Laboratorium VI

Wstęp

Dzisiejsze zajęcia dotyczyły będą algorytmów biblioteki STL. Zanim przejdziemy do ich opisu, podsumujmy pokrótce poznane dotychczas zagadnienia, które pozwolą nam na zrozumienie zasady ich działania:

- szablony funkcji - wszystkie algorytmy dostępne w STL to szablony funkcji
- kontenery i iteratory - algorytmy operują na zakresach elementów kontenerów zdefiniowanych przez parę iteratorów
- przeciążanie operatorów - działanie algorytmów możemy dodatkowo modyfikować podając do nich funktory, tzn. obiekty posiadające przeciążenie operatora `( )`.

Biblioteka standardowa oferuje w nagłówkach `algorithm` i `numeric` szeroki wachlarz różnych algorytmów. Zwięzły przegląd algorytmów dostępnych w STL można zobaczyć np. tutaj. Celem zajęć nie jest jednak omówienie ich wszystkich, lecz zaprezentowanie ogólnej idei ich wykorzystania. Nadzieją autora jest, aby po wykonaniu instrukcji do poznania działania nowego algorytmu wystarczyło czytelnikowi jedynie szybkie zerknięcie do odpowiedniej referencji. Z tego powodu nie będziemy też tłumaczyć szczegółów działania poszczególnych algorytmów, lecz odsyłamy czytelnika do dokumentacji.

Z algorytmów STL warto korzystać z następujących 3 powodów:

- Uzyskujemy dostęp do wydajnych implementacji dużej liczby algorytmów. Najlepszym przykładem jest tutaj sortowanie. Jest to jeden z najstarszych problemów w informatyce, którego wydajne rozwiązanie stanowi jednak temat badań po dzień dzisiejszy (bardzo ciekawy wykład na ten temat można znaleźć tutaj).
- Skracamy kod oraz czas jego pisania. Często zawiłe pętle for można zamienić na 1 elegancką linijkę.
- Zwiększenie ekspresyjności kodu. Jest to kluczowa sprawa przy programowaniu, szczególnie w środowisku komercyjnym. Gdy praca odbywa się w zespołach, bardzo ważne jest, aby ich członkowie mogli możliwie jak najszybciej zrozumieć kod napisany przez innych.

Anatomia std::algorytmu

Każdy algorytm STL operuje na zakresie elementów, definiowanych przez iteratory. Zdecydowana większość przyjmuje parę iteratorów, która definiuje zakres zgodnie z konwencją `[first, last)`, tzn. przedział na który wskazują jest domknięty z lewej strony i otwarty z prawej (podobnie jak iteratory zwracane przez metody `begin` i `end` kontenerów). Dzięki temu algorytm jest niezależny od kontenera, na którego elementach ma operować. Zobaczmy teraz, że już para iteratorów wystarczy, aby zrobić coś ciekawego.

> Do generacji losowych wektorów w zadaniach poniżej możesz wykorzystać np. szablon funkcji makeRandomVector.

Zadanie 1 Wygeneruj wektor losowych liczb całkowitych z przedziału [0, 10] i wyświetl go. Następnie posortuj go rosnąco używając algorytmu `std::sort`. Zweryfikuj poprawność działania algorytmu wyświetlając ponownie wektor.

Wiele algorytmów przyjmuje także dodatkowe parametry. Zobaczmy to na przykładzie.

Zadanie 2 Wygeneruj wektor losowych liczb całkowitych z przedziału [0, 10]. Policz wystąpienia liczby 7 używając algorytmu `std::count`.

Funktory

Czasem możemy chcieć dostroić nie tylko parametry, ale także elementy zachowania algorytmu. Na przykład, możemy chcieć posortować zakres nie rosnąco, lecz malejąco. Domyślnie, algorytm `std::sort` porównuje elementy operatorem `<`. Aby sortować malejąco, wystarczy zamienić operator `<` na operator `>`. `std::sort` daje nam możliwie ogólny interfejs: możemy podać dodatkowy argument, który definiuje dla algorytmu operację porównania. Argument ten musi być funktorem przyjmującym 2 argumenty sortowanego typu. Funktor to po prostu obiekt, dla którego możemy zawołać metodę `operator()`. Może to być zatem np. obiekt klasy, dla której odpowiednio przeciążony jest ten operator, albo wskaźnik do odpowiedniej funkcji. Po szczegółową dyskusję dotyczącą funktorów odsyłamy czytelnika do drugiego akapitu podrozdziału pt. `std::visit` instrukcji nr 4.

Zadanie 3 Wygeneruj wektor losowych liczb całkowitych. Wyświetl go. Posortuj go malejąco używając algorytmu `std::sort` i odpowiedniego funktora. Zweryfikuj poprawność działania algorytmu wyświetlając ponownie wektor.

Zobaczmy inny przykład, gdzie przydatne mogą być funktory.

Zadanie 4 Wygeneruj wektor losowych liczb całkowitych z przedziału [0, 10]. Policz wystąpienia liczb większych od 7 używając algorytmu `std::count_if`.

W zadaniu 4 wykorzystaliśmy fakt, że liczba, do której porównywaliśmy elementy wektora, była znana w czasie kompilacji (ponieważ 7 występuje jawnie w treści zadania). W kodzie wystąpiła zatem w pewnym miejscu linijka typu `return x > 7`. Co jednak, jeżeli potrzebujemy przekazać do funktora parametry, które poznamy dopiero w czasie wykonania programu? Przykładem takiego problemu jest zadanie 5. Przeczytajmy teraz jego treść (ale jeszcze go nie wykonujmy). Zazwyczaj, gdy chcemy przekazać do funkcji (lub funktora) jakieś parametry, dołączamy je do listy argumentów. Tutaj nie mamy jednak tej opcji, gdyż `std::count_if` wymaga konkrentej sygnatury funkcji (funktora). Najprostszym (i najgorszym) sposobem rozwiązania tego problemu jest zdefiniowanie globalnych zmiennej, której wartość wczytujemy z konsoli, a następnie odczytujemy w ciele funkcji (funktora). Spróbujmy teraz wykonać w ten sposób poniższe polecenie.

Zadanie 5 Napisz program, który generuje wektor losowych liczb z przedziału [0, 10], wczytuje z konsoli liczbę całkowitą `a`, a następnie drukuje liczbę elementów wektora większych od `a`.

W języku C++ unikamy za wszelką cenę obiektów globalnych, gdyż ich czas życia nie jest w żaden sposób ograniczony (jest równy czasowi wykonania progrmu). Ich destruktor wołany jest więc dopiero po wyjściu z funkcji `main`. Jest to mało optymalne oraz bardzo pogarsza przejrzystość kodu (czytając kod i natrafiając nagle na jakieś dziwne parametry globalne możemy nie rozumieć, skąd one się w ogóle biorą). Szczęśliwie, w odróżnieniu od C, w C++ możemy korzystać z funktorów posiadających stan (ang. *stateful*). Oznacza to, że możemy zdefiniować klasę, przeciążającą operator `( )` oraz posiadającą jakieś pola. Możemy następnie podać do algorytmu obiekt takiej klasy, którego polom wcześniej przypisaliśmy odpowiednie parametry. Czas jego życia jest ograniczony, a jego sens istnienia widoczny jest na pierwszy rzut oka (nie ma więc problemu z czytelnością).

Zadanie 6 Wykonaj zadanie 5, tym razem posługując się funktorem posiadającym stan. Zdefiniuj klasę posiadającą pole typu całkowitego oraz przeciążającą odpowiednio operator `( )`. Stwórz obiekt tej klasy, przypisz do jego pola wartość wczytaną z klawiatury, a następnie podaj go do `std::count_if`.

Wyrażenia lambda

Aby uniknąć konieczności definiowania nowych klas za każdym razem, gdy chcemy stworzyć relatywnie prosty funktor, od C++11 możemy używać wyrażeń lambda (zwanych także funkcjami anonimowymi, a potocznie po prostu lambdami). Wyrażenie lambda ma następującą składnię (po wszystkie szczegóły odsyłamy jak zawsze do dokumentacji):

```
[ /* capture */ ] ( /* argumenty */ ) { /* ciało */ }
```

Tworzy ono obiekt funkcyjny (funktor), którego operator nawiasów okrągłych przyjmuje argumenty wskazanego typu i wykonuje na nich operacje zdefiniowane w ciele (zupełnia jak tradycyjna funkcja!). Nawiasy kwadratowe i rolę *capture* omówimy za chwilę. Na razie zobaczmy prosty przykład definicji lambdy, która zwraca sinus argumentu:

```
auto lambda_sin = [](double x){ return sin(x); }; // lambda_sin jest funktorem
double x = M_PI / 2.;
double sin_x = lambda_sin(x); // sin_x == 1. (+/- błąd zmiennoprzecinkowy)
```

Typ zmiennej `lambda_sin` jest nadawany przez kompilator, w związku z czym musimy użyć słowa kluczowego `auto`. Nie musimy jednak przypisywać lambdy do zmiennej, możemy użyć jej jako rvalue:

```
double x = M_PI / 2.;
double sin_x = [](double x){ return sin(x); }(x);
```

Tak będziemy też najczęściej postępować w przypadku algorytmów, gdzie zazwyczaj definiujemy wyrażenie lambda bezpośrednio w liście argumentów. Korzystanie z prostych lambd obrazuje także ten kawałek kodu. Zachęcamy czytelnika do podejrzenia, jak tak naprawdę kompilator je rozwija (należy wcisnąć przycisk "Cpp Insights", a następnie przycisk "Play"). Jak widać nie dzieje się tu nic magicznego, po prostu definiowane są za nas klasy z odpowiednimi polami i operatorami. Użycie funkcji anonimowych pozwala nam jednak oszczędzić sporo wysiłku.

Lambdy przedstawione powyżej można śmiało zastąpić tradycyjnymi funkcjami, jedyna korzyść z ich użycia to niewielka oszczędność czasu oraz ograniczenie widoczności nazwy funkcji pomocniczej do bieżącego scope'u. Zobaczmy teraz, do czego służy *capture* lambdy (autor nie zna dokładnego i zwięzłego ekwiwalentu tego określenia w języku polskim, *capture* w kontekście regexów nazywa się czasem kolokwialnie "grupą kapturkową"). W nawiasach kwadratowych możemy "łapać" zmienne ze scope'u, w którym zdefiniowana została lambda. Możemy to robić przez kopię lub referencję (a także przeniesienie, jeżeli jest taka potrzeba). W instrukcji ograniczymy się do łapania przez referencję całego scope'u. Dodając do definicji lambdy jeden znak `-` `&` `-` zyskujemy w jej ciele dostęp do wszystkich zmiennych, które widoczne są dla jej definicji. Na przykład:

```
double omega = M_PI;
double phi   = M_PI / 2.;

// Błąd kompilacji - omega i phi są niewidoczne w środku lambdy
auto bad_harmonic = [](double t){ return sin(omega * t + phi); };

// OK - łapiemy referencje do omega i phi
auto good_harmonic = [&](double t){ return sin(omega * t + phi); };

// Korzystamy tak samo
double t0 = 0;
double y0 = good_harmonic(t0);
```

Nie musimy zatem samodzielnie definiować dodatkowej klasy, tak jak w zadaniu 6!

Zadanie 7 Wykonaj ponownie zadanie 6, tym razem posługując się lambdą zamiast funktorem jawnie zdefiniowanej klasy.

Ćwiczenia

Poniżej zamieszczono zadania treningowe, rozwiązanie których pomoże poznać kilka nowych algorytmów oraz przećwiczyć pracę z szablonami funkcji dostępnymi w nagłówkach `algorithm` i `numeric`. Kolejność zadań dobrana została zgodnie z rosnącym stopniem trudności (w opinii autora). Zachęcamy czytelnika do wykonania jak największej ich liczby. Zdecydowanie wskazane jest wykonanie przynajmniej pierwszego zadania z każdej kategorii poza ostatnią.

Bezpośrednie zastosowania algorytmów

Ćwiczenie I - szukanie identycznych sąsiadów Napisz program, który wcztuje z klawiatury ciąg znaków (`std::string`) i zwraca 1 jeżeli występują w nim 2 te same znaki pod rząd i 0 w innym wypadku. użyj algorytmu `std::adjacent_find`.

Ćwiczenie II - szukanie podciągu Napisz program, który wczytuje z klawiatury ciąg znaków, a następnie zwraca 1 jeżeli podany ciąg zawiera w sobie podciąg "piesek" lub "kotek" i 0 w innym wypadku. Użyj algorytmu `std::search`.

Ćwiczenie III - odwracanie ciągów Napisz program, który wczytuje z klawiatury ciąg znaków, a następnie drukuje go do konsoli w odwróconej kolejności. Użyj algorytmu `std::reverse`.

Ćwiczenie IV - iloczyn skalarny Stwórz 2 wektory liczb zmiennoprzecinkowych. Oblicz ich iloczyn skalarny używając algorytmu `std::inner_product`.

Łączenie algorytmów

Ćwiczenie V - sortowanie podciągu Stwórz wektor losowych liczb całkowitych z przedziału [0, 10]. Znajdź w nim pierwsze wystąpienie liczby 7. Posortuj elementy wektora występujące przed znalezioną liczbą 7 (lub cały wektor, jeżeli 7 nie występuje). Użyj `std::find` i `std::sort`.

Przykład:

- input: [3 2 1 7 9 7 8 10]
- output: [1 2 3 7 9 7 8 10]

Ćwiczenie VI - idiom *remove-erase* Stwórz wektor losowych liczb całkowitych z przedziału [0, 10]. Następnie usuń z niego wszystkie wystąpienia liczby 3. Użyj algorytmu `std::remove` oraz metody wektora `erase`. Spróbuj wykonać zadanie w 1 linijce (wyjąwszy stworzenie wektora). Rozmiar (`size`) wektora powinien być po operacji mniejszy o liczbę wystąpień liczby 3.

Ćwiczenie VII - obrót względem wartości Stwórz wektor losowych liczb całkowitych z przedziału [0, 10]. Znajdź w nim pierwsze wystąpienie liczby 7. Przesuń elementy wektora występujące przed znalezioną liczbą 7 na koniec wektora (zachowaj ich kolejność). Użyj `std::find` i `std::rotate`.

Przykład:

- input: [3 2 1 7 9 7 8 10]
- output: [7 9 7 8 10 3 2 1]

Algorytmy przyjmujące funktory

Ćwiczenie VIII - sprawdzenie czy elementy z zakresu spełniają warunek logiczny Stwórz wektor losowych liczb zmiennoprzecinkowych z przedziału [-1, 1]. Sprawdź, czy co najmniej jedna ze stworzonych liczb jest większa niż 0.9. Użyj algorytmu `std::any_of`.

Ćwiczenie IX - sinus zakresu Stwórz wektor losowych liczb zmiennoprzecinkowych z przedziału [-1, 1]. Stwórz drugi wektor tego samego rozmiaru i wypełnij go sinusami wartości z pierwszego wektora. Użyj algorytmu `std::transform`.

Ćwiczenie X - sortowanie elementów wektora mniejszych od wartości Stwórz wektor losowych liczb całkowitych z przedziału [0, 10]. Przestaw jego elementy tak, aby najpierw wystąpiły wszystkie liczby większe od 6 (poza tym wymaganiem mogą być one dowolnie przestawione). Następnie posortuj część zakresu zawierającą elementy większe od 6. Użyj `std::partition` i `std::sort`.

Przykład:

- input: [1 10 2 9 3 8 4 7 5 6]
- output (jeden z dopuszczalnych): [7 8 9 10 3 1 5 6 2 4]

Zadanie trudniejsze

Ćwiczenie XI - indeksy sortowania Napisz algorytm `sorted_indices`, o sygnaturze:

```
template < typename ConstIt, typename Comp >
std::vector< size_t > sorted_indices(ConstIt first, ConstIt last, Comp compare
```

który zwraca indeksy elementów podanego zakresu `[first, last)` po jego posortowaniu przy użyciu funktora porównującego `compare`. Algorytm nie powinien modyfikować zakresu. Zachowanie jest analogiczne do napisania w MATLABie `[~, i] = sort(v)`. Innymi słowy, poniższy kod powinien się wykonać:

```
std::vector<unsigned> v = { /* ... */ };
auto r = sorted_indices(v.begin(), v.end(), std::less<unsigned>{});
int a = 0;
for(size_t i = 0; i < v.size(); ++i)
{
    assert(a <= v[r[i]]);
    a = v[r[i]];
}
```

Przykład:

- input: [1 10 2 9 3 8 4 7 5 6]
- output: [0 2 4 6 8 9 7 5 3 1]

Podpowiedzi:

1. Załóż, że podany iterator jest losowego dostępu. Iteratory losowego dostępu STL mają zdefiniowany operator `[]`, który pozwala na indeksowanie elementów następujących po danym iteratorze. Na przykład:

```
std::vector<int> v = {1, 2, 3, 4, 5};
auto it = v.begin();
int a = it[2]; // a ma wartość 3
```

2. Funkcja `std::distance` zwraca odległość między dwoma iteratorami (np. dla `begin` i `end` będzie to rozmiar kontenera).
3. Możesz użyć algorytmu `std::iota` (z nagłówka `numeric`) do generacji wektora kolejnych rosnących liczb całkowitych (zwięźlejszy zapis niż pętla for).
4. W ostateczności możesz spojrzeć na rozwiązanie.