AiSD Laboratorium 2

Wskazówki do zadań

Wskazówka do zadania 1.

Zasadnicza część algorytmu może wyglądać następująco:

```
forall 0<= i < N_A
    print A[i]
```

Wyświetlanie można wzbogacić o zachowanie stałej szerokości pola (setw) i/lub zaczęcia nowego wiersza, gdy przekroczona została pewna ilość znaków.

O ile tak przedstawiony sposób jest w zasadzie jedynym prawidłowym to jego implementacja może być zrealizowana na wiele różnych sposobów, wykorzystujących różne konstrukcje składniowe, które są nei dostepne na poziomie pseudokodu.

Poniżej zostanie pokazane kilka sposobów. Załóżmy, że zdefiniowane są nastepujące tablice:

```
// T oznacza pewien ustalony typ, dla ustalenia uwagi T = short

T A[A_size]             = {/*elementy tablicy A */};
std::array<T,B_size> A  = {/*elementy tablicy B */};
std::vector<T> C        = {/*elementy tablicy C */};
```

Implementacja bezpośrednia dla typu tablicowego

Najprostsza implementacja może wyglądać tak:

```
void print_legacy_array_mod_1(const T* A, const size_t size){

    for(size_t i = 0; i < size; ++i)
        std::cout << A[i] << " ";
    std::cout << '\n';
}
```

Zauważymy, że w tym wypadku konieczne jest przekazanie rozmiaru tablicy jako parametru. Wykorzystując fakt, że wskaźniki można iterować można zrobić, tak

```
void print_legacy_array_mod_2(const T* first,const T* last){

    while(first != last)
        std::cout << *(first++) << " ";
    std::cout << '\n';
}
```

Przedstawione funkcje łatwo można wzbogacić o formatowanie (np. wypisywanie w kolumnach).

Wywołanie powyższych funkcji wygląda następująco:

```
print_legacy_array_mod_1(A,A_size)
print_legacy_array_mod_2(A,A+A_size)
```

Uwaga: Można zrobic też tak, bez ryzyka wyjścia po za tablicę, ale zadziała tylklo gdy kompilator będzie umiał sam wydedukować rozmiar, szczegóły są nieco skomplikowane

```
#include <algorithm>

std::for_each(std::begin(A),std::end(A),[](auto& el){ std::cout << el << " ";} );
```

Patrz: `zad_1_legacy_demo.cpp`

Kontenery biblioteki standardowej

Naiwna implementacja:

```
// T w parametrze szablonu to nie to samo T zadeklarowane na początku
template<class T, size_t N>
void print_std_array(const std::array<T,N>& A){

    for(size_t i = 0; i < A.size(); ++i)
        std::cout << A[i] << " ";
}

template<class T>
void print_std_vector(const std::vector<T>& A){

    for(size_t i = 0; i < A.size(); ++i)
        std::cout << A[i] << " ";
}
```

Zdecydowanie lepszym pomysłem jest wykorzystanie mechanizmów na których zbudowana jest biblioteka standardowa, tzn. iteratorów. Poniższa funkcja zadziała dla każdego kontenera dla ktorego zdefiniowany jest tzw. `forward_iterator`.

```
template<class InputIt>
void print_table(InputIt first, InputIt last){

    while(first != last)
        std::cout << *(first++) << " ";
}
```

Wywołanei tej funkcji wygląda tak

```
// w tym wypadku wykorzystany został mechanizm detekcji typów dla argumentów szablonu
print_table(C.begin(),C.end());
```

Co ciekawe, ta funkcja zadziała także dla typu tablicowego:

```
print_table(A,A+A_size);
print_table(std::begin(A),std::end(A));
```

Innym sposobem jest wykorzystanie z algorytmu `std::for_each`:

```
std::for_each(C.begin(),C.end(),[](auto& el){ std::cout << el << " ";} );
```

Bez wątpienia można wypracować wiele innych sposobów. Koszące może się wydawać przeciążenie operatora <<. Nie koniecznie jest to dobry pomysł dla tablic zawierajace proste typy, zaś dla typów złożonych lepiej jest przeciążyć oprator << dla tego typu.

Patrz: `zad_1_containers_demo.cpp`

Wskazówka do zadania 2.

Uwaga Kopiowanie pomiędzy tablicami różnych rozmiarów jest proszeniem się o kłopoty. Dlatego lepiej unikać takich przypadków, szczególnie pisania funkcji która zezwala na kopiowanie do miejszej tablicy.

Zasadnicza część algorytmu może wyglądać następująco:

```
if N_A <= N_B
    for i = 0 to N_A-1
        B[i] = A[i]
else                    // N_A > N_B
    for i = 0 to N_B-1
        B[i] = A[i]
```

Ponieważ obie pętle są identyczne, można ten algorytm zredukować do jednej pętli znajdując rozmiar mniejszej tablicy

```
for i=0 to min(N_A,N_B)
    B[i]=A[i]
```

W przypadku kopiowania z offsetem algorytm mógłby korzystać wykonywać przypisanie `B[i+offset] = A[i]`. Trzeba wtedy zadbać, aby zawsze spełniony był warunek `i+offset < N_A` odpowiednio modyfikując warunek w pętli.

Implementacje tego algorytmu dla typów tablicowych można łatwo wykonac naśladując rozwiązanie poprzedniego zadania (patrz `zad_2_legacy_demo.cpp`).

Funkcje biblioteczne

Pisanie takiego prostego algorytmu w innym celu niż ćwicznie jest złym pomysłem. Lepszym rozwiązaniem jest skorzystanie z funkcji bibliotecznych

- `std::copy`
- `std::copy_n`
- zdefiniowanych w nagłówku `<algorithm>` lub z operatorów przypisania zdefiniowanych dla kontenerów.

Implementacja `std::copy` i `std::copy_n` może wyglądać tak

```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt d_first)
{
    while (first != last) {
        *d_first++ = *first++;
    }
}

template< class InputIt, class Size, class OutputIt>
OutputIt copy_n(InputIt first, Size count, OutputIt d_first)
{
    if (count > 0) {
        *result++ = *first;
        for (Size i = 1; i < count; ++i) {
            *result++ = *++first;
        }
    }
}
```

Żadnach z tych funkcji nie sprawdza czy w kontenerze docelowym (wskazywanym przez `d_first`) jest odpowiednia ilość miejsca. Zapewnienie tego leży po stronie programisty.

Kopiowanie dla typu tablicowego

```
T A[A_size] = {...};
T B[B_size] {};

// jeżeli A_size<=B_size takie koopiowanie jest bezpieczne
std::copy(A, A+A_size, B);

// jeżeli A_size > B_size zostanie nadpisany
// obszar pamięci bezpośrednio za tablicą B
    std::copy(A, A+A_size, B);      // ALERT !!!

// jeżeli naprawdę wiemy, że chcemy przekopiowac do tablicy mniejszej
// trzeba zadbać o to aby przekopiować tylko tyle ile się zmieści
    std::copy_n(A, std::min(A_size,B_size), B);
```

Kopiowanie dla `std::array`

Aby przekopiowć tablice `std::array` można postąpić podobnie jak wyżej

```
std::array<T,A_size> A = {...};
std::array<T,B_size> B = {...};

// bezpieczne jeżeli A_size <= B_size
std::copy(A.begin(), A.end(), B.begin())

// jeżeli A_size > B_size
    std::copy_n(A.begin(), std::min(A.size(),B.size()), B.begin());
```

Jeżeli `A.size() == B.size()` lepiej jest skorzystać z wbudowanego operatora przypisania:

```
// typ std::array posiada zdefiowany operator=
// pozwalajacy na kopiowanie zawartości pomiędzy
// tablicami tego samego rozmiaru tzn. w takiej sytuacji

std::array<T,N> A = {/* elementy */};
std::array<T,N> B = {};

A = B;
```

Patrz: `zad_2_stdarray_demo.cpp`

Zadanie: Użyć `std::copy_n` do kopiowania z offsetem.

Kopiowanie dla `std::vector`

W przypadku `std::vector`, możemy postąpić analogicznie jak wyżej, uzywając `std:: copy`, pozostawiamy to jako ćwiczenie. Klasa `vector`, jako implementacja tablicy dynamicznej, udostępnia także inne metody. Patrz: `zad_2_std::vector_demo.cpp`

Kopiowanie pomiędzy tablicami różnych typów

Do tego celu można napsać funckje która przyjmuje odpowiednie argumenty, ale lepiej jest użyć `std::copy` albo operatora =. Szczegóły pozostawiamy jako ćwiczenie.

Wskazówka do zadania 3.

Podobnie jak w zadaniu 1. tylko zamiast `std::cout` trzeba użyć `std::ofstream`. Plik powinien byc otwarty i zamknięty w ciele funkcji.

Ponieważ `std::cout`, `std::ofstream` i `std::ostringstream` są ezgemplarzami tej samej ogólniejszej klasy `std::ostream` więc można za jednym zamachem napsać funkcję która zapisuje zawartość tablicy do kazdego z tych strumieni. Z zastrzerzeniem, że są one dostępne, np.: plik jest otwarty przed wywołaniem funkcji.

W niekoniecznie najprostszej wersji może to wyglądać tak:

```
template<typename InputIt>
void tostream_table(std::ostream& out, InputIt first, InputIt last){

    while(first != last)
        out << *(first++) << " ";
}
```

Oczywiście można to zrobić bez użycie szablonów tak jak w zadaniu 1.

Wykorzystanie tej funkcji wygląda następująco:

```
std::array<T,10> A = {/* elementy A*/};
std::vector<T> B = {/* elementy B */};

std::stringstream out_sstream;

    // wyswietlanie na ekranie
    tostream(std::cout,A.begin(),A.end());
    tostream(std::cout,B.begin(),B.end());

    // zapis do strumienia znakowego
    tostream(out_sstream,A.begin(),A.end());
    tostream(out_sstream,B.begin(),B.end());

std::ofstream out_file("output.txt");

    //zapis do pliku
    tostream(out_file,B.begin(),B.end());
    tostream(out_file,A.begin(),A.end());

    output.close();  // don't forget!
```

Patrz: `zad_3_demo.cpp`

Wskazówka do zadania 4.

Algorytm jest oczywisty: przejść po całej tablicy, w przypadku napotkania zliczanego elementu zwiększyć licznik.

Jego implementacja nie powinna być trudna.

Patrz także: `std::count`

Wskazówka do zadania 5.

Naiwny algorytm mogłby przekopiować tablicę do tablicy pomocniczej zaczynając od końca, a potem spowrotem dla orginalnej. Jest on poprawny ale wymaga O(2n) (dwa kopiowania) i O(n) (pomocnicza tablica) pamięci. Oszczędniejszy algorytm zamienia kolejność elementów in-place, tzn. operując na tej samej tablicy.

Uzywane są dwa indeksy: `left` i `right`. Tablica przechodzona jest od obu końców, w każdym kroku zamieniane są elementy wskazywane przez `left` i `right`, a następnie `left` jest przesuwany o jeden do przodu, a `right` o jeden do tyłu. Algorytm kończy się, gdy oba indeksy się spotkają. Problemem (chyba) jest poradzenie sobie z tym czy rozmiar tablicy jest parzysty czy nie. Algorytm działa w czasie O(n) i nie potrzebuje dodatkowej pamięci.

Pierwsze podejście do algorytmu mogłoby wyglądać tak:

```
Input: tablica A = {A[1],...,A[N]}
Output: tablica A z odwróconą kolejnością elementów

left = 1
right = N

while(left != right)
    zamień A[left] z A[right]
    left++;
    right--;
```

Niestety jest on prawidłowy tylko dla parzystych/nieparzystych (niepotrzebne skreślić) N. Poprawienie algorytmu pozostawiamy jako ćwiczenie.

W bibliotece standardowej dostępna jest funkcja reverse.

Wskazówka do zadania 6. i 7. i 8.

Coś musi pozostać dla Państwa...