Wykorzystanie elementów z biblioteki standardowej C++: `vector`, `list`, `complex` oraz `string`.

1.Biblioteka standardowa w języku C++

Biblioteka standardowa to biblioteka zawierająca podstawowe funkcje i typy danych, dostarczana wraz kompilatorem lub interpreterem danego języka programowania. Dla niektórych języków, w tym C++, istnieje formalna specyfikacja zawartości i działania biblioteki standardowej.

Standard Template Library, STL (ang. = Standardowa Biblioteka Wzorców) – biblioteka C++ zawierająca algorytmy, pojemniki, iteratory oraz inne konstrukcje w formie szablonów, gotowe do użycia w programach. STL jest to tzw. biblioteka generyczna, co oznacza, że jej składniki współpracują równie dobrze z typami wbudowanymi w język, z typami wbudowanymi w bibliotekę, co z typami zdefiniowanymi przez użytkownika, pod warunkiem, że spełniają pewne określone warunki.

2.Pojemniki (zasobniki)

Jedną z najważniejszych rzeczy wprowadzonych przez STL, są pojemniki, czyli obiekty zbiorcze. Jest ich kilka rodzajów, różnią się konstrukcją i tym samym wydajnością poszczególnych operacji.

`2.1. vector`

Zasobnik `vector` jest klasą bardzo podobną do tablicy. Zawiera w sobie strukturę danych z sąsiadującym umiejscowieniem w pamięci. Jest to ulepszona wersja tablic z C lub C++. Nie jest konieczna wiedza jak dużej tablicy potrzebujemy, kiedy deklarujemy obiekt klasy `vector`. Możemy dodawać nowe elementy na koniec obiektu - kontenera korzystając z metody `push_back`.

Aby używać klasy wektor należy dodać plik nagłówkowy:

```
#include <vector>
```

`vector` jest wzorcem klasy, więc jeśli podczas deklaracji obiektu trzeba określić typ obiektów, jaki ma w sobie zawierać klasa vector. Na przykład poniższy kod:

```
vector<int> v1;
vector<string> v2;
vector<Transformator > v3;
```

deklaruje `v1` jako obiekt typu `vector`, która zawiera w sobie liczby całkowite, `v2` jako obiekt typu `vector` przechowującą napisy (klasa `string` została omówiona w dalszej części instrukcji) oraz obiekt `v3` jako obiekt typu `vector` przechowującą obiekty typu

`Transformator` (klasę zdefiniowaną przez użytkownika). Te deklaracje nic nie mówią na temat wielkości poszczególnych obiektów (co oznacza, że będą to wartości domyślne).
Można podać początkową wielkość obiektu klasy `vector` używając deklaracji takiej jak:

```
vector<char> v4(26);
```

w ten sposób deklarujemy obiekt typu `vector` o rozmiarze początkowym 26 elementów typu `char`. Jest również sposób na inicjalizację elementów obiektu typu `vector`. Deklaracja:

```
vector<float> v5(100,1.0);
```

mówi o tym, że `v5` jest obiektem typu `vector` przechowującym zmienne typu `float`. Obiekt ten składa się ze stu elementów i każdy element jest inicjalizowany wartością 1.0

Aby dowiedzieć się ile elementów zawiera obiekt typu `vector` należy użyć metodę `size`. Ta funkcja nie ma żadnych argumentów i zwraca liczbę całkowitą, która określa ile elementów zawiera w sobie dany obiekt. Przykład użycia:

```
cout << "wektor v1 ma " << v1.size() << " elementów\n";
```

Aby sprawdzić czy obiekt jest pusty można skorzystać z metody `empty`. Metoda ta nie ma żadnych argumentów i zwraca wartość typu `bool`: `true` (1) jeśli obiekt jest pusty lub `false` (0) jeśli nie jest pusty. Przykład:

```
if( v1.empty() )
     cout << "wektor pusty" << endl;
else
     cout << "wektor NIE pusty" << endl;
```

Dostęp do poszczególnych elementów obiektu typu `vector` można uzyskać korzystając z operatora []. Aby wydrukować wszystkie elementy obiektu typu `vector` można użyć kodu takiego jak poniżej :

```
for (int i=0; i<v1.size(); i++)
     cout << v1[i];
```

(ten kod jest bardzo podobny do kodu pisanego dla zwykłych tablic).
Z operatora[] można również korzystać żeby zmienić wartość poszczególnych elementów klasy vector.

```
for (int i=0; i<v1.size(); i++) v1[i] = 2*i;
```

Oprócz operatora[] jest wiele sposobów na dostęp lub zmianę elementów obiektu typu vector.

Funkcja `push_back` dodaje nowy element na końcu klasy `vector`. Metoda ta pobiera jeden argument typu przechowywanego w klasie `vector`:

```
for (int i=0; i<10; i++)
     v1.push_back(i);
```

Powyższy kod dodaje do tablicy `v1` 10 elementów o wartości od 0 do 9, za każdym razem zwiększając wymiar tablicy o jeden.

Funkcja `pop_back` usuwa ostatni element z klasy `vector`. Metoda ta nie pobiera żadnych argumentów. Błędem jest użycie `pop_back` na pustym obiekcie klasy `vector`.

```
if( !v1.empty() )
     v1.pop_back();
```

Powyższy kod dla niepustego wektora usuwa ostatni element z tablicy, zmniejszając jej wymiar o jeden.

Aby zmienić rozmiar tablicy można posłużyć się metodą resize. Jeżeli wymiar tablicy jest zwiększany, to nowe elementy są tworzone przez ich konstruktor domyślny lub jako kopia przekazanego elementu (patrz dalej). Jeżeli natomiast tablica jest zmniejszana, to zostają kasowane elementy od końca wektora. Metoda resize ma dwie postacie:

a) `resize( num );`
   Opis: Zmienia liczbę elementów na num. Jeśli wzrasta liczba elementów, to nowe elementy tworzone są poprzez ich konstruktor domyślny.
b) `resize( num, elem );`
   Opis: Zmienia liczbę elementów na num. Jeśli wzrasta liczba elementów, to nowe elementy tworzone są kopiami elementu elem..

Na przykład poniższy kod:

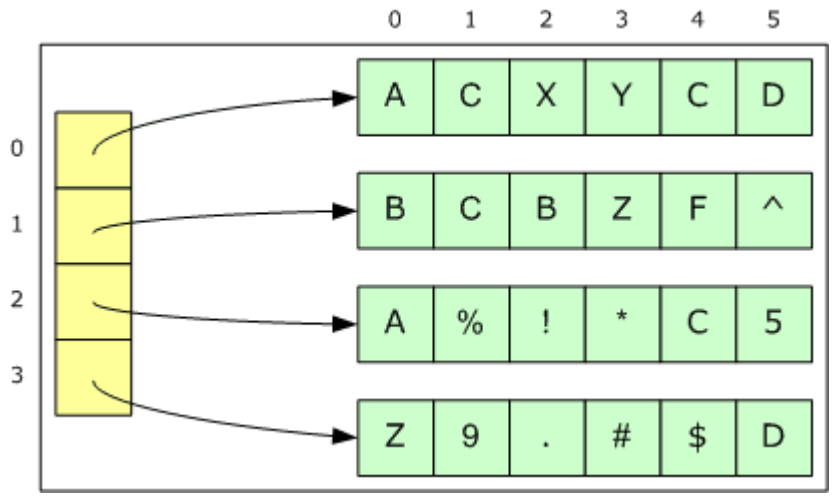
```
std::vector<int> v1(10,0);
v1.resize(15,1);
```

utworzy tablicę 10 elementów o wartości 0, a następnie zwiększy liczbę elementów do 15, a brakujące 5 elementów będzie miało wartość 1.

Funkcja `insert` wstawia jeden lub więcej elementów, na określoną pozycję w klasie `vector`. Natomiast metoda `erase` usuwa jeden lub więcej elementów z określonych pozycji. Obie funkcje są kosztownymi operacjami na zasobnikach typu `vector`. Jeśli zachodzi konieczność częstego wstawiania lub usuwania elementu na określonych pozycjach należy rozważyć użycie zasobnika `list` na którym operacje te są bardziej wydajne. Obie funkcje nie będą omawiane w niniejszej instrukcji, ponieważ wymagają one wprowadzenia i omówienia pojęcia *iteratorów*, kolejnego elementu biblioteki STL, co wykracza poza zakres ćwiczenia.

2.2. Tablica dwuwymiarowa

Kontener vector umożliwia również tworzenie tablic dwu i wielowymiarowych. Koncepcja deklarowania tablicy wielowymiarowej przedstawia się następująco:
Deklarowanie wektora, który będzie przechowywał inne wektory – wiersze tablicy. Dostęp do elementu *i,j* odbywa się w ten sposób, że najpierw z pierwszego wektora należy pobrać wektor z informacjami dla wiersza o indeksie *i*, a następnie *element* odpowiadający kolumnie *j*. Oto ilustracja graficzna powyższego sposobu:



Na przykład poniższy kod:

```
std::vector<std::vector<double> > tab1;
std::vector<std::vector< liczba_zesp > > tab2;
```

deklaruje `tab1` jako obiekt typu `vector`, która zawiera tablicę dwuwymiarową dla liczbami, obiekt `tab2` jako tablicę dwuwymiarową przechowującą obiekty typu `liczba_zesp` (klasę omówioną w dalszej części instrukcji). Te deklaracje nic nie mówią na temat wielkości poszczególnych obiektów (co oznacza, że będą to wartości domyślne).
Uwaga: pomiędzy double> a > *MUSI* być znak spacji, inaczej kompilator oznajmi błąd.

Aby zadeklarować tablicę dwuwymiarową o wymiarze 10 na 10 należy na przykład wpisać:

```
vector< vector<double> > v1(10,10);
```

Aby zmienić wymiar tablicy, należy użyć omówionej wcześniej metody `resize`, co ilustruje następujący kod:

```
v1.resize(20);
for(int i=0; i<v1.size(); i++)
     v1[i].resize(20);
```

Dostęp do elementu *i,j* można uzyskać za pomocą dwukrotnego użycia operatora []:

```
v1[0][0]=10.0;
v1[3][3]=v1[0][0] + 123;
```

Należy oczywiście uważać, aby nie próbować odczytać elementu z poza tablicy.

2.3. Inne kontenery

Oprócz kontenera `vector` biblioteka STL wprowadza inne kontenery:

a) `deque`

Jest to kolejka o dwóch końcach. Kolejka `deque` jest bardzo podobna do wektora `vector`. Różnica polega na tym, że w przypadku kolejki tablica dynamiczna jest otwarta z obydwu końców. Oznacza to, że kolejka zapewnia szybkie wstawianie i usuwanie elementów zarówno na końcu, jak i na początku kontenera.

b) `list`

Elementy listy zorganizowane są w postaci listy dwukierunkowej. Dostęp do elementów listy odbywa się poprzez przejście przez powiązania listy od początku (lub końca) do szukanego elementu. Lista ma tą zaletę, że wstawianie elementów w dowolnym miejscu listy jest dużo szybsze niż w przypadku wektorów i kolejek.

c) `set, multiset`

Kontenery `set` (zbiór) i `multiset` (wielozbiór) wykonują automatyczne sortowanie swoich elementów zgodnie z określonym kryterium. Różnica polega na tym, że wielozbiory dopuszczają powtórzenie elementów. Największą zaletą automatycznego sortowania jest wysoka wydajność przy wyszukiwaniu elementów o określonej wartości. Automatyczne sortowanie nakłada jednak ograniczenie: nie można bezpośrednio zmieniać wartości elementu, ponieważ mogłoby to zburzyć prawidłową kolejność.

d) `map, multimap`

Kontenery `map` (mapa) i `multimap` (multimapa) są kontenerami, których elementami są pary klucz-wartość. Ich elementy są automatycznie sortowane zgodnie z określonym kryterium sortowania, stosowanymi wobec bieżącego klucza (jednego z elementów przechowywanej pary).

3.Klasa string

4.Klasa `complex` dla reprezentacji liczb zespolonych.

Klasa `complex` powstała jako jeden z najwcześniejszych (jeśli nie liczyć `iostream`) części biblioteki standardowej C++. Aby móc używać tej klasy, należy dodać plik nagłówkowy:

```
#include <complex>
```

Klasa `complex` jest wzorcem z jednym argumentem – typem pola (zazwyczaj typu zmiennoprzecinkowego), zatem aby zadeklarować zmienną tego typu należy wpisać następujący kod:

```
complex<double> z1;
```

Nie jest to wygodne przy deklaracji wielu zmiennych oraz tablic liczb zespolonych, dlatego najlepiej posłużyć się instrukcją `typedef`, najlepiej wpisaną zaraz po dodaniu pliku nagłówkowego:

```
#include <complex>
using namespace std;
typedef complex<double> liczba_zesp;
```

Po tej instrukcji można deklarować obiekty typu `complex<double>`. Następujący kod:

```
liczba_zesp z1, z2;
liczba_zesp *pz = &z1;
liczba_zesp &ref = z2;
```

```
vector<liczba_zesp> vz(10);
```

tworzy dwie zmienne – liczby zespolone `z1` i `z2`, tworzy wskaźnik do liczby zespolonej `pz`, referencję do liczby zespolonej `ref` oraz wektor 10 liczb zespolonych `vz`.

Klasa liczb zespolonych ma implementować operacje na liczbach zespolonych, posiada zatem swoją część rzeczywistą i urojoną oraz różne operacje, które się zwykle na takich liczbach wykonuje (wraz z przeciążonymi operatorami).

4.1. Tworzenie obiektów typu complex

Aby zadeklarować zmienną typu complex należy wpisać:

```
liczba_zesp z1
```

Zostanie utworzona liczba zespolona, której część rzeczywista równa się części urojonej i jest równa zero. Aby nadać wartość początkową należy utworzyć obiekt poprzez konstruktor z parametrami, czyli:

```
liczba_zesp z2(10,20);
```

Powyższy kod utworzy liczbę zespoloną, dla której część rzeczywista jest równa 10, a część urojona 20.

Aby zmienić wartość dla liczby zespolonej, która już istnieje, należy utworzyć nowy obiekt tymczasowy i posłużyć się operatorem podstawienia:

```
liczba_zesp z1;
...
z1 = liczba_zesp(3,3);
```

4.2. Operatory arytmetyczne

Klasa complex dostarcza podstawowe operatory arytmetyczne: +, -, *, /, które są zgodne z tym, jak się te operacje wykonuje na liczbach zespolonych. Przykład:

```
liczba_zesp z1(10,10), z2(2,2), z3;
...
z3 = z1 * z2 – z1 + z2/z1;
```

Po prawej stronie operatora może być zarówno typ complex, jak liczba – typ parametryzujący wzorzec:

```
liczba_zesp z1,z2;
double d=10;
...
z2 = d * z1 + 120.0;
```

Uwaga: Można posługiwać się *TYLKO* typem parametryzującym wzorzec, w omawianym przypadku typem double. Ponad to w sposób jawny należy wpisać, że to jest liczba typu double, czyli nie można wpisywać 120, trzeba wpisać 120.0 (120 kropka 0). Inaczej kompilator zaprotestuje.

4.3. Operatory relacji

Do porównania (relacji) dwóch liczb zespolonych do dyspozycji są operatory == oraz !=. (równy i różny). NIE istnieją operatory >, >= itp., ponieważ nie mają one sensu matematycznego.

4.4. Funkcje specjalizowane

`real, imag` – zarówno jako funkcje globalne z jednym argumentem, jak i metody klas complex, zwracają odpowiednio część rzeczywistą i urojoną. Przykład:

```
cout << "część rzeczywista: " << z1.real() << endl;
```

Za wyjątkiem tych dwóch funkcji wszystkie pozostałe są funkcjami globalnymi.

`abs` – funkcja oblicza wartość absolutna/bezwzględna/moduł czyli sqrt(re2+im2). Przykład:

```
cout << "moduł z1: " << abs(z1) << endl;
```

`arg` – funkcja zwraca argument liczby zespolonej (z postaci wykładniczej liczby zespolonej). Argument jest wyrażany w radianach. Przykład:

```
cout << "argument z1: " << arg(z1) << endl;
```

`polar(mod,arg)` – funkcja służy do konwersji z postaci moduł/argument na liczbę zespoloną w reprezentacji algebraicznej – część rzeczywista i urojona. Argument jest wyrażany w radianach.
Przykład:

```
liczba_zesp z1;
z1 = polar(100, 3.1415/2.0);
```

`conj` – funkcja konwertuje podaną liczbę zespoloną na liczbę zespoloną sprzężoną (dla przypomnienia: jest to liczba, której część urojona ma odwrócony znak).

Ponadto jest przeciążony operator `<<` do wyprowadzania danych do strumienia. Wyprowadza on liczbę zespoloną w postaci "`(re,im)`". Jest również operator `>>`, który pozwala z kolei wprowadzić ze strumienia liczbę zespoloną, pod warunkiem, że jest w takiej właśnie postaci. Przykład:

```
liczba_zesp z1(10,20);
cout << "z1=" << z1;
```

4.5. Funkcje matematyczne

Do funkcji matematycznych działających na liczbach zespolonych, to należą: `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `exp`, `pow`, `log`, `log10` i `sqrt`.

Opracowano na podstawie:

- Materiały własne
- http://pl.wikipedia.org
- http://www.3miasto.net/~chq/c/howto/x1066.html
- http://programex.risp.pl/?strona=strukturyd_formatyp
- http://www.intercon.pl/~sektor/cbx/std/math.html
- "C++ Biblioteka standardowa. Podręcznik programisty". Nicolai M. Josuttis.Wydawnictwo Helion. ISBN: 83-7361-144-4