Krótkie przypomnienie mechanizmu szablonów i programowania obiektowego w C++ na przykładzie listy jednokierunkowej

Minisamouczek dla czujących się niepewnie w programowaniu w C++

Jacek Klimaszewski

Treść zadania

Dokonaj implementacji listy jednokierunkowej, korzystając z enkapsulacji (hermetyzacji) i mechanizmu szablonów.

Treść zadania

Dokonaj implementacji listy jednokierunkowej, korzystając z enkapsulacji (hermetyzacji) i mechanizmu szablonów.

- Klasa szablonowa opisująca węzeł listy ma zawierać pole reprezentujące dane (zgodne z typem szablonowym) i wskaźnik na następny element.

Treść zadania

Dokonaj implementacji listy jednokierunkowej, korzystając z enkapsulacji (hermetyzacji) i mechanizmu szablonów.

- Klasa szablonowa opisująca węzeł listy ma zawierać pole reprezentujące dane (zgodne z typem szablonowym) i wskaźnik na następny element.
- Klasa (szablonowa) opisująca listę powinna zawierać wskaźnik na głowę (pierwszy element listy) oraz długość listy.

Treść zadania

Dokonaj implementacji listy jednokierunkowej, korzystając z enkapsulacji (hermetyzacji) i mechanizmu szablonów.

- Klasa szablonowa opisująca węzeł listy ma zawierać pole reprezentujące dane (zgodne z typem szablonowym) i wskaźnik na następny element.
- Klasa (szablonowa) opisująca listę powinna zawierać wskaźnik na głowę (pierwszy element listy) oraz długość listy.
- Sama klasa opisująca węzeł może być klasą zagnieżdżoną.

Treść zadania

Dokonaj implementacji listy jednokierunkowej, korzystając z enkapsulacji (hermetyzacji) i mechanizmu szablonów.

- Klasa szablonowa opisująca węzeł listy ma zawierać pole reprezentujące dane (zgodne z typem szablonowym) i wskaźnik na następny element.
- Klasa (szablonowa) opisująca listę powinna zawierać wskaźnik na głowę (pierwszy element listy) oraz długość listy.
- Sama klasa opisująca węzeł może być klasą zagnieżdżoną.

Uproszczona implementacja ma umożliwiać dodawanie elementu na początek listy, wyszukiwanie elementu i pobieranie długości listy.

Treść zadania

Dokonaj implementacji listy jednokierunkowej, korzystając z enkapsulacji (hermetyzacji) i mechanizmu szablonów.

- Klasa szablonowa opisująca węzeł listy ma zawierać pole reprezentujące dane (zgodne z typem szablonowym) i wskaźnik na następny element.
- Klasa (szablonowa) opisująca listę powinna zawierać wskaźnik na głowę (pierwszy element listy) oraz długość listy.
- Sama klasa opisująca węzeł może być klasą zagnieżdżoną.

Uproszczona implementacja ma umożliwiać dodawanie elementu na początek listy, wyszukiwanie elementu i pobieranie długości listy.

Prezentację przewidziano w formie samouczka — najpierw pojawia się zagadnienie, które należy samodzielnie rozwiązać, a następnie skonfrontować własne rozwiązanie z podanym.

Wstępny zarys

Przyjmijmy wariant z klasą zagnieżdżoną — zacznijmy od szkicu naszych klas (proszę się wpierw zastanowić, jak to zrobić):

Wstępny zarys

Przyjmijmy wariant z klasą zagnieżdżoną — zacznijmy od szkicu naszych klas (proszę się wpierw zastanowić, jak to zrobić):

```
template<typename T>
class lista_jednokierunkowa
{
public:
    struct wezel_listy
    {
        T dane;
        wezel_listy *nast;
    };

private:
    wezel_listy *glowa;
    unsigned dlugosc;
};
```

Krótkie omówienie szkicu

- Klasa **lista_jednokierunkowa** jest klasą szablonową — w szablonie określamy typ obiektów, które będziemy trzymać w liście.

Krótkie omówienie szkicu

- Klasa **lista_jednokierunkowa** jest klasą szablonową — w szablonie określamy typ obiektów, które będziemy trzymać w liście.
- Głowa i długość listy są składnikami prywatnymi — użytkownik nie powinien znać szczegółów implementacji, do działania wystarczy mu interfejs.

Krótkie omówienie szkicu

- Klasa **lista_jednokierunkowa** jest klasą szablonową — w szablonie określamy typ obiektów, które będziemy trzymać w liście.
- Głowa i długość listy są składnikami prywatnymi — użytkownik nie powinien znać szczegółów implementacji, do działania wystarczy mu interfejs.
- Do składników węzła listy można swobodnie sięgać — elementy struktury mają z definicji dostęp publiczny.

Krótkie omówienie szkicu

- Klasa **lista_jednokierunkowa** jest klasą szablonową — w szablonie określamy typ obiektów, które będziemy trzymać w liście.
- Głowa i długość listy są składnikami prywatnymi — użytkownik nie powinien znać szczegółów implementacji, do działania wystarczy mu interfejs.
- Do składników węzła listy można swobodnie sięgać — elementy struktury mają z definicji dostęp publiczny.
- Klasa definiująca węzeł listy również znajduje się w sekcji **public** klasy definiującej listę, ażeby można było się do niej odwołać z zewnątrz.

Zarządzanie pamięcią węzła listy

Ażeby ułatwić sobie zarządzanie pamięcią, stwórzmy konstruktor(y) i destruktor w klasie **wezel_listy**:

Zarządzanie pamięcią węzła listy

Ażeby ułatwić sobie zarządzanie pamięcią, stwórzmy konstruktor(y) i destruktor w klasie **wezel_listy**:

```
wezel_listy(const T& dane, wezel_listy *nast)
    : dane(dane), nast(nast)
{
}

wezel_listy(T&& dane, wezel_listy *nast)
    : dane(std::move(dane)), nast(nast)
{
}

~wezel_listy()
{
    delete nast;
    nast = nullptr;
}
```

Krótkie omówienie zarządzania pamięcią węzła

- Pierwszy konstruktor umożliwia tworzenie nowego węzła, kopiuje dane i ustawia wskaźnik na kolejny element listy.

Krótkie omówienie zarządzania pamięcią węzła

- Pierwszy konstruktor umożliwia tworzenie nowego węzła, kopiuje dane i ustawia wskaźnik na kolejny element listy.
- Drugi konstruktor zachowuje się podobnie z tą różnicą, że nie kopiuje danych, tylko je przenosi (ang. *move semantics*).

Krótkie omówienie zarządzania pamięcią węzła

- Pierwszy konstruktor umożliwia tworzenie nowego węzła, kopiuje dane i ustawia wskaźnik na kolejny element listy.
- Drugi konstruktor zachowuje się podobnie z tą różnicą, że nie kopiuje danych, tylko je przenosi (ang. *move semantics*).
- Destruktor niszczy dodatkowo następny element.

Krótkie omówienie zarządzania pamięcią węzła

- Pierwszy konstruktor umożliwia tworzenie nowego węzła, kopiuje dane i ustawia wskaźnik na kolejny element listy.
- Drugi konstruktor zachowuje się podobnie z tą różnicą, że nie kopiuje danych, tylko je przenosi (ang. *move semantics*).
- Destruktor niszczy dodatkowo następny element.
- Ponoć dobrą praktyką jest zerowanie wskaźników po zwolnieniu pamięci (unikniemy w ten sposób podwójnego zwalniania pamięci).

Zarządzanie pamięcią listy jednokierunkowej

Stwórzmy teraz konstruktor domyślny i destruktor:

Zarządzanie pamięcią listy jednokierunkowej

Stwórzmy teraz konstruktor domyślny i destruktor:

```
lista_jednokierunkowa() noexcept
    : glowa(nullptr), dlugosc(0)
{
}
```

```
lista_jednokierunkowa(lista_jednokierunkowa &&lst) noexcept // konstruktor przenoszący
    : glowa(lst.glowa), dlugosc(lst.dlugosc)
{
    lst.glowa = nullptr;
    lst.dlugosc = 0;
}
```

```
~lista_jednokierunkowa()
{
    delete glowa;
    glowa = nullptr;
    dlugosc = 0;
}
```

Powyższe i następne metody powinny znaleźć się w sekcji **public**.

Krótkie omówienie zarządzania pamięcią listy

- Konstruktor wzbogacono dodatkowo słowem kluczowym **noexcept**, ponieważ konstruktor gwarantuje, że nie pojawi się jakikolwiek wyjątek w trakcie jego działania — pozwala to kompilatorowi wygenerować efektywniejszy kod, bo nie musi się przygotowywać na potencjalne pojawienie się wyjątku.

Krótkie omówienie zarządzania pamięcią listy

- Konstruktor wzbogacono dodatkowo słowem kluczowym **noexcept**, ponieważ konstruktor gwarantuje, że nie pojawi się jakikolwiek wyjątek w trakcie jego działania — pozwala to kompilatorowi wygenerować efektywniejszy kod, bo nie musi się przygotowywać na potencjalne pojawienie się wyjątku.
- Korzystając z destruktora klasy **wezel_listy** wystarczy jedynie zniszczyć głowę listy, żeby zniszczyć całą listę.

Krótkie omówienie zarządzania pamięcią listy

- Konstruktor wzbogacono dodatkowo słowem kluczowym **noexcept**, ponieważ konstruktor gwarantuje, że nie pojawi się jakikolwiek wyjątek w trakcie jego działania — pozwala to kompilatorowi wygenerować efektywniejszy kod, bo nie musi się przygotowywać na potencjalne pojawienie się wyjątku.
- Korzystając z destruktora klasy **wezel_listy** wystarczy jedynie zniszczyć głowę listy, żeby zniszczyć całą listę. **UWAGA**: zniszczenie głowy spowoduje rekurencyjne wywołanie destruktora następnego elementu — w przypadku bardzo długiej listy może dojść do **przepełnienia stosu** (w przypadku zrównoważonych drzew można pozwolić sobie na tę rekurencję). Proszę przekształcić destruktor(y) tak, aby korzystał(y) z iteracji zamiast rekurencji.

Pobieranie długości listy

Zacznijmy od najprostszej operacji, czyli zwracania długości listy:

Pobieranie długości listy

Zacznijmy od najprostszej operacji, czyli zwracania długości listy:

```
unsigned pobierz_dlugosc() const noexcept
{
    return dlugosc;
}
```

Pobieranie długości listy

Zacznijmy od najprostszej operacji, czyli zwracania długości listy:

```
unsigned pobierz_dlugosc() const noexcept
{
    return dlugosc;
}
```

Metodę opatrzono dodatkowo słowem kluczowym **const**, ponieważ pobranie długości nie modyfikuje listy — dzięki temu możemy wywołać tę metodę dla obiektu stałego.

Operacja dodawania na początek listy

Napiszmy teraz operację dodawania elementu na początek listy:

Operacja dodawania na początek listy

Napiszmy teraz operację dodawania elementu na początek listy:

```
void wstaw_na_poczatek(const T& dane)
{
    auto nowa_glowa = new wezel_listy(dane, glowa);
    glowa = nowa_glowa;
    // zamiast dwóch linii można krócej (ale czy czytelniej?):
//  glowa = new wezel_listy(dane, glowa);
    ++dlugosc;
}
```

Operacja dodawania na początek listy

Napiszmy teraz operację dodawania elementu na początek listy:

```
void wstaw_na_poczatek(const T& dane)
{
    auto nowa_glowa = new wezel_listy(dane, glowa);
    glowa = nowa_glowa;
    // zamiast dwóch linii można krócej (ale czy czytelniej?):
//  glowa = new wezel_listy(dane, glowa);
    ++dlugosc;
}
```

W ramach treningu możesz dodać przeładowaną wersję powyższej metody, która przyjmowałaby argument typu `T&&`.

Wyszukiwanie z użyciem komparatora

Metoda wyszukująca element w liście korzystająca z komparatora może wyglądać następująco:

Wyszukiwanie z użyciem komparatora

Metoda wyszukująca element w liście korzystająca z komparatora może wyglądać następująco:

```
template<typename Comp>
wezel_listy* znajdz_element(const T& el, Comp comp) const
{
    for (auto p = glowa; p; p = p->nast)
        if (comp(el, p->dane))
            return p;
    return nullptr;
}
```

Wyszukiwanie z użyciem komparatora

Metoda wyszukująca element w liście korzystająca z komparatora może wyglądać następująco:

```
template<typename Comp>
wezel_listy* znajdz_element(const T& el, Comp comp) const
{
    for (auto p = glowa; p; p = p->nast)
        if (comp(el, p->dane))
            return p;
    return nullptr;
}
```

Komparator jest funkcją (lub obiektem funkcyjnym) przyjmującą dwa obiekty i zwracającą (w szczególności) prawdę, gdy testowane obiekty są sobie równe.

Przykład użycia komparatora w postaci funkcji i obiektu funkcyjnego (wyrażenia *lambda*)

```
struct some_class
{
    int some_int;
};

bool f_komp(const some_class &a, const some_class &b)
{
    return a.some_int == b.some_int;
}

int main()
{
    auto komp = [](const some_class &a, const some_class &b) {
        return a.some_int == b.some_int;
    };
    lista_jednokierunkowa<some_class> lista;
    for (int i = 0; i < 10; i++)
        lista.wstaw_na_poczatek(some_class{i});
    lista.znajdz_element(some_class{2}, komp);
// lista.znajdz_element(some_class{2}, f_komp);
    return 0;
}
```

Słówko o liczbach pseudolosowych (nowym sposobem)

Przejdźmy od razu do przykładu:

```
#include <iostream>
#include <random>
#include <ctime>

int main()
{
    // ustawienie ziarna (by wyniki się nie powtarzały)
    std::default_random_engine gen(time(nullptr));

    // rozkład jednostajny liczb całkowitych od -5 do 5
    std::uniform_int_distribution<int> dist(-5, 5);

    for (int i = 0; i < 10; i++)
        std::cout << dist(gen) << std::endl;
    return 0;
}
```

Słówko o pomiarze czasu (nowym sposobem)

Przejdźmy od razu do przykładu:

```
#include <iostream>
#include <random>
#include <chrono>
#include <ctime>
// using namespace std; using namespace std::chrono;
using std::chrono::duration_cast, std::chrono::milliseconds;

// ...

int main()
{
    std::default_random_engine gen(time(nullptr));
    std::uniform_int_distribution<int> dist(-50000, 50000);

    auto komp = [](const some_class &a, const some_class &b) {
        return a.some_int < b.some_int;
    };

    lista_jednokierunkowa<some_class> lista;

    auto t1 = std::chrono::steady_clock::now(); // znacznik początku pomiaru
    for (int i = 0; i < 20000; i++)
        lista.wstaw_porzadek(some_class{dist(gen)}, komp); // wstawienie z zachowaniem porządku
    auto t2 = std::chrono::steady_clock::now(); // znacznik końca pomiaru

    std::cout << "Czas␣(ms):␣" << duration_cast<milliseconds>(t2-t1).count() << std::endl;
    return 0;
}
```

Krótko o strumieniu do napisu

Niemal na każdym ćwiczeniu będą musieli Państwo napisać metodę, która zwróci reprezentację napisową struktury danych. Można to bardzo łatwo zrealizować za pomocą przeładowania operatora strumieniowego (zdefiniowanego poza klasą!). Zobaczmy, jak to wygląda na przykładzie klasy **some_class**:

Krótko o strumieniu do napisu

Niemal na każdym ćwiczeniu będą musieli Państwo napisać metodę, która zwróci reprezentację napisową struktury danych. Można to bardzo łatwo zrealizować za pomocą przeładowania operatora strumieniowego (zdefiniowanego poza klasą!). Zobaczmy, jak to wygląda na przykładzie klasy **some_class**:

```
std::ostream& operator<< (std::ostream &out, const some_class &obj)
{
    out << obj.some_int;
    return out;
}
```

Krótko o strumieniu do napisu

Niemal na każdym ćwiczeniu będą musieli Państwo napisać metodę, która zwróci reprezentację napisową struktury danych. Można to bardzo łatwo zrealizować za pomocą przeładowania operatora strumieniowego (zdefiniowanego poza klasą!). Zobaczmy, jak to wygląda na przykładzie klasy **some_class**:

```
std::ostream& operator<< (std::ostream &out, const some_class &obj)
{
    out << obj.some_int;
    return out;
}
```

Dodajmy teraz metodę `to_string` do naszej klasy reprezentującej listę jednokierunkową:

Krótko o strumieniu do napisu

Niemal na każdym ćwiczeniu będą musieli Państwo napisać metodę, która zwróci reprezentację napisową struktury danych. Można to bardzo łatwo zrealizować za pomocą przeładowania operatora strumieniowego (zdefiniowanego poza klasą!). Zobaczmy, jak to wygląda na przykładzie klasy **some_class**:

```
std::ostream& operator<< (std::ostream &out, const some_class &obj)
{
    out << obj.some_int;
    return out;
}
```

Dodajmy teraz metodę `to_string` do naszej klasy reprezentującej listę jednokierunkową:

```
std::string to_string() const
{
    if (dlugosc == 0)
        return "[]";
    std::ostringstream str; // pamiętaj o #include <sstream>
    str << "[" << glowa->dane;
    for (auto p = glowa->nast; p; p = p->nast)
        str << ",␣" << p->dane;
    str << "]";
    return str.str(); // wydobycie napisu ze strumienia
}
```

Krótko o sprawdzaniu szczelności

`valgrind` to linuksowe narzędzie wykrywające wycieki pamięci (w VC++ można użyć biblioteki CRT). Po zbudowaniu programu komendą `g++ -Wall -std=c++1z -O2 lista.cpp` możemy zbadać wycieki poleceniem `valgrind a.out`. Gdy program się zakończy, pojawi się podsumowanie, np.:

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 14 allocs, 14 frees, 74,473 bytes allocated

All heap blocks were freed -- no leaks are possible
```

które informuje o braku wycieków. Gdyby jakieś były, komunikat mógłby brzmieć tak:

```
HEAP SUMMARY:
    in use at exit: 16 bytes in 1 blocks
  total heap usage: 14 allocs, 13 frees, 74,475 bytes allocated

LEAK SUMMARY:
   definitely lost: 16 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory
```

Krótko o sytuacjach wyjątkowych

Czasami jakaś operacja na strukturze danych okaże się niedozwolona (np. usunięcie głowy w pustej liście) — warto przewidzieć takie przypadki i odpowiednio zasygnalizować użytkownikowi błąd (jeśli prowadziłby do wykrzaczenia programu). Można np. wyprowadzić stosowny komunikat do strumienia błędu (`std::cerr`) i zakończyć program wywołaniem funkcji `abort()` (z pliku nagłówkowego `<cstdlib>`) bądź rzucić odpowiedni wyjątek (`https://en.cppreference.com/w/cpp/error/exception`).

Krótko o sytuacjach wyjątkowych

Czasami jakaś operacja na strukturze danych okaże się niedozwolona (np. usunięcie głowy w pustej liście) — warto przewidzieć takie przypadki i odpowiednio zasygnalizować użytkownikowi błąd (jeśli prowadziłby do wykrzaczenia programu). Można np. wyprowadzić stosowny komunikat do strumienia błędu (`std::cerr`) i zakończyć program wywołaniem funkcji `abort()` (z pliku nagłówkowego `<cstdlib>`) bądź rzucić odpowiedni wyjątek (`https://en.cppreference.com/w/cpp/error/exception`).

```
void usun_glowe()
{
    if (glowa == nullptr)
        throw std::domain_error("Pusta lista!");
    auto nowa_glowa = glowa->nast;
    glowa->nast = nullptr; // zbędne, jeśli brak rekurencji
    delete glowa;
    glowa = nowa_glowa;
    --dlugosc;
}
```

Podsumowanie

Znają już Państwo konstrukcje języka C++, których będziecie potrzebowali do realizacji ćwiczeń. Kod, który powstał w ramach tego samouczka możecie Państwo dalej rozbudowywać na potrzeby pierwszego laboratorium, to jest listy dwukierunkowej (konspekty do zajęć są pod adresem `http://wikizmsi.zut.edu.pl/wiki/A2/L`). W razie jakichkolwiek trudności podczas realizacji ćwiczeń, proszę śmiało zadawać pytania.
Zachęcam również do zmierzenia się z zadaniami ze strony `https://www.hackerrank.com/domains/data-structures?filters%5Bsubdomains%5D%5B%5D=linked-lists`, szczególnie tych z Państwa, którzy nadal nie czują się swobodnie z programowania.

Dodatek — krótkie wprowadzenie do BST

- Węzeł listy zawierał wskaźnik na następny element, z kolei węzeł binarnego drzewa poszukiwań zawierać będzie wskaźnik na węzeł rodzicielski, wskaźnik na lewego potomka i wskaźnik na prawego potomka.

Dodatek — krótkie wprowadzenie do BST

- Węzeł listy zawierał wskaźnik na następny element, z kolei węzeł binarnego drzewa poszukiwań zawierać będzie wskaźnik na węzeł rodzicielski, wskaźnik na lewego potomka i wskaźnik na prawego potomka.
- Klasa reprezentująca binarne drzewo poszukiwań zawierać będzie wskaźnik na korzeń drzewa (czyli pierwszy element) oraz liczbę węzłów w drzewie.

Dodatek — krótkie wprowadzenie do BST

- Węzeł listy zawierał wskaźnik na następny element, z kolei węzeł binarnego drzewa poszukiwań zawierać będzie wskaźnik na węzeł rodzicielski, wskaźnik na lewego potomka i wskaźnik na prawego potomka.
- Klasa reprezentująca binarne drzewo poszukiwań zawierać będzie wskaźnik na korzeń drzewa (czyli pierwszy element) oraz liczbę węzłów w drzewie.
- Duplikatów proszę nie trzymać w drzewie.

Dodatek — krótkie wprowadzenie do BST

- Metoda wyszukująca element w drzewie korzystająca z komparatora może wyglądać następująco (komparator musi definiować operator mniejszości(!)):

```
template<typename Comp>
wezel_drzewa* znajdz_element(const T& el, Comp comp) const
{
    auto p = korzen;
    while (p)
    {
        if (comp(el, p->dane))       // czyli "el < p->dane"
            p = p->lewy;
        else if (comp(p->dane, el)) // czyli "el > p->dane"
            p = p->prawy;
        else                         // czyli "el == p->dane"
            return p;
    }
    return nullptr; // albo "return p;"
}
```

Dodatek — krótkie wprowadzenie do BST

- Metoda wyszukująca element w drzewie korzystająca z komparatora może wyglądać następująco (komparator musi definiować operator mniejszości(!)):

```
template<typename Comp>
wezel_drzewa* znajdz_element(const T& el, Comp comp) const
{
    auto p = korzen;
    while (p)
    {
        if (comp(el, p->dane))       // czyli "el < p->dane"
            p = p->lewy;
        else if (comp(p->dane, el)) // czyli "el > p->dane"
            p = p->prawy;
        else                         // czyli "el == p->dane"
            return p;
    }
    return nullptr; // albo "return p;"
}
```

- Więcej na `https://jklimaszewski.zut.edu.pl/a2`