SMART POINTERY

MATEUSZ MIGOT

KRZYSZTOF DEC

CZYM SĄ SMART POINTERY?

- To obiekty zachowujące się jak wskaźniki, ale oferujące dodatkowe usługi.
- Przy użyciu zwykłego wskaźnika trzeba pamiętać o każdorazowym zwalnianiu pamięci co nie jest rozwiązaniem idealnym, gdyż łatwo o tym zapomnieć.
- Wskaźniki inteligentne (auto_ptr, unique_ptr, shared_ptr, weak_ptr) definiują obiekty podobne do wskaźnika uzyskanego - bezpośrednio lub pośrednio - w wyniku wywołania instrukcji new. Kiedy inteligentny wskaźnik jest usuwany, jego destruktor używa instrukcji delete do zwolnienia pamięci. Dlatego jeśli przypiszemy do niego adres zwracany przez instrukcję new, nie musimy pamiętać o późniejszym zwalnianiu tej pamięci. Jest ona zwalniana automatycznie, w momencie usuwania obiektu inteligentnego wskaźnika.

RODZAJE SMART POINTEROW

- auto_ptr

W dalszym rozważaniu nie będziemy zajmować się szablonem auto_ptr. Od C++11 uznaje się je za przestarzałe i proponuje w to miejsce pozostałe trzy, które zostały wprowadzone wraz ze standardem C++11.

W C++11 zostały zdefiniowane 3 inteligentne wskaźniki, wszystkie zdefiniowane w bibliotece <memory>

- unique_ptr
- shared_ptr
- weak_ptr

UNIQUE_PTR

```
std::unique_ptr<type_name> ptr(new typename);
```

- Inteligentne wskaźniki klasy unique_ptr są unikalne. Korzystając z unikalnego wskaźnika gwarantujemy, że on i tylko on będzie wskazywał na dany obszar w pamięci. Jest on właścicielem tego miejsca. Zatem skorzystamy z niego w momencie, gdy będziemy chcieli mieć pewność, że pamięć nie zostanie swobodnie przekazana do innego wskaźnika.

SHARED_PTR

```
std::shared_ptr<int> p_shared = std::make_shared<int>(100);
```

- Wskaźnik współdzielony. Różnica względem wskaźnika std::unique_ptr<> polega na tym, że można go kopiować. Kopie wskaźników std::shared_ptr<> utrzymują między sobą licznik referencji, zwiększany w momencie utworzenia kopii, zmniejszany w momencie zniszczenia kopii. Pamięć zostaje zwolniona według reguły "kto ostatni, ten sprząta".
- Shared_ptr użyjemy w przypadku, gdy będziemy chcieli umożliwić wskazywanie tego samego miejsca w pamięci przez kilka wskaźników. Inteligentne wskaźniki shared_ptr zostały obmyślone w taki sposób, aby w przypadku korzystania z nich również nie doszło do próby zwolnienia tego samego obszaru pamięci przez kilka destruktorów. Zaimplementowano w tym celu rodzaj licznika zliczającego liczbę wskaźników wskazujących na tą samą pamięć. Dzięki temu do zwolnienia pamięci dojdzie wówczas, gdy żywot zakończy ostatni z wskaźników.

WEAK_PTR

- weak_ptr jest w zasadzie wskaźnikiem dzielonym(shared_ptr) z tą różnicą, że weak_ptr nie korzysta z licznika referencji
- Można stworzyć weak_ptr tylko ze wskaźnika shared_ptr lub z innego wskaźnika weak_ptr
- Wskaźnik typu `weak_ptr` obserwuje wskazywany obiekt, ale nie ma kontroli nad czasem jego życia i nie może zmieniać jego licznika odniesień.
- Nie udostępnia operatorów dereferencji. Aby mieć dostęp do wskazywanego obiektu konieczne jest dokonanie konwersji do `shared_ptr`.

PRZYKŁADY UŻYCIA SHARED_PTR I WEAK_PTR

shared_ptr

```
std::unique_ptr<int>    p1(new int);
std::unique_ptr<int[]>  p2(new int[50]);
std::unique_ptr<Object> p3(new Object("Lamp"));
```

weak_ptr

```
std::shared_ptr<int> p_shared = std::make_shared<int>(100);

std::weak_ptr<int>   p_weak1(p_shared);
std::weak_ptr<int>   p_weak2(p_weak1);

//in action
std::shared_ptr<int> p_shared = std::make_shared<int>(100);
std::weak_ptr<int>   p_weak(p_shared);
// ...
std::shared_ptr<int> p_shared_orig = p_weak.lock();
//
```

WYBÓR INTELIGENTNEGO WSKAŹNIKA

- Jeśli program używa obiektu wskazywanego za pośrednictwem więcej niż jednego wskaźnika, należałoby skorzystać ze wskaźnika shared_ptr - na przykład jeśli w programie używana jest tablica wskaźników wraz z pewnymi wskaźnikami pomocniczymi do identyfikowania poszczególnych elementów. Zdarza się używać dwóch obiektów różnych rodzajów, zawierających wskaźniki do tego samego trzeciego obiektu wskazywanego. Podobnie jest w przypadku STL-owego kontenera wskaźników. Wiele algorytmów z biblioteki STL angażuje operacje kopiowania lub przypisania, które działają poprawnie ze wskaźnikami shared_ptr, ale nie działają z unique_ptr.
- Jeśli program nie potrzebuje wielu wskaźników do tego samego obiektu, zupełnie wystarczający będzie wskaźnik unique_ptr. Jest to szczególnie dobry kandydat do roli typu zwracanego z funkcji zwracających obiekty przydzielane na stercie. Obiekty unique_ptr nadają się do składowania w kontenerach STL, pod warunkiem że nie używa się ich z algorytmami takimi jak sort(), które wykonują kopiowanie i przypisywanie elementów do siebie nawzajem.

PRZEKSZTAŁCENIE FUNKCJI

```
void remodel(std::string & str)
{
    std::string *ps = new std::string(str);
    ...
    str = ps;
    delete ps;
    return;
}
```

```
#include <memory>
void remodel(std::string & str)
{
    unique_ptr<std::string> ps (new std::string(str));
    ...
    if(weird_thing())
      throw exception();
    str = *ps;
    return;
}
```

W celu przekszatłcenia funkcji remodel musimy wprowadzić trzy zmiany:

- Dołączyć plik nagłówkowy *memory.*
- Zamienić wskaźniki na obiekt string na obiekty inteligentnych wskaźników wskazujące na obiekt string.
- Usunąć instrukcję delete.

IMPLEMENTACJA STOSU Z WYKORZYSTANIEM SUROWYCH WSKAŹNIKÓW

```cpp
#include <iostream>

const int MAX_SIZE = 100;

template<typename Type>
class Stack {
    Type* tab;
    int _size;
public:
    Stack() :_size(0) {
        tab = new Type[MAX_SIZE];
    }
    ~Stack();
    bool empty();
    Type pop();
    int size();
    void push(Type item);
};

template<typename Type>
bool Stack<Type>::empty() {
    return _size == 0;
}
template<typename Type>
Stack<Type>::~Stack()
{
    //delete tab;
}
template<typename Type>
Type Stack<Type>::pop() {
    if (empty()) {
        std::cerr << "Empty stack!";
        return -1;
    }
    Type tmp = tab[_size - 1];
    _size--;
    return tmp;
}
```

```cpp
template<typename Type>
int Stack<Type>::size() {
    return _size;
}

template<typename Type>
void Stack<Type>::push(Type item) {
    if (_size == MAX_SIZE) {
        std::cerr << "Stack overflow!";
        return;
    }
    _size++;
    tab[_size - 1] = item;
    std::cout << item << " <- pushed on stack"<<std::endl;
}

int main()
{
    Stack<int> stos;
    stos.push(5);
    for (int i = 0; i < 4; i++)
        stos.push(i);
    stos.pop();
    std::cout << stos.size();
    return 0;
}
```

IMPLEMENTACJA STOSU Z WYKORZYSTANIEM INTELIGENTNYCH WSKAŹNIKÓW

```cpp
#include <string>
#include <memory>
#include <iostream>

template<typename Type>
struct node {
    Type data;
    std::unique_ptr<node> next_on_stack;
    ~node() {
        std::cout << "Node deleted: " << data << std::endl;
    }
};

template<typename Type>
class Stack {
    std::unique_ptr<node<Type>> top;
    int _size;
public:
    Stack();
    bool empty();
    Type pop();
    int size();
    void push(Type item);
    void destroy();
};
```

```cpp
template<typename Type>
bool Stack<Type>::empty() {
    return _size == 0;
}
template<typename Type>
Type Stack<Type>::pop() {
    if (!top) {
        std::cout << "Error: Empty stack";
        return -1;
    }
    else {
        top = std::move(top->next_on_stack);
        _size--;
        return top->data;
    }

}

template<typename Type>
int Stack<Type>::size() {
    return _size;
}

template<typename Type>
void Stack<Type>::push(Type item) {
    std::unique_ptr<node<Type>> newTop(new node<Type>);
    newTop->data = item;

    newTop->next_on_stack = std::move(top);
    top = std::move(newTop);

    _size++;
}
```

```cpp
template<typename Type>
void Stack<Type>::destroy()
{
    //usunięcie top spowoduje najpierw usunięcie wskaźnika
    // na node unique_ptr next_on_stack co spowoduje, że ten też
    //usunie najpierw swojego next_on_stack
    //i tak w łatwy sposób zostanie zwolniona cała pamięć
    top.reset();
}
int main()
{
    Stack<int> stos;
    std::cout << stos.empty();
    std::cout << stos.size();
    std::cout << std::endl;
    stos.pop();
    stos.push(5);
    stos.push(1);
    stos.push(2);
    std::cout << std::endl;
    stos.destroy();
    std::cout << stos.empty();
    std::cout << stos.size();
    /*stos.push(5);
    for (int i = 0; i < 4; i++)
        stos.push(i);
    stos.pop();
    std::cout << stos.size();*/
    return 0;
}
```