Podstawy programowania: Laboratorium nr 9
Wskaźniki.

2017-2018

mgr inż. Przemysław Walkowiak
dr inż. Michał Ciesielczyk

Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania): `std::cout << "Zadanie X"<< std::endl;`,
4. umieszczeniu wszystkich rozwiązań w jednym pliku, chyba, że w poleceniu napisano inaczej.
5. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie).

Wprowadzenie

W języku C++ wskaźnik jest to zmienna, która przechowuje adres pod którym znajduje się inna zmienna/obiekt. Umożliwia się w ten sposób pośredni dostęp do jakiegoś zasobu, podobnie jak poprzez referencję. Aby zadeklarować zmienną typu wskaźnikowego należy dodać do typu zmiennej modyfikator w postaci symbolu '*'. Np.:

```
int *px = nullptr; // zmienna wskaźnikowa zaincjalizowana
                   // pustym adresem
double *py = nullptr;
```

Aby przypisać zmiennej typu wskaźnikowego wartość, tj. adres innej zmiennej, należy posłużyć się operatorem pobrania adresu '&'. Np.:

```
int x = 15;
double  y = 3.14;
px = &x;
py = &y
```

Teraz zmienne `px` oraz `py` przechowują adresy w pamięci, gdzie znajdują się zmienne `x` oraz `y`. Aby wyciągnąć wartość pod danym adresem (który możemy przechowywać w zmiennej wskaźnikowej) posługujemy się operatorem wyłuskania '*'.

```
std::cout << *px << *py; // na ekranie zostanie wyświetlone: 15 3.14
int z = *px;     // z ma wartość 15
```

Zauważ, że symbol '*' ma trzy różne znaczenia w zależności od kontekstu. Gdy występuje w deklaracje przy typie, pełni rolę modyfikatora typu. Gdy występuje obok nazwy zmiennej w wyrażeniu, działa jako operator wyłuskania. Gdy otoczymy go wartościami z dwóch stron, pełni rolę operator dwuargumentowego mnożenia.

Wskaźniki, podobnie jak referencje umożliwiają pośredni dostęp do jakiegoś zasobu, z tą różnicą, że w przypadku referencji mamy ZAWSZE pewność, że zmienna ma wartość, referencje muszą być zainicjalizowane w czasie deklaracji. W przypadku wskaźników, w czasie deklaracji powinno się nadawać im wartość `nullptr`, oznaczającą, że zmienna na nic nie wskazuje. A przed odwołaniem się do elementu wskazywanego należy się wpierw upewnić, że jest on różny od `nullptr`, nabrać pewności, że ma poprawną wartość.

Tego typu wskaźniki. tzw. surowe wskaźniki, wykorzystuje się, gdy chcemy przekazać programiście, że wskazywany obiekt do niego nie należy. Czyli przechowuje tylko wskazanie, uchwyt, lokalizację zasobu.

Gdy chcemy utworzyć nowy obiekt, a nie wskazywać na już istniejący, możemy się posłużyć tzw. wskaźnikami inteligentnymi (ang. smart pointers). Oprócz samego przechowywania adresu, gdzie znajduje się obiekt, zarządzają jednocześnie jego czasem życia, są "właścicielami" obiektu wskazywanego. Gwarantują równocześnie, że w momencie jak skończy się czas życia samego wskaźnika, to obiekt wskazywany również zostanie usunięty.

std::unique_ptr

Pierwszym, najprostszym wskaźnikiem jest `std::unique_ptr`. Reprezentuje on pełną kontrolę nad czasem życia wskazywanego obiektu. W momencie jak skończy się zasięg widoczności `std::unique_ptr` obiekt wskazywany zostanie od razu zniszczony. Oprócz samego typu wskaźnikowego, istnieje jeszcze funkcja pomocnicza `make_unique`, która tworzy dynamicznie obiekt i zwraca wskaźnik. Np.:

```
struct Person {
        std::string  firstname;
        std::string lastname;
        int age;
};

{
    std::unique_ptr<int> pi = std::make_unique<int>(); // wartość wskazywana
                                        // zostanie zainicjalizowana 0

    std::unique_ptr<Person> person = std::make_unique<Person>("Jan",
                                                        "Kowalski", 42);
    auto person2 = std::make_unique<Person>("Adam", "Nowak", 13);

    std::cout << person->firstname << person->lastname; // "Jan Kowalski";
    std::cout << person2->firstname << person2->lastname; // Ądam Nowak";
```

```
} // oba obiekty wskazywane przez pi, person, person2 zostaną
  // w tym miejscu usunięte
```

Wskaźniki `std::unique_ptr` można również przekazywać np. do funkcji. Jednakże należy pamiętać, że w tym przypadku może istnieć TYLKO jeden właściciel wskazywanego obiektu.

```
    void fun1(Person *p) {
        // funkcja ma dostęp do obiektu wskazywanego przez 'p',
        // ale nie jest jej właścicielem
        std::cout << p->firstname;
    }
    {
        auto person = std::make_unique<Person>("Jan", "Kowalski", 15);
        fun1(person.get()); // za pomocą funkcji 'get()' pobieramy
                            // surowy wskaźnik.
        std::cout << person->firstname; // wyświetli "Jan"
    } // niszczony jest obiekt wskazywany przez person
```

```
    void fun2(std::unique_ptr<Person> &p) {
        // przekazujemy wskaźnik na obiekt przez referencję.
        // "Nie kradniemy" obiektu.
        std::cout << p->firstname;
    }
    {
        auto person = std::make_unique<Person>("Jan", "Kowalski", 15);
        fun2(person);
        std::cout << person->firstname; // wyświetli "Jan"
    } // niszczony jest obiekt wskazywany przez person
```

```
    void fun3(std::unique_ptr<Person> p) {
        // przekazujemy wskaźnik na obiekt. "Kradniemy prawa własności".
        std::cout << p->firstname;
    } // przekazany obiekt jest niszczony
    {
        auto person = std::make_unique<Person>("Jan", "Kowalski", 15);
        fun3(person);  // błąd, w ten sposób nie można
                       // "przekazać"własności.
        fun3(std::move(person));  // ok, przekazujemy wskaźniki
                                  // i wszystkie prawa do funkcji.
        std::cout << person->firstname; // błąd, nie mamy już tutaj
                                        // 'person', przenieśliśmy go.
    } // person został przekazany do funkcji, ona go niszczy.
```

std::shared_ptr

`std::shared_ptr` działa odmiennie od `std::unique_ptr` pod względem czasu życia obiektów. W tym przypadku obiekt jest niszczony wtedy i tylko wtedy, gdy WSZYSTKIE kopie wskaźnika `std::shared_ptr` na dany obiekt zostaną zniszczone. Do tworzenia tego typu wskaźnika można użyć funkcji `std::make_shared`.

```
void fun1(Person *p) {
    // funkcja ma dostęp do obiektu wskazywanego przez 'p',
    // ale nie jest jej właścicielem
    std::cout << p->firstname;
}
{
    auto person = std::make_shared<Person>("Jan", "Kowalski", 15);
    fun1(person.get()); // za pomocą funkcji 'get()' pobieramy
                        // surowy wskaźnik.
    std::cout << person->firstname; // wyświetli "Jan"
} // niszczony jest obiekt wskazywany przez person
```

```
void fun2(std::shared_ptr<Person> &p) {
    // przekazujemy wskaźnik na obiekt przez referencję.
    std::cout << p->firstname;
}
{
    auto person = std::make_shared<Person>("Jan", "Kowalski", 15);
    fun2(person);
    std::cout << person->firstname; // wyświetli "Jan"
} // niszczony jest obiekt wskazywany przez person
```

```
void fun3(std::shared_ptr<Person> p) {
    // przekazujemy wskaźnik na obiekt. Kopiujemy prawa własności.
    std::cout << p->firstname;
}
{
    auto person = std::make_shared<Person>("Jan", "Kowalski", 15);
    fun3(person);  // ok, robimy kopię wskaźnika
    std::cout << person->firstname; // błąd, nie mamy już tutaj
                                    // 'person', przenieśliśmy go.
} // person zostanie zniszczony, gdy oba wskaźniki stracą zasięg
```

```
void fun4(std::shared_ptr<Person> p) {
    std::cout << p->firstname;
} // przekazany obiekt jest niszczony (to jest ostatni właściciel)
{
    auto person = std::make_shared<Person>("Jan", "Kowalski", 15);
    fun3(std::move(person));  // ok, przekazujemy wskaźniki
                              // i wszystkie prawa do funkcji.
    std::cout << person->firstname; // błąd, nie mamy już tutaj
```

```
                                    // 'person', przenieśliśmy go.
10  } // person został przekazany do funkcji, ona go niszczy.
```

Podsumowanie

Gdy chcemy pisać ładny i czytelny kod, w ogólności powinniśmy unikać używania wskaźników. Należy się zastanowić wpierw, czy problem z jakim mamy do czynienia nie będzie możliwy do rozwiązania za pomocą podstawowych typów i struktur danych, czy też moze kontenerów w stylu `std::vector`,`std::array`.

Jeżeli natomiast potrzebujemy obiektów zaalokowanych dynamicznie, na stercie, wpierw powinniśmy rozważyć wykorzystanie `std::unique_ptr`, gdyż dają one największą kontrolę nad tym, kto jest odpowiedzialny za zwolnienie zasobu.

W przypadku, gdy wiemy, że do danego obiektu odwoływać się będziemy w wielu miejscach, i prawa "własności" nie są trywialne do rozstrzygnięcia, dopiero wtedy powinno się używać wskaźników typu `std::shared_ptr`.

Zadania

Zadanie 1

Zdefiniuj strukturę `BstNode` reprezentującą pojedynczy węzeł binarnego drzewa poszukiwań (BST) przechowującego liczby całkowite (`int`). Każdy węzeł powinien składać się z następujących składowych:

a) przechowywanej wartości (`value`),
b) wskaźnika na lewe poddrzewo (`left`),
c) wskaźnika na prawe poddrzewo (`right`).

Do przechowywania wskaźników wykorzystaj typ `std::unique_ptr`. Zaimplementuj konstruktor jednoargumentowy – `BstNode(int value)` – inicjalizujący wszystkie pola struktury.

Wszystkie deklaracje umieść w pliku nagłówkowym *bst.hpp*, natomiast całą implementację w pliku *bst.cpp*.

Wskazówka Domyślnie oba poddrzewa każdego węzła powinny być puste, tzn. wskaźniki powinny być zainicjalizowane `nullptr`.

Zadanie 2

Przeciąż operator wstawienia w taki sposób, aby możliwe było wyświetlanie całej zawartości drzewa binarnego w kolejności *in-order* w następujący sposób:

```
std::cout << bst.get();
```

gdzie `bst` to zmienna typu `std::unique_ptr<BstNode>` (korzeń drzewa BST). Zwróć uwagę, że najprawdopodobniej będziesz musiał skorzystać z funkcji `get()` do pobrania surowego wskaźnika (`BstNode*`).

Wskazówka Zaimplementuj funkcję:

```
std::ostream& operator << (std::ostream& stream, BstNode* bst)
```

Dodatkowe informacje:

- Sposoby przechodzenia drzewa binarnego

Zadanie 3

Zdefiniuj funkcję `addToTree` w taki umożliwiała ona dodawanie nowych liczb do drzewa BST w następujący sposób:

```
addToTree(bst, 5);
addToTree(bst, 2);
addToTree(bst, 8);
```

gdzie `bst` to zmienna typu `std::unique_ptr<BstNode>`, reprezentująca korzeń drzewa. Drzewo powinno przechowywać wyłącznie unikalne wartości.

Przetestuj swoją implementację dodając do drzewa kilka przykładowych liczb, a następnie wyświetlając jego zawartość na ekranie. W jakiej kolejności wyświetlane są wszystkie liczby?

Wskazówka 1 Zaimplementuj funkcję:

```
void addToTree(std::unique_ptr<BstNode>& bst, int value)
```

Wskazówka 2 Możesz zaimplementować dodawanie nowych elementów do drzewa w wersji rekurencyjnej.

Zadanie 4

Przeciąż operator wstawienia w taki sposób, aby możliwe było dodawanie nowych liczb do drzewa BST w następujący sposób:

```
bst << 5 << 2 << 8;
```

gdzie `bst` to zmienna typu `std::unique_ptr<BstNode>`, reprezentująca korzeń drzewa.

Przetestuj swoją implementację dodając do drzewa kilka przykładowych liczb, a następnie wyświetlając jego zawartość na ekranie.

Wskazówka 1 Zaimplementuj funkcję:

```
std::unique_ptr<BstNode>& operator << (std::unique_ptr<BstNode>& bst,
                                       int value)
```

Wskazówka 2 Możesz wykorzystać implementację funkcji `addToTree` z poprzedniego zadania.

Zadanie 5

Zaimplementuj następujące funkcje:

a) `int minimum(BstNode* root)` – zwracającą najmniejszą wartość przechowywaną na drzewie, oraz

b) `int maximum(BstNode* root)` – zwracającą największą wartość przechowywaną na drzewie.

Przetestuj swoją implementację.

Wskazówka Wartość minimalna oraz maksymalna znajdują się odpowiednio w skrajnym lewym oraz skrajnym prawym liściu drzewa.

Zadanie 6

Zaimplementuj funkcję **bool** `contains(BstNode* root, int value)` sprawdzającą czy na drzewie je podana wartość. Funkcja powinna zwracać prawdę jeśli dany element istnieje, fałsz w przeciwnym wypadku. Implementacje odpowiednio przetestuj.

Dodatkowe informacje:

- Wyszukiwanie elementów na drzewie

Zadanie 7

Zaimplementuj funkcję **unsigned int** `size(BstNode* root)` zliczającą elementy znajdujące się na drzewie. Implementacje odpowiednio przetestuj.

Na następne zajęcia

- Szablony: `http://www.cplusplus.com/doc/tutorial/templates/`