządkujących stan programu (zwolnienie pamięci, zwolnienie zasobów skojarzonych z obiektem).
- ▶ Semantykę „sprzątanania po obiekcie, kiedy ten nie jest już nigdzie używany” realizuje klasa `shared_ptr`.
 - ▶ Klasa `shared_ptr` daje możliwość używania wielu wskaźników odnoszących się do tego samego obiektu.
 - ▶ Ostatni istniejący wskaźnik współdzielony, będący właścicielem obiektu wskazywanego, jest odpowiedzialny za zwolnienie obiektu i jego zasobów.



Klasa `shared_ptr` – zwalnianie zasobów

- ▶ Zadaniem wskaźników typu `shared_ptr` jest automatyzacja usuwania zasobów skojarzonych z obiektem wskazywanym w momencie, kiedy obiekt ten nie będzie już więcej używany.
- ▶ Domyślnie zwolnienie obiektu polega na wywołaniu na jego rzecz operatora `delete` (przy założeniu, że obiekt został utworzony wywołaniem `new`).
- ▶ Możliwe jest jednak (a czasem konieczne) definiowanie własnych operacji porządkowania stanu obiektu w postaci własnych wytycznych zwalniania obiektu wskazywanego:
 - ▶ jeśli na przykład obiekt wskazywany jest tablicą przydzielaną przez `new []`, zwolnienie obiektu powinno odbywać się przez wywołanie `delete []`;
 - ▶ inne wytyczne zwalniania będą dotyczyły obiektów przechowujących dodatkowe zasoby, takie jak pliki, uchwyty systemowe, blokady itp.

Klasa `shared_ptr` – stosowanie wskaźników współdzielonych

- ▶ Klasę `shared_ptr` stosuje się bardzo podobnie jak zwyczajny wskaźnik: wskaźniki współdzielone można przypisywać, kopiować i porównywać, a także stosować z operatorami wyłuskania `*` i `->`, udostępniającymi obiekt wskazywany i jego składowe.
- ▶ Przykład. Deklarujemy i inicjalizujemy dwa wskaźniki współdzielone typu `shared_ptr`, ujmujące wskaźniki do dwóch łańcuchów znakowych:

```
shared_ptr<string> pNico(new string("nico"));
shared_ptr<string> pJutta(new string("jutta"));
```
- ▶ Ponieważ konstruktor `shared_ptr` z pojedynczym argumentem jest deklarowany jako `explicit`, nie można tu zastosować składni inicjalizacji przez przypisanie.
- ▶ Można za to użyć składni jednolitej inicjalizacji:

```
shared_ptr<string> pNico { new string("nico") };
```
- ▶ Można też skorzystać z funkcji pomocniczej `make_shared()`:

```
shared_ptr<string> pJutta =
    make_shared<string>("jutta");
```

Klasa `shared_ptr` – stosowanie wskaźników współdzielonych

- ▶ Wskaźnik współdzielony typu `shared_ptr` można zadeklarować osobno i później przypisać do niego nowy wskaźnik – można to zrobić tylko za pośrednictwem funkcji składowej `reset()`:

```
shared_ptr<string> pNico4;
```

```
...
```

```
pNico4.reset(new string("nico"));
```

- ▶ Używanie wskaźników typu `shared_ptr` nie różni się od stosowania zwyczajnych wskaźników:

```
(*pNico)[0] = 'N';
```

```
pJutta->replace(0,1,"J");
```

Klasa `shared_ptr` – stosowanie wskaźników współdzielonych

- ▶ Oba wskaźniki współdzielone z poprzedniego przykładu wielokrotnie umieścimy w kontenerze typu `vector<>`:

```
vector<shared_ptr<string>> whoMadeCoffee;
whoMadeCoffee.push_back(pJutta);
whoMadeCoffee.push_back(pJutta);
whoMadeCoffee.push_back(pNico);
whoMadeCoffee.push_back(pJutta);
whoMadeCoffee.push_back(pNico);

...
for (auto ptr : whoMadeCoffee)
    cout << *ptr << " ";
cout << endl;
```

- ▶ Kontener zwyczajowo przechowuje kopie przekazywanych elementów, co oznacza, że gdybyśmy umieszczali tam wprost łańcuchy znakowe, kontener wykonałby ich kopie. Ponieważ jednak umieszczamy w nim współdzielone wskaźniki, kontener przechowuje kopie wielu wskaźników odnoszących się do tego samego obiektu.
- ▶ Funkcja składowa `use_count()` zwraca liczbę istniejących w danym momencie wskaźników współdzielących dostęp do tego samego obiektu wskazywanego.

Klasa `shared_ptr` – stosowanie wskaźników współdzielonych

- Do wcześniejszego kodu dopiszemy jeszcze:

```
*pNico = "Nicolai";  
for (auto ptr : whoMadeCoffee)  
    cout << *ptr << " ";  
cout << endl;  
cout << "use_count: " <<  
whoMadeCoffee[0].use_count() << endl;
```
- Otrzymamy ostatecznie następujące wyniki:

```
Jutta Jutta Nico Jutta Nico  
Jutta Jutta Nicolai Jutta Nicolai  
use_count: 4
```



Klasa `shared_ptr` – własna funkcja usuwająca

- ▶ Wskaźnik typu `shared_ptr` pozwala na określenie własnej funkcji usuwającej.
- ▶ Drugim argumentem konstruktora typu `shared_ptr` może być funkcja, funktor albo lambda – wówczas inteligentny wskaźnik uruchomi podaną funkcję tuż przed zwolnieniem wskazywanego obiektu.

- ▶ Przykład definicji inteligentnego wskaźnika.

```
shared_ptr<string> pNico(  
    new string("nico"),  
    [] (string *p) {  
        cerr << "delete " << *p << endl;  
        delete p;  
    }  
);
```

- ▶ Przykład zadziałania funkcji usuwającej:


```
pNico = nullptr;  
whoMadeCoffee.resize(2);
```

Klasa `shared_ptr` – obsługa tablic

- ▶ Domyślna operacja zwalniania implementowana we wskaźnikach typu `shared_ptr` sprowadza się do wywołania `delete`, nigdy `delete []` – oznacza to, że domyślna funkcja usuwająca jest właściwa wyłącznie dla wskaźników do obiektów tworzonych za pośrednictwem operatora `new`.
- ▶ Przykład inteligentnego wskaźnika obsługującego tablicę:

```
shared_ptr<int> p(  
    new int[10],  
    [] (int *p) { delete[] p; }  
);
```
- ▶ Można też skorzystać z udogodnienia przewidzianego pierwotnie do użycia z klasą typu `unique_ptr`, definiującego funkcję usuwającą jako `delete []`:

```
shared_ptr<int> p(  
    new int[10],  
    default_delete<int[]>()  
);
```

Klasa `shared_ptr` – funkcja usuwająca

- ▶ Jeśli usuwanie obiektu wskazywanego wraz z ostatnim współdzielonym wskaźnikiem nie sprowadza się do zwolnienia pamięci, należy dostarczyć własną funkcję usuwającą – w ten sposób określa się własną regułę destrukcji obiektu wskazywanego.

Klasa `shared_ptr` – funkcja usuwająca plik

- ▶ Przykład funktora usuwającego plik.

```
class FileDeleter
{
private:
    string filename;
public:
    FileDeleter (const string& fn)
        : filename(fn) {}
    void operator () (ofstream* fp) {
        fp->close();
        remove(filename.c_str());
    }
};
```

- ▶ Przykład wykorzystania funktora usuwającego plik.

```
shared_ptr<ofstream> fp(
    new std::ofstream("tmpfile.txt"),
    FileDeleter("tmpfile.txt")
);
```



Klasa weak_ptr

- ▶ Klasa `shared_ptr` pozwala na automatyczne zwalnianie zasobów skojarzonych z nieużywanymi już obiektami. Jednak w pewnych okolicznościach zachowanie to nie jest wcale korzystne ani oczekiwane:
 - ▶ w przypadku odwołań cyklicznych, kiedy dwa obiekty odwołują się do siebie nawzajem za pośrednictwem wskaźników typu `shared_ptr`, nigdy nie dojdzie do zwolnienia zasobów skojarzonych z tymi obiektami, bo nigdy nie dojdzie do wyzerowania liczby odwołań;
 - ▶ w przypadku, gdy czas życia wskaźnika do obiektu jest dłuższy niż czas życia obiektu wskazywanego – w takiej sytuacji użycie wskaźników `shared_ptr` uniemożliwi zwolnienie zasobów, natomiast użycie zwyczajnych wskaźników narażałoby na niepoprawne próby odwołań do już nieistniejących obiektów wskazywanych.
- ▶ Dla takich właśnie przypadków przewidziano klasę wskaźnika „słabego” `weak_ptr`, pozwalającą na współdzielenie dostępu do obiektu wskazywanego bez przejmowania współwłasności tego obiektu, a tym samym odpowiedzialności za jego zwolnienie.



Klasa `weak_ptr`

- ▶ Kiedy ostatni współdzielony wskaźnik `shared_ptr` wymusi zwolnienie obiektu wskazywanego, wszystkie „słabe” wskaźniki automatycznie zostaną wyzerowane – dlatego klasa `weak_ptr` poza konstruktorem domyślnym i kopiującym definiuje jedynie konstruktor przyjmujący argument typu `shared_ptr`.
- ▶ Obiekt wskazywany przez wskaźnik `weak_ptr` nie jest wprost dostępny przez operatory `*` i `->`.
- ▶ Klasa `weak_ptr` udostępnia naprawdę skromny interfejs, wystarczający jedynie do utworzenia, skopiowania i przypisania słabego wskaźnika oraz do wykonania konwersji na wskaźnik współdzielony tudzież do sprawdzenia, czy wskazuje on jeszcze poprawny obiekt.

Klasa weak_ptr – zastosowanie

► Przykład.

```
class Person {
public:
    string name;
    shared_ptr<Person> mother;
    shared_ptr<Person> father;
    vector<weak_ptr<Person>> kids;
    Person(
        const string& n,
        shared_ptr<Person> m = nullptr,
        shared_ptr<Person> f = nullptr)
        : name(n), mother(m), father(f) {}
    ~Person() {
        cout << "usuwam " << name << endl;
    }
};
```

Klasa weak_ptr – zastosowanie

► Przykład:

```
shared_ptr<Person> initFamily (const string& name) {  
    shared_ptr<Person> mom(new Person("mama "+name));  
    shared_ptr<Person> dad(new Person("tata "+name));  
    shared_ptr<Person> kid(new Person(name,mom,dad));  
    mom->kids.push_back(kid);  
    dad->kids.push_back(kid);  
    return kid;  
}
```




Klasa `unique_ptr`

- ▶ Klasa `unique_ptr` implementuje koncepcję wyłącznego posiadania obiektu wskazywanego, co oznacza, że obiekt wskazywany jest w danym momencie w posiadaniu dokładnie jednego wskaźnika wyłącznego.
- ▶ Kiedy wskaźnik wyłączny jest usuwany albo staje się wskaźnikiem pustym bądź zaczyna wskazywać do innego obiektu, dotychczasowy obiekt wskazywany również jest usuwany (co może oznaczać nie tylko zwolnienie pamięci obiektu i wywołanie destruktor, ale również wykonanie innych operacji porządkujących).
- ▶ Klasa `unique_ptr` zastąpiła klasę `auto_ptr` wprowadzoną w C++98 - klasa ta definiuje prostszy i czytelniejszy interfejs, przez co jest znacznie mniej podatna na nieprawidłowe użycia niż klasa `auto_ptr`.



Klasa `unique_ptr`

- ▶ Wskaźnik `std::unique_ptr` uosabia semantykę wyłącznego posiadania.
- ▶ Wskaźnik `std::unique_ptr` o wartości innej niż `nullptr` zawsze posiada to, co wskazuje.
- ▶ Przenoszenie wskaźnika `std::unique_ptr` powoduje zmianę stanu posiadania z wskaźnika źródłowego na docelowy – wskaźnik źródłowy przyjmuje wartość `nullptr`.
- ▶ Kopiowanie wskaźnika `std::unique_ptr` nie jest dozwolone.
- ▶ Wskaźnik `std::unique_ptr` jest typem dopuszczającym tylko przenoszenie.
- ▶ W ramach destrukcji wskaźnik `std::unique_ptr` o wartości innej niż `nullptr` niszczy swój zasób przez zastosowanie instrukcji `delete`.



Klasa `unique_ptr`

- ▶ Wskaźnik `std::unique_ptr` często jest stosowany jako typ, który zwraca funkcja fabrykująca dla obiektów w hierarchii.
- ▶ Kod wywołujący przejmuje odpowiedzialność za zasób zwracany przez funkcję fabrykującą (czyli wyłączone posiadanie zasobu), a wskaźnik `std::unique_ptr`, gdy jest niszczone, automatycznie usuwa to, na co wskazuje.
- ▶ Wskaźnika `std::unique_ptr` możemy też użyć w scenariuszach z migracją posiadania, gdzie wskaźnik `std::unique_ptr` zwracany przez funkcję fabrykującą jest przenoszony do kontenera, element kontenera jest następnie przenoszony do danych składowych obiektu, a następnie obiekt jest niszczone.

Klasa `unique_ptr` – obsługa tablic

- ▶ W przypadku wskaźników typu `unique_ptr` zdefiniowanie własnej funkcji usuwającej wymaga jawnego dookreślenia drugiego parametru szablonu:

```
unique_ptr<int, void(*) (int*)> p(  
    new int[10],  
    [](int* p) { delete[] p; }  
);
```

- ▶ Wskaźnik typu `shared_ptr` nie definiuje operatora indeksowania tablic `[]`. Tymczasem dla typu `unique_ptr` istnieje częściowa specjalizacja dla typów tablicowych, z operatorem indeksowania `[]` zamiast operatorów `* i ->`.

```
unique_ptr<int[]> tab(new int[10]);
```

Klasa `unique_ptr` – jak nie robić błędów

- ▶ `std::unique_ptr<int> up = new int; // BŁĄD!!!`
`std::unique_ptr<int> up(new int); // OK`
- ▶ Transfer obiektu wskazywanego:
`std::string* sp = new std::string("witaj");`
`std::unique_ptr<std::string> up1(sp);`
`std::unique_ptr<std::string> up2(sp);`
`// BŁĄD czasu wykonania: up1 i up2 posiadają ten sam obiekt`