Wprowadzenie do STL

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf".
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI
INSTYTUT INFORMATYKI

Wprowadzenie do STL

Paweł Mikołajczak

LUBLIN 2012

Instytut Informatyki UMCS
Lublin 2012

Paweł Mikołajczak

WPROWADZENIE DO STL

Recenzent: Jakub Smołka
Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedziba w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-28-2

SPIS TREŚCI

PRZEDMOWA VII

1. WIADOMOŚCI WSTĘPNE 1
1.1. Wstęp 2
1.2. Programowanie ogólne (generyczne) 2
1.3. Rys historyczny 3
1.4. Elementy biblioteki STL 3
1.5. Przykłady użycia elementów biblioteki STL 5

2. KONTENERY 17
2.1. Wstęp 18
2.2. Kontenery sekwencyjne 21
2.3. Kontenery sekwencyjne - vector 24
2.4. Kontenery sekwencyjne - deque 35
2.5. Kontenery sekwencyjne - list 40
2.6. Kontenery asocjacyjne 56
2.7. Kontenery asocjacyjne - set 57
2.8. Kontenery asocjacyjne - multiset 64
2.9. Kontenery asocjacyjne - map 64
2.10. Kontenery asocjacyjne - multimap 69

3. KONTENERY I KLASY SPECJALNE 79
3.1. Wstęp 80
3.2. Adaptator kontenerów stack 80
3.3. Adaptator kontenerów queue 84
3.4. Adaptator kontenerów priority_queue 86
3.5. Klasa bitset 89
3.6. Klasa vector<bool> 92
3.7. Klasa complex 94

4. ITERATORY 105
4.1. Wstęp 106
4.2. Iteratory wejściowe 112
4.3. Iteratory wyjściowe 113
4.4. Iteratory postępujące 113
4.5. Iteratory dwukierunkowe 114

4.6. Iteratory dostępu swobodnego 115
4.7. Dodatkowe operacje na iteratorach 117

5. ALGORYTMY 121
5.1. Wstęp 122
5.2. Algorytmy niemodyfikujące 127
5.3. Algorytmy modyfikujące 135
5.4. Algorytmy usuwające 138
5.5. Algorytmy mutujące 144
5.6. Algorytmy sortujące 148
5.7. Algorytmy przeznaczone dla zakresów posortowanych 155
5.8. Algorytmy numeryczne 160

6. OBIEKTY FUNKCYJNE 165
6.1. Wstęp 166
6.2. Opis obiektów funkcyjnych 173
6.3. Zastosowania obiektów funkcyjnych unarnych 178
6.4. Zastosowania obiektów funkcyjnych binarnych 181

7. ŁAŃCUCHY I KLASA STRING 187
7.1. Wstęp 188
7.2. Łańcuchy znakowe w stylu języka C 188
7.3. Łańcuchy w stylu języka C++ 192
7.4. Operacje i metody klasy string 197
7.5. Kontenery i klasa string 203

BIBLIOGRAFIA 211

PRZEDMOWA

Język C++ jest uznawany za jeden z najlepszych języków programowania. Powstał w Bell Labs (USA), gdzie wspaniały programista Bjarne Stroustrup stworzył go na początku lat osiemdziesiątych. W ciągu ostatnich 30 lat język C++ podlegał licznym modyfikacjom. Istotnym elementem rozwoju języka C++ było dołączenie do standardu bardzo elastycznej i o dużych możliwościach biblioteki szablonów, znanej pod nazwą Standardowej Biblioteki Szablonów (ang. *Standard Template Library*, STL). Włączenie tej biblioteki do standardu języka nastąpiło w 1997 roku. Biblioteka szablonów STL umożliwiła realizowanie nowego paradygmatu programowania - **programowania uogólnionego** (ang. *generic programming*). STL początkowo powstała, jako niezależna biblioteka. Jej autorami byli Alexander Stepanov (z Hewlett-Packard) i Meng Lee (z Silicon Graphics). Jak sama nazwa wskazuje, biblioteka ta jest bazowana na wzorcach, a nie na właściwościach obiektowych. Dzięki czemu możliwe jest używanie jej właściwości z użyciem obiektów wszystkich typów, również wbudowanych. STL zatem dostarcza struktury generycznych komponentów C++, które współpracują bez powiązań. To rozwiązanie pozwala na wykorzystywanie elementów STL tworząc zarówno programy obiektowe jak i generyczne (uogólnione). Celem twórców tej biblioteki jest przede wszystkim elastyczność i wydajność poszczególnych jej komponentów. Formalnie STL jest częścią biblioteki standardowej C++ w standardzie ISO.

W niniejszym skrypcie opisujemy elementy STL dość szczegółowo, ale nie wyczerpująco. Trudność w przyswojeniu sobie elementów STL nie polega na jej skomplikowaniu, a tylko w dużej ilości nowych pojęć. Istnieje szeroko dostępna literatura przedmiotu, szczegółowy opis tej biblioteki można znaleźć na stronie SGI, gdzie opisana jest STL.

STL zawiera pięć najważniejszych rodzajów komponentów:

- algorytmy: definiują procedury obliczeniowe
- kontenery: zarządzają zestawami obiektów
- iteratory: są to narzędzia, dzięki któremu poruszamy sie wewnątrz kontenera.
- funktory: opakowują funkcję w obiekt do używania przez inne komponenty
- adaptatory: adaptują komponenty dla dostarczenia innego interfejsu.

STL jest przede wszystkim zbiorem:

- reguł wedle, których należy tworzyć komponenty
- podstawowych komponentów utworzonych wedle tych reguł

Należy wyraźnie zaznaczyć, że skrypt nie jest instrukcją obsługi STL ani nie stanowi wyczerpującej dokumentacji tej biblioteki. Skrypt ma na celu wprowadzenie czytelnika w fascynujący świat praktycznego wykorzystania elementów STL do tworzenia aplikacji. Czytelnik z łatwością znajdzie potrzebną dokumentację STL na stronach internetowych lub w odpowiednich monografiach. Staram się promować polską terminologię. Dla wygody Czytelnika, zwykle zmuszonego korzystać z dokumentacji w języku angielskim, w przypadku użyciu terminu, który nie jest powszechnie znany, umieszczam w nawiasach jego angielski odpowiednik.

Biblioteka standardowa C++ to zestaw klas oraz interfejsów znacznie rozszerzających język C++. Nie jest ona jednak łatwa do przyswojenia. Niniejszy skrypt omawia wstępnie zasadnicze komponenty biblioteki, jak również stara się przystępnie wyjaśnić złożone zagadnienia; prezentuje praktyczne szczegóły programowania, niezbędne do skutecznego zastosowania omawianej biblioteki w praktyce. W skrypcie znajdują się liczne przykłady działającego kodu źródłowego.

Skrypt opisuje aktualną wersję biblioteki standardowej C++, w tym jej najnowsze elementy dołączone do pełnego standardu języka ANSI/ISO C++. Opis skoncentrowany jest na kontenerach, iteratorach, obiektach funkcyjnych oraz algorytmach STL. W skrypcie są również szczegółowe opisy kontenerów specjalnych, łańcuchów znakowych, klas numerycznych.

Należy zauważyć, że stosowanie technik zawartych w STL nie jest najlepszym przykładem programowania obiektowego, w zamian realizować możemy tak zwane programowanie ogólne (*programowanie generyczne).* Podstawą programowania ogólnego jest maksymalne wykorzystywanie szablonów. W przeciwieństwie do kodów realizujących programowanie obiektowe (konstrukcja klasy), koncepcja biblioteki STL oparta jest na rozdzieleniu danych i metod (operacji). Dane przechowywane są w kontenerach a operacje na danych wykonywane są przy pomocy uogólnionych algorytmów. Operując na danych algorytmy wykorzystują iteratory. Dzięki rozdzieleniu danych i algorytmów mamy możliwość operowania dowolnego algorytmu na dowolnym kontenerze, niezależnie od typu danych. Jest to istota programowania ogólnego. Kontenery z danymi oraz działające na tych danych algorytmy są ogólne, obsługują dowolne typy oraz klasy.

Zgodnie, z B. Stroustrupem przyjmiemy następującą definicję:

> *Programowanie ogólne*: pisanie kodu, który może działać z różnymi typami przekazywanymi do niego, jako argumenty, pod warunkiem, że typy te spełniają określone wymagania syntaktyczne i semantyczne.

Wszystkie zamieszone przykłady programów komputerowych były testowane i starannie sprawdzone. Kody źródłowe opracowane były w środowisku MS Windows (XP i Windows 7) w systemie Borland C++ Builder 6. Są one przenaszalne, wybrane programy były także sprawdzane w systemie C++ QT oraz na C++ Dev.

ROZDZIAŁ 1

WIADOMOŚCI WSTĘPNE

1.1. Wstęp 2
1.2. Programowanie ogólne (generyczne) 2
1.3. Rys historyczny 3
1.4. Elementy biblioteki STL 3
1.5. Przykłady użycia elementów biblioteki STL 5

1.1. Wstęp

Wielu programistów uważa, że jednym z najważniejszych rozszerzeń dodanych do języka C++ jest biblioteka standardowych wzorców – STL (w polskiej literaturze przedmiotu spotkamy także określenie *standardowa biblioteka szablonów*, określenie angielskie: Standard Template Library). Biblioteka STL została opracowana przez Alexandra Stepanowa i Menga Lee, pracowników amerykańskiej firmy Hewlett Packard.

Biblioteka STL zawiera szablony klas oraz funkcje implementujące. W zestawie szablonów między innymi mamy kontenery, iteratory, obiekty funkcyjne oraz algorytmy. Elegancka biblioteka STL umożliwiła realizację kolejnego paradygmatu programowania – programowania ogólnego (ang. *generic programming*). W dużym uproszczeniu możemy powiedzieć, że programowanie ogólne polega na wykorzystywaniu wzorców (szablonów).

1.2. Programowanie ogólne (generyczne)

Istnieje wiele paradygmatów programowania. Aby realizować wybrany paradygmat, programista ma do wyboru wiele języków programowania. W paradygmacie programowania proceduralnego (język C), podstawowym elementem programu jest funkcja. Funkcje działają niezależnie od danych. Możemy implementować na przykład funkcję realizującą obliczanie pierwiastka kwadratowego z liczby. Odpowiednio napisana funkcja oblicza pierwiastek kwadratowy z przekazanego argumentu do tej funkcji. Mówimy, że funkcja jest sparametryzowana wartościami. Ponieważ istnieje precyzyjna kontrola typów zazwyczaj musimy pisać oddzielne wersje funkcji dla różnych typów danych (np. dla typu **int** oraz typu **double**).

Realizując paradygmat programowania obiektowego (język C++), łączymy w klasie dane i metody (funkcje). Stosując proste techniki, metody są parametryzowane wartościami (nawet w przypadku stosowania przeciążenia funkcji). Dopiero wykorzystanie szablonów (język C++) pozwala na parametryzowanie wartościami oraz typami. Dzięki temu możemy tworzyć kod klasy, która będzie w stanie obsłużyć żądany typ.

Należy zauważyć, że stosowanie technik zawartych w bibliotece STL nie jest najlepszym przykładem programowania obiektowego, w zamian realizować możemy tak zwane programowanie ogólne (*programowanie generyczne).* Podstawą programowania ogólnego jest maksymalne wykorzystywanie szablonów. W przeciwieństwie do kodów realizujących programowanie obiektowe (konstrukcja klasy), koncepcja biblioteki STL oparta jest na rozdzieleniu danych i metod (operacji). Dane przechowywane są w kontenerach a operacje na danych wykonywane są przy pomocy uogólnionych algorytmów. Operując na danych algorytmy wykorzystują iteratory. Dzięki rozdzieleniu danych i algorytmów mamy możliwość operowania dowolnego algorytmu na dowolnym kontenerze, niezależnie od typu danych.

Jest to istota programowania ogólnego. Kontenery z danymi oraz działające na

tych danych algorytmy są ogólne, obsługują dowolne typy oraz klasy.
Zgodnie, z B. Stroustrupem przyjmiemy następującą definicję:

> *Programowanie ogólne*: pisanie kodu, który może działać z różnymi typami przekazywanymi do niego, jako argumenty, pod warunkiem, że typy te spełniają określone wymagania syntaktyczne i semantyczne.

Istnieje wiele istotnych różnic pomiędzy programowaniem obiektowym i ogólnym. Bardzo ważny jest fakt, że wyboru funkcji do wywołania kompilator dokonuje w czasie kompilacji, natomiast w programowaniu obiektowym wywołanie to odbywa się w czasie wykonywania programu. *Programowanie ogólne* realizowane jest wykorzystując szablony, wywoływanie metod odbywa się w czasie kompilacji, *programowanie obiektowe* wykorzystuje hierarchię klas oraz wirtualne funkcje i klasy, wywoływanie metod odbywa się w czasie wykonywania programu.

1.3. Rys historyczny

Bibliotekę tworzyli Alex Stepanow i Meng Lee, pracownicy Hewlett-Packard Laboratories, duży wkład miał także David Musser.

Pod koniec lat siedemdziesiątych Aleksander Stepanow zauważył, że ilość użytecznych w praktycznych zastosowaniach algorytmów jest niewielka. Co więcej analizując wybrane algorytmy zauważył, że ich implementacja nie zależy od użytej struktury danych. Np. dla algorytmów sortujących nie jest istotne (rozważając wydajność) czy dane przechowywane są w tablicach czy listach powiązanych. W 1985 roku Stepanow zrealizował ogólną bibliotekę, (ang. *generic library*) dla języka ADA. Był to znaczny sukces, więc poproszono go o zrealizowanie takiej biblioteki dla języka C++. Niestety w tym czasie (1987 rok) nie zaimplementowano jeszcze szablonów. W 1988 roku Stepanow został zatrudniony w HP Laboratories.

Duże zainteresowanie stosowaniem biblioteki STL nastąpiło między innymi dzięki pracy Bjarne Stroustrupa, opublikowanej pod tytułem *Parametrized Types for C++* w Proceedings of the USENIX C++ Conference, konferencja odbyła się w Denver w 1998 roku. Komitet ISO/ANSI zalecił włączenie tej biblioteki do standardu języka C++.

Najbardziej oczywisty opis podstaw biblioteki zamieszczony jest w artykule pt. „The Standard Template Library" napisanym przez Alexandra Stepanowa i Menga Lee (copyright © 1994 Hewlett-Packard Company).

1.4. Elementy biblioteki STL

Zgodnie z opisem podanym przez Alexandra Stepanov'a i Menga Lee (A.Stepanov, M.Lee, The Standard Template Library, Hewlett-Packard Company, 1994) biblioteka STL dostarcza zbiór dobrze skonstruowanych generycznych składników (ang. *well structured generic C++ components*), które

działają razem bezproblemowo (ang. *in a seamless way*). Biblioteka STL dostarcza narzędzi do zarządzania kolekcjami danych przy użyciu wydajnych algorytmów. Biblioteka STL zawiera szablony klas ogólnego przeznaczenia, (czyli daje możliwość obsługi dowolnych typów danych) oraz funkcje implementujące. Dzięki bibliotece STL mamy możliwość realizacji kolejnego paradygmatu programowania – możemy realizować ***programowanie generyczne*** (ang. *generic programming*). W odróżnieniu od programowania obiektowego, które koncentruje się na danych i konstruowaniu typów użytkownika, w programowaniu generycznym główny nacisk kładzie się na algorytmy.

Nicolai Josuttis w swoim doskonałym podręczniku o STL (*The C++ Standard Library: A Tutorial and Reference, 1999*) za najważniejsze składniki STL uważa kontenery, iteratory i algorytmy.

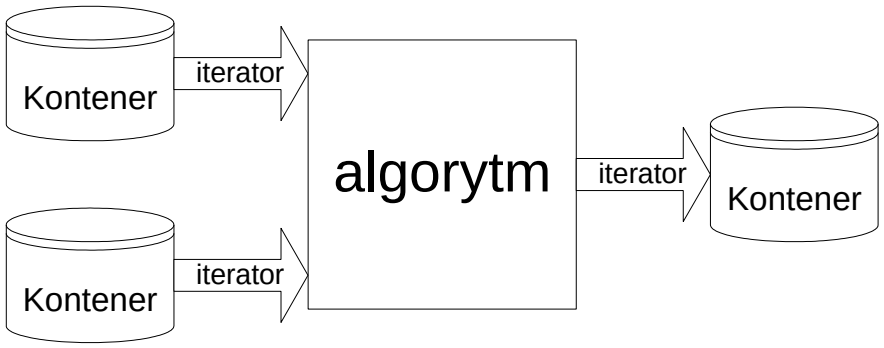
Krótki opis podstawowych elementów STL jest następujący.

Kontenery służą do obsługi kolekcji obiektów, istnieje wiele rodzajów kontenerów. Najpopularniejszym kontenerem jest klasa **vector**, ta klasa zawiera definicję tablicy dynamicznej. Klasa **vector**, podobnie jak klasa **list** czy klasa **deque** są nazywane kontenerami sekwencyjnymi (ang. *sequence containers*). Sekwencyjny kontener jest takim typem kontenera, który organizuje skończony zbiór obiektów tego samego typu, powiązanych w ściśle liniowym porządku. Istnieją też kontenery stowarzyszone, inna nazwa kontenery asocjacyjne (ang. *associative containers*). Przykładem takiej klasy jest klas **set** czy **map.** Tego typu kontenery umożliwiają efektywne operowanie wartościami na podstawie kluczy. Iteratory są uogólnieniem wskaźników. Dzięki iteratorom (ang. *iterators*) programista może pracować z różnymi strukturami danych (kontenerami) w jednorodny i uporządkowany sposób. Dzięki nim można poruszać się po zawartości kontenera w podobny sposób jak dzięki wskaźnikom przemieszczamy się w obrębie tablicy. Iteratory zachowują się bardzo podobnie do wskaźników. Możliwa jest ich inkrementacja i dekrementacja. Iteratory deklaruje się za pomocą typu **iterator** zdefiniowanego w różnych kontenerach. Istnieje pięć różnych typów iteratorów. Algorytmy operują na kontenerach. Dzięki iteratorom mogą operować na elementach kontenerów niezależnie od struktury danych. Algorytmy używają szablonów, dzięki czemu można stosować generyczne typy danych. Dzięki iteratorom określamy zakres przetwarzanych danych oraz określamy miejsce, do którego należ wysłać wyniki. Algorytmy pozwalają w sposób stosunkowo prosty na przykład sortować dane bez względu na typ - algorytm **sort()** sortuje zakres bez względu czy mamy zbiór liczb całkowitych czy zbiór napisów. Aby uzyskać dostęp do algorytmów STL należy włączyć do programu plik nagłówkowy **<algorithm>.** Algorytmy w STL dzielą się na cztery grupy:

- Operacje sekwencyjne niezmieniające wartości
- Operacje sekwencyjne zmieniające wartości
- Operacje sortowania i pokrewne
- Uogólnione operacje numeryczne

Algorytmy przeznaczone do przetwarzania danych numerycznych najczęściej

wymagają pliku nagłówkowego **<numeric>.**



Rys. 1.1. Relacje między głównymi składnikami STL (M. Josuttis)

Relacje pomiędzy głównymi składnikami STL pokazane są na rysunku 1.1. Z kolei A.Stepanov i M. Lee wyróżniają pięć głównych składników w STL:

- Algorytmy (ang. *algorithm*)
- Kontenery (ang. *containers*)
- Iteratory (ang. *iterator*)
- Obiekty funkcyjne (ang. *function object*)
- Adaptatory (ang. *adaptor*)

Obiekty funkcyjne są to klasy, dla których zdefiniowano operator (). W przypadku definiowania obiektu funkcyjnego korzysta się z szablonów zdefiniowanych w pliku **<functional>.** W wielu przypadkach obiekty funkcyjne są wykorzystywane, jako wskaźniki do funkcji. Jednym z najczęściej używanych obiektów funkcyjnych jest **less,** dzięki któremu możemy określić czy dany obiekt jest mniejszy od innego. Adaptatory są szablonami klas, dzięki którym można mapować interfejsy. Mówiąc prościej, dzięki adaptatorom wykorzystujemy istniejące klasy do tworzenia nowych klas z nowymi właściwościami. Klasycznym przykładem jest tworzenie stosu wykorzystując klasy **vector, list** lub **deque**.

1.5. Przykłady użycia elementów biblioteki STL

Biblioteka STL jest bardzo duża, składnia w niej stosowana, przynajmniej na początku wydaje się bardzo skomplikowana, ale praktyczne stosowanie elementów tej biblioteki nie jest zbyt trudne. Biblioteka STL udostępnia zestaw wzorców reprezentujących kontenery, iteratory, obiekty funkcyjne oraz algorytmy.

Jako przykład pokażemy użycie jednego z najbardziej ogólnych kontenerów, jakim jest **vector**. Kontener, podobnie jak tablica pozwala na przechowywanie grupy wartości i podobnie jak tablica może obsługiwać tylko wartości tego samego typu. Klasa (kontener) **vektor** obsługuje tablice dynamiczne (należy zauważyć, że nazwa kontenera jest trochę myląca, klasa **vektor** ma niewiele wspólnego z matematycznym pojęciem wektora). Obiekt szablonu **vector** przechowuje zestaw wartości tego samego typu o dostępie swobodnym, wobec tego możemy operować poszczególnymi wartościami przy pomocy indeksu. Po utworzeniu obiektu **vector** przy pomocy przeciążonego operatora [] oraz indeksu uzyskujemy dostęp do poszczególnych elementów, tak samo jak w przypadku tablic. Prezentowany program pokazuje sposób tworzenia obiektu szablonu **vector**. W programie tworzymy wektor (obiekt szablonu **vector**) liczb całkowitych, wprowadzimy pięć liczb całkowitych przy pomocy klawiatury, a następnie program wyświetli wprowadzone liczby. Należy przypomnieć, że klasa **vector** obsługuje tablice dynamiczne, w miarę potrzeby rozmiar kontenera (tablicy) będzie powiększany. Specyfikacja szablonu dla klasy **vector** ma postać:

template <class T, class Allocator = allocator<T>> class vector

W tej specyfikacji **T** oznacza typ, jaki chcemy zaimplementować, **Allocator** oznacza alokator (alokator jest obiektem klasy **allocator**, służącym do zarządzania pamięcią), domyślną wartością jest alokator standardowy.

W celu utworzenia obiektu klasy **vector** musimy dołączyć plik **vector** oraz zadeklarować kolekcję (obiekt szablonu **vector**), co w praktyce sprowadza się do naśladowania posługiwania się zwykłymi szablonami klas.

Wydruk 1.1. Przykład użycia wzorca **vector,** typ **int**

```
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;
const int ILE = 5 ;
int main()
{ vector<int>liczby(ILE);
 for (int i=0; i < ILE; i++)
   { cout << "liczba " << i+1 << " = " ;
     cin >> liczby[i];
   }
cout << "wprowadzono liczby : " << endl;
for (int i=0; i < ILE; i++)        cout << liczby[i] << "  "  ;
 getche();
 return 0;
}
```

Po uruchomieniu programu mamy następujący wydruk:

```
liczba 1 = 1
liczba 2 = 3
liczba 3 = 5
liczba 4 = 7
liczba 5 = 9
wprowadzono liczby :
1 3 5 7 9
```

Plik nagłówkowy potrzebny do obsługi wektorów dołączamy poleceniem :

```
#include <vector>
```

Deklarując kolekcję (wektor, obiekt szablonu **vector**) musimy określić typ elementów:

```
vector < int > liczby( ILE );
```

Został utworzony konkretny rodzaj kontenera, w tym przypadku jest to wektor o nazwie **liczby**, którego elementy są typu **int**, kontener może początkowo posiadać 5 elementów (zmienna **ILE**). Jak widać, pominięto drugi argument (alokator) w deklaracji szablonu klasy. W takim przypadku szablon kontenera użyje domyślnie klasy **allocator<T>**. Klasa ta zarządza dynamicznie pamięcią, wykorzysta w zwykły sposób operatory **new** i **delete**.

Wektor **liczby** jest inicjalizowany liczbami całkowitymi wprowadzonymi z klawiatury. Zastosowano klasyczną składnie operacji na tablicach, wykorzystując indeksy. W następującym fragmencie programu trudno zorientować się, że obsługujemy obiekty klasy **vector** a nie zwykłe tablice:

```
for (int i=0; i < ILE; i++)
    { cout << "liczba " << i+1 << " = " ;
      cin >> liczby[i];
    }
```

Główną zaletą biblioteki STL jest fakt, że nie musimy zajmować się szczegółami implementacji struktur danych i zarządzaniem pamięcią podczas przetwarzania kolekcji. W kolejnym przykładzie utworzymy obiekt **vector** dla typu **string.**

Wydruk 1.2. Przykład użycia wzorca **vector,** typ **string**

```
#include <iostream>
#include <vector>
```

```
#include <string>
#include <conio.h>
using namespace std;
const int ILE = 5 ;

int main()
{ vector<string> imiona(ILE);              // wektor typu string
 for (int i=0; i < ILE; i++)
   { cout << "imie " << i+1 << " = " ;
     getline(cin, imiona[i]);
   }
  cout << "wprowadzono napisy : " << endl;
  for (int i=0; i < ILE; i++)
      cout << "     " << imiona[i] << endl ;
 getche();
 return 0;
}
```

Wydruk z programu ma następującą postać:

```
imie 1 = Ala
imie 2 = Ola
imie 3 = Ziuta
imie 4 = Buba
imie 5 = Lola
wprowadzono napisy :
 Ala
 Ola
 Ziuta
 Buba
   Lola
```

W programie, aby utworzyć kolekcję imion (typ **string**) musimy włączyć dwa pliki:

```
#include <vector>
#include <string>
```

Definicja wektora napisów ma postać:

```
vector<string> imiona(ILE);
```

Pętla do wprowadzania napisów (w naszym przypadku imion) ma postać:

```
for (int i=0; i < ILE; i++)
```

```
{ cout << "imie " << i+1 << " = " ;
    getline(cin, imiona[i]);
}
```

Wszystkie kontenery biblioteki STL udostępniają potrzebne metody. Do najważniejszych metod możemy zaliczyć metodę **size(),** która umożliwia określenie liczby elementów umieszczonych w kontenerze oraz dwie metody **begin()** oraz **end().** Metoda **begin()** zwraca iterator (iterator jest obiektem bardzo przypominającym w działaniu wskaźnik, umożliwia poruszanie się po zawartości kontenera, tak jak wskaźnik pozwala przemieszczać się po zawartości tablicy). Szablon klasy **vector** ma dodatkowo metodę **push_back(),** która dodaje elementy na koniec wektora. Kolejny program ilustruje wykorzystanie metody **push_back().**

Wydruk 1.3. Przykład użycia wzorca **vector,** metody kontenerów

```
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;

int main ()
{ vector<int> v;     // utworzenie pustego wektora(kontenera), typ: int
  int we;
  cout << "podaj liczby calkowite, q konczy " << endl;
  while (cin >> we)   // wprowadzanie liczb z klawiatury
    v.push_back (we); // umieszczanie w kontenerze
  int n = v.size();
  cout << "wprowadzono liczby : \n" ;
  for (int i = 0; i < n; i++)
    cout << v[i] << "  " ;
  getche();
  return 0;
}
```

Po uruchomieniu programu, mamy wydruk:

```
podaj liczby całkowite, q konczy
2 4 6 q
wprowadzono liczby :
2   4   6
```

W pokazanym programie wykorzystano metodę **push_back(),** która dodaje element na koniec wektora:

```
vector<int> v;      // utworzenie pustego wektora(kontenera), typ: int
```

```
int we;
cout << "podaj liczby calkowite, q konczy " << endl;
while (cin >> we)                    // wprowadzanie liczb z klawiatury
      v.push_back (we);              // umieszczanie w kontenerze
```

Każde wykonanie pętli wymusza dodanie nowego elementu do wektora **v**. Zauważmy, że nie określono na początku liczby elementów, jakie ma przechować wektor **v**. Wektor zwiększa swój rozmiar automatycznie, tzn. pamięć jest przydzielana automatycznie. Dopóki program będzie posiadał dostęp do odpowiedniej ilości pamięci, będzie powiększał swój rozmiar. W programie wykorzystano także metodę **size()** do wyliczenia ilości elementów umieszczonych w kontenerze **v**. Prawdziwe korzyści wynikające ze stosowania szablonów STL możemy zilustrować programem sortującym tablice liczb całkowitych (wydruk 1.4).

Wydruk 1.4. Przykład użycia wzorca **vector,** sortowanie

```
#include <iostream>
#include <vector>
#include <algorithm>   // dla sort()
#include <conio.h>

using namespace std;

int main ()
{
 vector<int> v;     // utworzenie pustego wektora(kontenera), typ: int
  int we;
  cout << "podaj liczby calkowite, q konczy " << endl;
  while (cin >> we)                    // wprowadzanie liczb z klawiatury
     v.push_back (we);                 // umieszczanie w kontenerze
  int n = v.size();
  cout << "wprowadzono liczby : \n" ;
  for (int i = 0; i < n; i++)
     cout << v[i] << "  " ;
  sort(v.begin(), v.end());
  cout << "\nposortowane liczby : \n" ;
  for (int i = 0; i < n; i++)
     cout << v[i] << "  " ;
  getche();
  return 0;
  }
```

Jak wiadomo w standardowej bibliotece języka C oraz C++ mamy funkcje **qsort(),** która może być wykorzystana do sortowania elementów. Zamiast funkcji **qsort()** wykorzystamy procedurę **sort()** z biblioteki STL. Jest ona

przykładem jednego z wielu algorytmów, jakie dostarcza nam biblioteka STL. Procedura **sort()** jest generyczna, tzn. może sortować wiele różnych typów danych przechowywanych w kontenerach. Algorytm **sort()** wykorzystuje iteratory.

Po uruchomieniu programu mamy następujący wydruk:

```
podaj liczby całkowite, q konczy
1 5 6 7 4 3 2 q
wprowadzono liczby :
1 5 6 7 4 3 2
posortowane liczby :
1 2 3 4 5 6 7
```

Operacje sortowania na elementach kontenera wykonywane są bardzo często przy pomocy iteratorów. Przypominamy, że iteratory są obiektami, które umożliwiają przeglądanie zawartości kontenerów. Każdy iterator reprezentuje pozycję w kontenerze. Należy zwrócić uwagę, że **begin()** zwraca iterator reprezentujący pozycje pierwszego elementu w kontenerze, a **end()** zwraca iterator reprezentujący pozycję za ostatnim elementem w kontenerze. Sortowanie elementów kontenera wykonane jest dzięki instrukcji:

```
sort( v.begin(), v.end() );
```

Algorytm **sort()** wymaga dołączenia pliku nagłówkowego:

```
#include <algorithm>      // dla sort()
```

Algorytm **sort()** przyjmuje jako argumenty dwa iteratory, które definiują sortowany zakres.

Programiści języka C oraz C++ wiedzą, że manipulowanie napisami (łańcuchami znakowymi) nie jest proste. W języku C oraz C++ nie ma standardowego typu napisowego (ang. *string*). W tych językach napis traktuje się jak ciąg znaków (istnieje typ wbudowany – znakowy). Łańcuchy to tablice składające się ze znaków. Możemy w następujący sposób zdefiniować tablicę znakową o nazwie **wyraz**:

```
char wyraz [ ] = { 'W', 'i', 't', 'a', 'j'} ;
```

Ponieważ jawnie nie podano wielkości tablicy, kompilator C automatycznie obliczy wielkość tablicy i zarezerwuje pamięć do przechowania 5 znaków. Ponieważ w wielu aplikacjach musimy przetwarzać ciągi znaków, opracowano całą rodzinę funkcji do manipulowania nimi. Większość tych funkcji umieszczona jest w pliku <string.h> oraz <cstring.h>. Bardzo często do

przetwarzania tablicy znaków musimy znać liczbę znaków. Każdorazowe zliczanie liczby znaków w napisie jest niepraktyczne, mamy metody pozwalającej obsługiwać tablice znakowe bez konieczności ustalania liczby znaków. Metoda polega na umieszczeniu znaku specjalnego na końcu łańcucha.

Jest to znak **null** zapisywany jako '\0' , przykład użycia pokazany jest poniżej:

```
char wyraz [ ] = { 'W', 'i', 't', 'a', 'j', '\0'} ;
```

Tablice znakowe można inicjalizować także w następujący sposób:

```
char wyraz [ ] = { "Witaj"} ;
```

lub:

```
char wyraz [ ] = "Witaj" ;
```

Można utworzyć wskaźnik na napis i wykonać inicjalizację:

```
char imie[10] = "napis";
char *wsk = imie;
```

lub zainicjować wskaźnik literałem:

```
char *wsk = "inny_napis";
```

W C++ można dynamicznie reprezentować ciąg znakowy w następujący sposób:

```
char * imie = new char [ n ];
```

W tym przypadku powstaje dynamicznie zaalokowana tablica znaków , która może być zainicjalizowana do długości wskazanej przez **n.**

Użycie klasy ciągu znaków (ciągu tekstowego) zamiast tablicy znakowej bądź wskaźników było doskonałym pomysłem twórców biblioteki STL. Klasa ciągu tekstowego w bibliotece STL:

```
std:: string
```

pozwala na proste manipulowanie napisami i jest bardzo odporna na błędy. W celu skorzystania z klasy **string** należy do programu włączyć plik nagłówkowy <string>.

Klasa **string** ma wiele metod, kilka konstruktorów oraz duży zestaw przeciążonych operatorów. Prosty przykład pokazujący wykorzystanie klasy **string** do manipulowania napisami pokazuje poniższy wydruk programu.

Wydruk 1.5. Przykład użycia klasy **string**

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;

int main()
{ string n1 ("w kompilatorze C++");
  string n2 ( "korzystamy z klasy string");
  string n3 ( "mamy bibliteke STL");
  string n12;
  n12 = n1 + n2;
  cout << n12 << endl;
  getche();
  return 0;
}
```

Wynikiem wykonania programu jest komunikat:

w kompilatorze C++ korzystamy z klasy string

Prosta inicjalizacja i przypisanie ciągu tekstowego do zwykłego obiektu **n1** ma postać:

string n1 ("w kompilatorze C++");

Łączenie dwóch napisów jest proste:

n12 = n1 + n2;

Klasa **string** posiada wiele metod. Obiekt STL **string** zawiera miedzy innymi metodę **find(),** dzięki której możemy wyszukiwać znak lub sekwencję znaków w napisie. Kolejny program ilustruje wykorzystanie metody **find().**

Wydruk 1.6. Przykład użycia klasy **string**, metoda **find()**

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;

int main()
{ string n1 ( " w kompilatorze C++");
  string n2 ( " mamy bibliteke STL");
  string n12;
  n12 = n1 + n2;
```

```
    cout << n12 << endl;
    size_t  poz = n12.find("STL",0);
    if (poz != string:: npos)
        cout << "szukany tekst na pozycji " << poz;
    else
        cout <<" nie ma takiego tekstu" << endl;
    getche();
    return 0;
}
```

Wynikiem działania programu jest komunikat:

```
w kompilatorze C++ mamy bibliotekę STL
szukany tekst na pozycji 35
```

Pokazano prosty sposób wykorzystania funkcji **find()** aby określić czy w tekście znajduje się skrót „STL”. Wyszukiwanie realizowane jest we fragmencie programu:

```
size_t  poz = n12.find("STL",0);
 if (poz != string:: npos)
      cout << "szukany tekst na pozycji " << poz;
 else
      cout <<" nie ma takiego tekstu" << endl;
```

W klasie **string** umieszczone są specjalne definicje, między innymi definicja typu **size_t**. Definicja **size_t** oznacza typ całkowitoliczbowy bez znaku określający rozmiar wartości oraz indeksów. Funkcja **find()** szuka napisu ”STL” w łańcuchu **n12** i znalezioną wartość umieszcza w zmiennej **poz** (jest ona typu **size_t**). W klasach łańcuchów znakowych istnieje wiele funkcji dzięki którym możemy poszukiwać określonego ciągu znaków w napisach. W przypadku, gdy nie zostanie znaleziony wyspecyfikowany znak lub ciąg znaków, funkcje te zwracają wartość **string::npos**. Formalnie jest to wartość -1, ale należy stosować typ **string::size_typ** lub równoważny typ **string:size_t**, (nie należy używać typu **int**). W pokazanym fragmencie programu wynik poszukiwania funkcji **find()** jest porównywany z wartością **npos** (niepowodzenie). Gdy funkcja **find()** nie zwraca wartości **npos**, oznacza to, że wyspecyfikowany ciąg znaków został znaleziony, a zwrócona wartość wskazuje położenie tego ciągu. Pokazana funkcja **find():**

```
find("STL",0)
```

przyjmuje dwa parametry: - pierwszy oznacza znak lub ciąg znaków, drugi argument oznacza miejsce w badanym ciągu od którego funkcja ma zacząć przeszukiwanie. W naszym przypadku zaczynamy poszukiwanie od początku łańcucha.

Wykorzystanie algorytmów STL daje duże możliwości wygodnego manipulowania napisami. Na koniec pokażemy program wykonujący klasyczne zadanie - zamiana ciągu tekstowego pisanego małymi literami na ciąg pisany dużymi literami. W tym celu wykorzystamy algorytm **transform().** Wykorzystanie algorytmu **transform()** z biblioteki STL wymaga dołączenia pliku nagłówkowego

#include<algorithm>

Wydruk 1.7. Przykład użycia klasy **string**, algorytm **transform()**

```
#include<string>
#include<iostream>
#include<algorithm>
#include <conio.h>
using namespace std;
int main()

{
 string n;
 cout << "prosze napisac tekst:"<<endl;
 getline(cin, n);
 cout << endl;
 transform (n.begin(), n.end(), n.begin(), toupper);
 cout << "wynik konwersji:"<< endl;
 cout << n << endl;

 getche();
 return o;
}
```

Po uruchomieniu programy otrzymamy następujący wydruk:

```
prosze napisac tekst:
program w C++

wynik konwersji:
PROGRAM W C++
```

W pliku nagłówkowym <algorithm> znajduje się około 60 algorytmów. Te algorytmy działają na sekwencjach zdefiniowanych przez pary iteratorów. Wyróżniamy algorytmy modyfikujące sekwencje oraz algorytmy niemodyfikujące. Algorytm **transform()** jest algorytmem modyfikującym. W bibliotece STL mamy dwie wersje tego algorytmu – wersja pierwsza wymaga przekazania czterech argumentów, wersja druga wymaga przekazania pięciu argumentów. W naszym programie korzystamy z wersji czteroargumentowej. Dwa pierwsze argumenty to iteratory, które określają zakres przekształcanych

elementów. Trzeci argument jest iteratorem wskazującym miejsce do którego przekopiowane będą wyniki. Czwartym argumentem jest funktor (obiekt funkcyjny), który wykona żądaną transformację. Takim funktorem może być np. **sqrt,** w tym przypadku zostanie wyliczony pierwiastek kwadratowy z każdego wyspecyfikowanego elementu. W naszym programie wykorzystany został funktor **toupper**:

```
transform (n.begin(), n.end(), n.begin(), toupper);
```

Gdyby zaszła konieczność zamiany tekstu pisanego dużymi literami na tekst pisany małymi literami należałoby użyć funktora **tolower**:

```
transform (n.begin(), n.end(), n.begin(), tolower);
```

W tych kilku prostych przykładach wykazaliśmy, że stosowanie konstrukcji biblioteki STL znacznie upraszcza oraz przyspiesza pisanie poprawnych programów. Niewątpliwie stosowanie paradygmatu programowania ogólnego (generycznego) powinno być bardziej popularne, niż to się praktykuje obecnie (rok 2011).

ROZDZIAŁ 2

KONTENERY

2.1. Wstęp 18
2.2. Kontenery sekwencyjne 21
2.3. Kontenery sekwencyjne - vector 24
2.4. Kontenery sekwencyjne - deque 35
2.5. Kontenery sekwencyjne - list 40
2.6. Kontenery asocjacyjne 56
2.7. Kontenery asocjacyjne - set 57
2.8. Kontenery asocjacyjne - multiset 64
2.9. Kontenery asocjacyjne - map 64
2.10. Kontenery asocjacyjne - multimap 69

2.1. Wstęp

Klasy kontenerowe (ang. *container classes*), nazywane kontenerami lub zasobnikami służą do zarządzania zbiorami elementów. Wyróżniamy dwa typy kontenerów:

- Kontenery sekwencyjne
- Kontenery asocjacyjne

Kontenery sekwencyjne są kolekcjami uporządkowanymi, pozycja każdego elementu jest określona w momencie wstawiania i jest niezależna od wartości elementu.
Kontenery asocjacyjne są kolekcjami sortowanymi, pozycja każdego elementu zależy od jego wartości zgodnie z kryterium sortowania. A. Stepanov i M. Lee wyróżniają siedem rodzajów kontenerów, opisane są one w tabeli 2.1.

Tabela 2.1. Kontenery zdefiniowane w STL

Lp.	Typ kontenera	Nazwa kontenera
1	Kontener sekwencyjny	Vector
2		Deque
3		List
4	Kontener asocjacyjny	Set
5		Multiset
6		Map
7		Multimap

Inni autorzy (np. S. Prata, Szkoła programowania, język C++, 2005) wyróżniają 11 kontenerów: **vector, deque, list, set, multiset, map, multimap, queue, priority_queue**, **stack,** oraz **bitset**. Krótkie opisy kontenerów oraz pliki nagłówkowe, które muszą być włączane do programu pokazane są w tabeli 2.2.

Stosując kontenery **bitset** należy pamiętać, że ta klasa nie zachowuje się jak typowa klasa biblioteki STL, należy, więc podejść z dużą starannością do tworzenia obiektów klasy **bitset** i technik operowania tymi obiektami.

Nie ma ściśle zdefiniowanej budowy kontenera. Pojęcie kontenera określa zestaw wymagań, które muszą spełniać wszystkie klasy kontenerowe w STL. Pamiętając, że kontenery przechowują obiekty tego samego typu, mamy możliwość używania, jako przechowywanych obiektów wartości typów wbudowanych jak i obiektów w sensie programowania obiektowego. Wszystkie dane, które są przechowywane w kontenerze są przez niego posiadane. Jeżeli usuwamy kontener, usuwamy także przechowywane w nim dane. Dla obiektów przechowywanych w kontenerach musimy mieć możliwość tworzenia kopii oraz przypisywania do nich wartości.

Dlatego w typach opartych na klasach musi być zdefiniowany konstruktor kopiujący i operator przypisania, nie mogą one być tworzone w prywatnym ani chronionym obszarze dostępu. Wszystkie kontenery posiadają określone właściwości i udostępniają określone operacje. Jak już mówiliśmy, w STL nie jest zdefiniowana ściśle konkretna implementacja żadnego kontenera.

Standard określa jedynie zachowanie i złożoność. W praktyce poszczególne implementacje różnią się jedynie drobnymi szczegółami.

Tabel 2.2. Opisy kontenerów i pliki nagłówkowe

Lp.	Kontener	Opis	Plik
1	vector	Tablica dynamiczna	<vector>
2	deque	Kolejka dwukierunkowa	<deque>
3	list	Lista liniowa	<list>
4	set	Zbiór elementów unikatowych	<set>
5	multiset	Zbiór, elementy nie muszą być unikatowe	<set>
6	map	Przechowuje pary klucz/wartość, klucz powiązany z jedną wartością	<map>
7	multimap	Przechowuje pary klucz/wartość, klucz powiązany z wieloma wartościami	<map>
8	bitset	Zbiór bitów	<bitset>
9	stack	Stos	<stack>
10	queue	Kolejka	<queue>
11	priority_queue	Kolejka priorytetowa	<queue>

Kontenery STL były projektowane zgodnie z ustalonymi zasadami, mają, więc bardzo podobne funkcjonalności. Posiadają też wspólne dla wszystkich funkcje składowe. Te funkcje pokazane są w tabeli 2.3 i 2.4. Biblioteka STL jest rozbudowana i dość skomplikowana, ale wykorzystanie jej składników nie jest zbyt trudne. Przede wszystkim należy określić, jaki typ kontenera będzie najlepszy dla rozwiązania postawionego zadania. Każdy kontener ma swoje zalety. Jeżeli tworzymy obiekty typu tablicowego i chcemy mieć dostęp swobodny do nich oraz nie przewidujemy zbyt wielu operacji wstawiania i usuwania to najlepiej jest zastosować kontener **vector.** Jeżeli przewidujemy wykonywanie dużo operacji wstawiania i usuwania obiektów, wtedy najlepiej jest wykorzystać kontener **list.**

Tabela 2.3. Przeciążone funkcje operatorowe (dla **vector, deque, list, set, multiset, map i multimap**)

Lp.	operator	Opis
1	operator=	Przypisuje jeden kontener do drugiego
2	operator<	Zwraca **true** jeżeli pierwszy kontener jest mniejszy od drugiego, w przeciwnym przypadku zwraca **false**
3	operator<=	Zwraca **true** jeżeli pierwszy kontener jest mniejszy lub równy drugiemu, w przeciwnym przypadku zwraca **false**
4	operator>	Zwraca **true**, jeżeli pierwszy kontener jest większy od drugiego, w przeciwnym przypadku zwraca **false**
5	operator>=	Zwraca **true**, jeżeli pierwszy kontener jest większy lub równy drugiemu, w przeciwnym przypadku zwraca **false**
6	operator==	Zwraca **true**, jeżeli pierwszy kontener jest równy drugiemu, w przeciwnym przypadku zwraca **false**
7	operator!=	Zwraca **true**, jeżeli pierwszy kontener nie jest równy drugiemu, w przeciwnym przypadku zwraca **false**

Za pomocą metod możemy dodawać do kontenera żądane elementy, modyfikować je, usuwać i wykonywać inne potrzebne operacje.

Należy pamiętać, że kontenery są automatycznie powiększane, gdy dodawany jest nowy element i zmniejszane, gdy element jest usuwany. Ta właściwość nie dotyczy kontenera **bitset.** Najczęściej, aby uzyskać dostęp do elementów w kontenerze stosujemy iteratory. Wszystkie kontenery maja metody **begin()** i **end(),** które zwracają iteratory wskazujące na początek i koniec kontenera. Mając ustalony początek kontenera przy pomocy inkrementacji iteratora możemy przeglądać kolejne elementy. Jeżeli w kontenerze są przechowywane dane, to możemy nimi manipulować przy pomocy algorytmów w sensie STL.

Tabela 2.4. Funkcje składowe (**vector, deque, list, set, multiset, map** i **multimap**)

Lp.	Funkcja	Opis
1	Konstruktor domyślny	Wykonuje inicjowanie kontenera, zazwyczaj kontener posiada kilka konstruktorów
2	Konstruktor kopiujący	Wykonuje kopię istniejącego kontenera
3	empty()	Zwraca **true**, jeżeli w kontenerze nie ma elementów, w przeciwnym przypadku zwraca **false**
4	max_size()	Zwraca maksymalną liczbę elementów w kontenerze
5	size()	Zwraca bieżącą liczbę elementów w kontenerze
6	swap()	Wymienia elementy dwóch kontenerów
7	begin()	Zwraca iterator, który wskazuje na pierwszy element kontenera
8	end()	Zwraca iterator, który wskazuje na pozycję po ostatnim elemencie kontenera
9	rbegin()	Zwraca iterator odwrotny dla pierwszego elementu iteracji odwrotnej
10	rend()	Zwraca iterator odwrotny dla pozycji za ostatnim elementem iteracji odwrotnej
11	erase()	Usuwa jeden albo wszystkie elementy z zakresu w kontenerze
12	clear()	Opróżnia kontener
13	insert()	Wstawia kopię elementu do kontenera
14	get_allocator()	Zwraca model pamięci kontenera

2.2. Kontenery sekwencyjne

Kontenery sekwencyjne są kontenerami, które obsługują skończony zbiór elementów, wszystkie muszą być tego samego typu, elementy musza być uporządkowane ściśle liniowo. W STL mamy trzy rodzaje takich kontenerów: **vector**, **list** oraz **deque**. Uporządkowanie liniowe oznacza, że istnieje pierwszy element . Każdy element pomiędzy pierwszy i ostatnim ma ściśle jeden element poprzedzający i jeden element za nim.

Zgodnie z pracą A.Stepanowa i M. Lee kontenery sekwencyjne mają wspólne właściwości przedstawione w tabeli 2.5.

Tabela 2.5. Właściwości kontenerów sekwencyjnych (dla **vector, deque** i **list**)

Lp.	Wyrażenie	Zwracany typ	Opis
1	X(n, t)		Tworzy sekwencje **n** kopii wartości **t**
2	X a(n, t)		Deklaruje sekwencje **n** kopii wartości **t**
3	X a(i, j)		Deklaruje sekwencję **a** inicjalizowaną na podstawie zawartości zakresu (i,j)
4	X(i, j)		Tworzy sekwencję zainicjalizowaną za pomocą wartości zakresu (i,j)
5	a.insert(p, t)	iterator	Wstawia kopię **t** przed **p**, zwracana wartość wskazuje na wstawiana kopie
6	a.insert(p, n, t)	void	Wstawia **n** kopii t przed **p**
7	a.insert(p, i, j)	void	Wstawia kopie elementów z zakresu (i, j) przed **p**
8	a.erase(q)	iterator	Usuwa element, na który wskazuje **q**
9	a.erase(q1, q2)	Niektóre kontenery zwracają następny nieusunięty element	Usuwa elementy z zakresu (q1, q2)
10	a.clear()	void	To samo co **erase(begin(), end())**

STL dostarcza pewną ilość dodatkowych właściwości kontenerów sekwencyjnych, pokazane one są w tabeli 2.6.

Tabela 2.6. Dodatkowe właściwości kontenerów sekwencyjnych

Lp.	Wyrażenie	Zwracany typ	Znaczenie	kontenery
1	a.front()	T&	*a.begin()	vector, list, deque
2	a.back()	T&	*a.(--end())	vector, list, deque
3	a.push_front(t)	void	a.insert(a.begin(), t)	list, deque
4	a.push_back(t)	void	a.insert(a.end(),t)	vector, list, deque
5	a.pop_front()	void	a.erase(a.begin())	list, deque
6	a.pop_back()	void	a.erase(--a.end())	vector, list, deque
7	a[n]	T&	*(a.begin() + n)	vector, deque
8	a.at(n)	T&	*(a.begin() + n)	vector, deque

Kontenery są użyteczne, gdy można w nich umieszczać obiekty i je usuwać. Opracowując składniki STL zwrócono uwagę na wydajność metod i algorytmów. Jak pamiętamy, w informatyce dla porównywania względnej złożoności algorytmów używa się specjalistycznej notacji – jest to notacja O (ang. *big-O notation*). Notacja O wyraża czas wykonania algorytmu, jako funkcję podanego rozmiaru danych wejściowych. Złożoność algorytmu wynosi O(n), gdy czas wykonania rośnie liniowo wraz ze wzrostem danych wejściowych n. Gdy czas wykonania algorytmu jest niezależny od ilości danych to złożoność wynosi O(1), czas wykonania jest stały. W teorii złożoności algorytmów mówimy także o czasie najgorszego przypadku (ang. *worst case time*) oraz o średnim czasie (ang. *average time*).

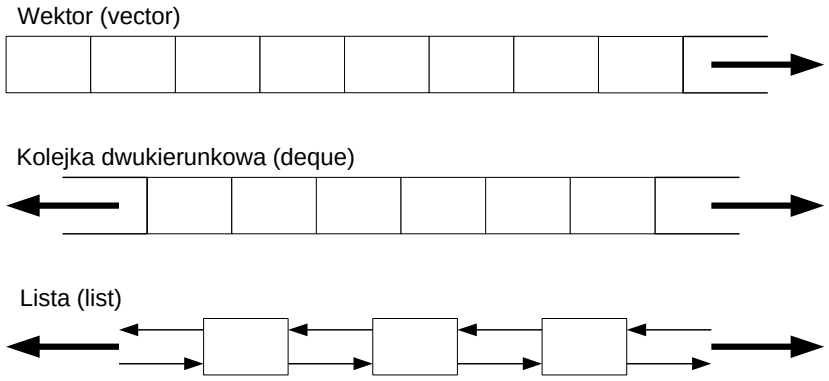
W opisie składników STL do opisu wydajności stosowane jest szeroko pojęcie czasu zamortyzowanego (ang. *amortized time*). Czas zamortyzowany jest użyteczny do charakteryzowania kontenerów, ponieważ czas potrzebny do wykonania konkretnej operacji może zmieniać się w szerokich granicach. STL gwarantuje, że wstawianie i usuwanie elementów na końcu wektora potrzebuje zamortyzowanego stałego czasu, podczas gdy wstawianie i usuwanie elementów w środku potrzebuje czasu liniowego. Powinniśmy pamiętać, że wektor jest zdolny do automatycznego powiększania swojego rozmiaru. To rozszerzenie jest wykonywane, gdy wydane jest polecenie wstawiania, a nie ma już miejsca na pomieszczenie nowego elementu. W takim przypadku STL alokuje miejsce dla 2n elementów (gdzie n jest aktualnym rozmiarem kontenera) i kopiuje n istniejących elementów do nowej pamięci. Alokacja i kopiowanie potrzebują liniowego czasu wykonania. Wstawianie następnych elementów nie potrzebuje rozszerzania pamięci aż do momentu, gdy przydzielona pamięć nie będzie całkowicie zajęta. Widać z tego przykładu, że czas zamortyzowany lepiej odzwierciedla czas wykonania niż czas najgorszego przypadku, który wynosi O(n). Widzimy, że czas zamortyzowany odnosi się do operacji wykonywanych w dłuższym okresie czasu. Pojedyncze operacje mogą trwać znacznie dłużej. W tabeli 2.7 Pokazano czasy wykonania dla kontenerów sekwencyjnych

Tabela 2.7. Czasy wykonania operacji umieszczania i usuwania elementów w kontenerach sekwencyjnych

Lp.	Kontener	początek	środek	koniec
1	vector	Liniowy	Liniowy	Zamortyzowany stały
2	list	Stały	Stały	Stały
3	deque	Zamortyzowany stały	Liniowy	Zamortyzowany stały

Schematycznie typy kontenerów sekwencyjnych przedstawione są na rysunku 2.1.

Programista ustala, jaki typ kontenera należy wybrać do konkretnego zadania.



Rys. 2.1. Typy kontenerów sekwencyjnych STL

Tabela 2.7 z czasami wykonywania podstawowych operacji jest przydatna w wyborze typu kontenera. Wstawianie na końcu kontenera klasy **vector** jest szybkie, **vector** może szybko zwiększać swój rozmiar, jeżeli musimy dodawać nowe elementy. Wstawianie elementów w środku lub ich usuwanie jest znacznie wolniejsze.

Jeżeli projektujemy zadanie, w którym będziemy często wstawiali i usuwali elementy na obu końcach kontenera, powinniśmy raczej zastosować klasę **deque**. Oczywiście do tego zadania możemy użyć także klasy **vector,** ale **deque** wykona to zadanie szybciej. Jeżeli w projektowanym zadaniu wymagać będziemy częstego wstawiania elementów w środku oraz na końcach kontenera, najlepiej jest wykorzystać kontener klasy **list.**

2.3. Kontenery sekwencyjne - vector

Wektor (**vector**) jest jednym z najczęściej używanych kontenerów sekwencyjnych. W zasadzie zastępuje tak często wykorzystywane w C++ tablice. Wektor zarządza swoimi elementami dynamicznie. Pozwala wykorzystać przeciążony operator indeksu, [ ] aby mieć bezpośredni dostęp do swojego dowolnego elementu. Podobnie jak w tablicach języka C, nie jest sprawdzany automatycznie zasięg (nie sygnalizuje automatycznie sięgnięcie po indeks spoza tablicy). Klasa **vector** za to posiada metodę **at()**, która pozwala na kontrolę zakresu indeksów.

Każdy obiekt przechowywany w kontenerze **vector** musi mieć zdefiniowane operator < oraz operator ==.

Specyfikacja szablonu dla klasy **vector** ma postać:

```
template <class T, class Allocator = allocator <T>>  class vector
```

W tej specyfikacji **T** oznacza typ, który będzie przechowywany w kontenerze, **Allocator** określa alokator, domyślnym alokatorem jest alokator standardowy. Klasa **vector** ma następujące konstruktory (Borland Builder v.6.0):

```
explicit vector(const Allocator& alloc = Allocator());
```

Konstruktor tworzy wektor długości zerowej (pusty wektor). Alokator **alloc** służy do zarządzania pamięcią.

```
explicit vector(size_type n);
```

Konstruktor tworzy wektor długości **n**, zawiera **n** kopii domyślnych wartości dla typu **T**. Wymagane jest, aby typ **T** miał domyślny konstruktor. Wektor wykorzysta alokator **Allocator()** do zarządzania pamięcią.

```
vector(size_type n, const T& value, const Allocator& alloc = Allocator());
```

Konstruktor tworzy wektor długości **n**, zawiera **n** kopii **value**. Wektor użyje alokatora **alloc** do zarządzania pamięcią.

```
vector(const vector<T, Allocator>& x);
```

Konstruktor tworzy kopię x.

```
template <class InputIterator>
vector(InputIterator first, InputIterator last, const Allocator& alloc =
Allocator());
```

Konstruktor tworzy wektor długości **last – first**, wektor zostanie wypełniony wszystkimi wartościami otrzymanymi z dereferencji iteratora **InputIterators** w przedziale [**first, last**]. Wektor użyje alokatora **alloc** do zarządzania pamięcią. Klasa vector posiada jeden destruktor:

```
~vector();
```

Destruktor uwalnia całą pamięć zajmowaną przez **vector.** Poniżej pokazane są przykłady tworzenia wektorów:

```
vector <int> v1;                 //tworzy zerowy wektor dla wartości typu int
vector <double> v2(10);      //tworzy 10-elementowy wektor dla wartości
double
vector <char> v3(10, 'c');   //tworzy i inicjuje 10-elementowy wektor do
                              //przechowywania znaków
```

vector <int> v4(v1); //tworzy wektor **int** z innego wektora **int**

W kolejnym programie ilustrujemy podstawowe operacje wykonywane na elementach kontenera **vector**.

Wydruk 2.1. Przykład użycia kontenera **vector,** odwracanie

```
#include <vector>
#include <algorithm>       // reverse()
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
 vector<int> v(4);          // deklaracja wektora v, zawiera 4 elementy
 v[0] = 1;
 v[1] = v[0] + 4;
 v[2] = 2 * v[1];
 v[3] = v[1] + v[2];
 cout << "\n obsluga wektora STL " << endl;
 for (int i = 0; i < v.size(); ++i)
     cout << "v[" << i << "] = " << v[i]<< endl;
 reverse(v.begin(), v.end());              //algorytm odwracania wektora
 cout << " odwraca wektor v " << endl;
 for (int i = 0; i < 4; ++i)
     cout << "v[" << i << "] = " << v[i]<< endl;
 cout << "\n obsluga zwyklej tablicy " << endl;
    double A[5] = { 1.1, 1.2, 1.3, 1.4, 1.5 };
 for (int i = 0; i < 5; ++i)
     cout << "A[" << i << "] = " << A[i]<< endl;
 cout << " odwraca tablice " << endl;
 reverse(A, A + 5);
 for (int i = 0; i < 5; ++i)
     cout << "A[" << i << "] = " << A[i]<< endl;

 getche();
 return 0;
}
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
 obsluga wektora STL
v[0] = 1
v[1] = 5
```

```
v[2] = 10
v[3] = 15
 odwraca wektor v
v[0] = 15
v[1] = 10
v[2] = 5
v[3] = 1
 obsluga zwyklej tablicy
A[0] = 1.1
A[1] = 1.2
A[2] = 1.3
A[3] = 1.4
A[4] = 1.5
 odwraca tablice
A[0] = 1.5
A[1] = 1.4
A[2] = 1.3
A[3] = 1.2
A[4] = 1.1
```

Przeanalizujemy dokładnie przykładowy program. W **main()** tworzony jest wektor **v** potrzebny do przechowywania liczb całkowitych (typu **int**):

```
vector < int > v( 4 ) ;
```

Pojemność początkowa wektora **v** jest równa 4, początkowo wektor zawiera cztery elementy. W kolejnych przypisaniach cztery elementy są inicjalizowane liczbami, a zawartość wektora jest wyświetlana:

```
v[0] = 1;
  v[1] = v[0] + 4;
  v[2] = 2 * v[1];
  v[3] = v[1] + v[2];
  cout << "\n obsluga wektora STL " << endl;
  for (int i = 0; i < v.size(); ++i)
       cout << "v[" << i << "] = " << v[i]<< endl;
```

Zastosowano klasyczną obsługę tablicową, wykorzystując indeksy. Funkcja **size(),** dostępna w każdym kontenerze, zwraca liczbę elementów aktualnie przechowywanych w kontenerze. STL posiada bardzo dużą kolekcję algorytmów (algorytmy są składnikami biblioteki), dzięki którym można manipulować danymi przechowywanymi w kontenerach. W pokazanym programie możemy na przykład odwrócić porządek elementów przechowywanych w kontenerze **v** stosując algorytm **reverse().** Algorytm

reverse() wymaga włączenia pliku nagłówkowego <algorithm> .
Przypominamy, że funkcja **begin()** zwraca iterator, który odwołuje się do pierwszego elementy wektora **v**, a funkcja **end()** zwraca iterator, który odwołuje się do następnej pozycji po ostatnim elemencie w wektorze.

```
reverse(v.begin(), v.end());          //algorytm odwracania wektora
cout << " odwraca wektor v " << endl;
for (int i = 0; i < 4; ++i)
    cout << "v[" << i << "] = " << v[i]<< endl;
```

Należy wiedzieć, że **reverse()** jest funkcją globalną a nie metodą klasy. Pobiera dwa argumenty, operuje na zakresie elementów, w pokazanym przykładzie odwraca wszystkie elementy kontenera **v.** Funkcja **reverse()** jest oddzielona od kontenerów STL, co oznacza, że może także odwracać elementy w zwykłej tablicy, co ilustruje kolejny fragment programu:

```
cout << "\n obsluga zwyklej tablicy " << endl;
   double A[5] = { 1.1, 1.2, 1.3, 1.4, 1.5 };
for (int i = 0; i < 5; ++i)
    cout << "A[" << i << "] = " << A[i]<< endl;
cout << " odwraca tablice " << endl;
reverse(A, A + 5);
for (int i = 0; i < 5; ++i)
    cout << "A[" << i << "] = " << A[i]<< endl;
```

Kontener **vector** posiada dużą liczbę funkcji składowych. Jak już wspomniano, do najczęściej stosowanych należą **size(), begin(), end(), push_back(), insert()** i **erase().** Funkcja **push_back()** umieszcza wartość na końcu wektora. W miarę potrzemy można elementy dodawać także w środku wektora, w tym celu korzystamy z metody **insert().** Elementy mogą być usuwane przy pomocy funkcji **erase().** Funkcja **size()** podaje aktualną liczbę elementów w kontenerze, **capacity()** zwraca liczbę elementów, jakie mogą być przechowywane w kontenerze **vector** zanim **vector** sam zmieni dynamicznie swój rozmiar, aby przyjmować następne elementy. Funkcja **max_size()** podaje maksymalny rozmiar kontenera (jest on zależny od konkretnej implementacji i architektury komputera). Funkcja **empty()** zwraca **true, gdy** kontener jest pusty.

W kolejnym przykładzie demonstrujemy działanie metod informujących o liczbie elementów, jakie przechowuje kontener i jego pojemności.

Wydruk 2.2. Przykład użycia kontenera **vector, size(), capacity()**

```
#include <vector>
#include <iostream>
#include <conio.h>
```

```
using namespace std;

int main()
{ vector<int> v;
  v.push_back( 2 );
  v.push_back( 3 );
  v.push_back( 4 );
  cout <<"   wektor pierwszy :" << endl;
  for (int i = 0; i < v.size(); ++i)
      cout << "v[" << i << "] = " << v[i]<< "   " ;
  cout <<"\nrozmiar (size)  v           : " << v.size()<<endl;
  cout <<"pojemnosc (capacity) v       : " << v.capacity()<<endl;
  cout <<"maksymalny rozmiar (max_size) v : " << v.max_size()<<endl;
  cout <<"pusty (empty) v              : " <<(v.empty()?"prawda":"falsz");
  cout << endl;
  cout <<"\n   nowy wektor :" << endl;
  v.push_back( 5 );
  v.push_back( 6 );
  for (int i = 0; i < v.size(); ++i)
      cout << "v[" << i << "] = " << v[i]<< "   " ;
  cout <<"\nrozmiar (size)  v           : " << v.size()<<endl;
  cout <<"pojemnosc (capacity) v       : " << v.capacity()<<endl;
  getche();
  return 0;
   }
```

W wyniku wykonania programu mamy następujący komunikat:

```
                wektor pierwszy :
v[0] = 2      v[1] = 3      v[2] = 4
rozmiar (size) v                        : 3
pojemność (capacity) v                  : 4
maksymalny rozmiar (max_size) v         : 1073741823
pusty (empty) v                         : falsz

                nowy wektor :
v[0] = 2      v[1] = 3      v[2] = 4      v[3] = 5      v[4] = 6
rozmiar (size) v                        : 5
pojemność (capacity) v                  : 8
```

W **main()** tworzony jest wektor **v** do przechowywania liczb całkowitych. Metoda **push_back()** dodaje nowe elementy umieszczane na końcu:

```
vector<int> v;
```

```
v.push_back( 2 );
v.push_back( 3 );
v.push_back( 4 );
```

Następnie przy pomocy metod **size(), capacity(), max_size()** oraz **empty()** sprawdzamy stan kontenera:

```
cout <<"\nrozmiar (size)  v              : " << v.size()<<endl;
cout <<"pojemnosc (capacity) v        : " << v.capacity()<<endl;
cout <<"maksymalny rozmiar (max_size) v : " << v.max_size()<<endl;
cout <<"pusty (empty) v               : " <<(v.empty()?"prawda":"falsz");
```

Funkcja **size()** zwraca wartość 3 – liczbę elementów umieszczonych w kontenerze **vector**. Funkcja **capacity()** zwraca wartość 4, co oznacza, że pierwotnie nasz pojemnik **v** może przechowywać 4 liczby typu **int.**
W tej sytuacji możemy dodać jeszcze jeden element bez powiększania naszego kontenera. Funkcja **max_size()** zwraca największą liczbę elementów, jaką może zawierać kontener. W podanym przykładzie liczba ta wynosi 1073741823, na innym komputerze może być inna. Funkcja **empty()** informuje czy kontener nie zawiera żadnych elementów, w naszym przykładzie mamy komunikat ”falsz”, co oznacza, że kontener przechowuje elementy. Gdy dodaliśmy dwa nowe elementy do kontenera **v**:

```
v.push_back( 5 );
v.push_back( 6 );
```

to rozmiar **v** będzie 5, a pojemność wyniesie 8. Pojemność za każdym razem podwaja całkowitą przestrzeń przydzieloną na wektor, gdy pojemność początkowa zostaje przekroczona. Elementy wektora mogą być bezpośrednio utworzone z elementów klasycznej tablicy w stylu języka C. W pokazanym poniżej programie mamy instrukcje:

```
int arr[ ] = { 13, 21, 27, 44 } ;
vector < int > v( arr, arr+4 ) ;
```

W pierwszym wierszu mamy deklarację tablicy **arr[ ]** z inicjalizacją w stylu języka C. W drugim wierszy wykorzystano przeciążony konstruktor klasy **vector** do utworzenia i zainicjalizowania wektora v elementami tablicy **arr[ ].** Ta postać konstruktora stosuje dwa iteratory, jako argumenty. Zgodnie z konwencją stosowaną w STL, wskaźniki do tablicy mogą być użyte, jako iteratory. Pokazane wyrażenie tworzy wektor **v** i inicjalizuje go zawartością tablicy liczb całkowitych **arr** od lokalizacji **arr** aż do lokalizacji **arr + 4**.
W kolejnych instrukcjach:

```
while ( ! v.empty() )              // petla dopoki v nie bedzie pusty
  { cout << v.back() <<  " " ;   // pokazuje ostatni element wektora v
    v.pop_back();                  // usuwa ostatni element wektora v
   }
```

zademonstrowano działanie dwóch metod: **back()** i **pop_back().** Funkcja **back()** zwraca ostatni element i dostępna jest w kontenerach wektor, kolejkach **deque** oraz w listach. Metoda **pop_back()** usuwa ostatni element kontenera (nie zwraca go). W naszym przykładzie wykorzystano pętle **while** do wyświetlania elementów kolekcji **v**.
W pętli wyświetlany jest ostatni element wektora **v**, po czym jest kasowany, kontener zmniejsza się o jeden element. W kolejnym wykonaniu pętli sytuacja się powtarza aż do opróżnienia wektora.

Na koniec w instrukcji:

```
cout <<"\n pusty (empty) v : " <<(v.empty() ? "prawda" : "falsz");
```

sprowadzamy, czy rzeczywiście kontener **v** jest pusty. Prawdę mówiąc jest to dość egzotyczny sposób wyświetlania zawartości kontenera.

Wydruk 2.3. Kontener **vector, empty(), back(), pop_back()**

```
#include <vector>
#include <iostream>
#include <conio.h>
using namespace std;
int main()
{ int arr[] = { 13, 21, 27, 44 };        // tablica w stylu C
  vector<int> v(arr, arr+4);     // inicjalizacja wektora v elementami tablicy
  cout << " elementy wektora v :" << endl;
  for (int i=0; i<v.size(); i++)
       cout << v[i] << " ";
 cout << "\n wydruk elementow wektora v od konca i usuwanie : "<< endl;
 while ( ! v.empty() )                  // petla dopoki v nie bedzie pusty
 { cout << v.back() <<  " " ;          // pokazuje ostatni element wektora v
   v.pop_back();                       // usuwa ostatni element wektora v
 }
 cout <<"\n pusty (empty) v : " <<(v.empty() ? "prawda" : "falsz");
  getche();
  return 0;
}
```

Wynikiem uruchomienia programu jest następujący wydruk:

```
                elementy wektora v :
```

```
13  21  27  44
  wydruk elementow wektora v od konca i usuwanie :
44 27  21  13
   pusty (empty) v : prawda
```

W kolejnym przykładzie ilustrujemy tworzenie wektora **v** przechowującego liczby rzeczywiste (typu **double**), który dynamicznie powiększa swój rozmiar (nie wiemy z góry ile liczb wprowadzimy) a następnie wyliczamy medianę. W celu wyliczenia mediany, liczby kontenera przechowywane w kontenerze muszą być posortowane. W tym celu wykorzystano algorytm STL **sort().**

Wydruk 2.4. Kontener vector, mediana**, push_ back(), size(),**

```
#include <vector>
#include <algorithm>                    //sort()
#include <iostream>
#include <conio.h>

using namespace std;

int main()
{
 vector<double> v;
 double x;
 cout <<"wprowadz dane, 0.0 konczy : "<< endl;
  cin >> x;
  while (x > 0.0)
    { v.push_back(x);
      cin >> x;
    }
  cout << "wprowadzone dane :  " ;
  for (int i=0; i<v.size(); i++)  cout << v[i] << " ";
  sort ( v.begin(), v.end() );                              //sortowanie
  cout << "\nposortowane dane :  " ;
   for (int i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\nmediana =  " ;                                  //mediana
  if (v.size() % 2 == 0)                  // parzysta liczba danych
     cout << (v[v.size()/2 - 1] + v[(v.size()/2)])/2 << endl;
  else
     cout << v[v.size()/2] << endl;

getche();
 return 0;
}
```

Wydruk z programu ma postać (dla parzystej liczby danych):

```
wprowadz dane, 0.0 konczy :
4.4  3.3  6.6  5.5  0.0
wprowadzone dane :  4.4  3.3  6.6  5.5
posortowane dane  :   3.3  4.4  5.5  6.6
mediana = 4.95
```

Wydruk dla nieparzystej liczby danych ma postać:

```
wprowadz dane, 0.0 konczy :
4.4  3.3  6.6  5.5  1.1  0.0
wprowadzone dane :  4.4  3.3  6.6  5.5  1.1
posortowane dane  :   1.1  3.3  4.4  5.5  6.6
mediana = 4.4
```

Kolejny program ilustruje obsługę złożonych typów danych. Elementem kontenera tym przykładzie jest struktura, zbudowana z dwóch pól: typu **string** i typu **double**.

Wydruk 2.5. Kontener **vector,** elementy wektora w postaci struktury**,**

```
#include <vector>
#include <string>
#include <iostream>
#include <conio.h>
using namespace std;
struct Dane {                              //elementy kontenera wektor
  string nazwa;
  double pow;
};
bool Wprowadz_Dane(Dane & dd);
void Drukuj(const Dane & dd);
int main()
{ vector<Dane> kraj;
  Dane  tmp;
  while( Wprowadz_Dane(tmp) )
        kraj.push_back(tmp);
  cout << "\n Kraj       powierzchnia" << endl;
  for ( int i =0; i < kraj.size(); i++)
       Drukuj(kraj[i]);
  getche();
  return 0;
}
bool Wprowadz_Dane ( Dane & dd )
{ cout <<"wpisz kraj (koniec - aby zakonczyc): ";
  cin >>  dd.nazwa;
```

```
  if (dd.nazwa == "koniec")
          return false;
  cout << "podaj powierzchnie : ";
  cin >> dd.pow;
          return true;
}
void Drukuj ( const Dane & dd )
   { cout << dd.nazwa << "\t\t" << dd.pow << endl; }
```

Wydruk z programu ma postać:

```
wpisz kraj (koniec – aby zakonczyc): Polska
podaj powierzchnie : 321685
wpisz kraj (koniec – aby zakonczyc): Ukraina
podaj powierzchnie : 603700
wpisz kraj (koniec – aby zakonczyc): Niemcy
podaj powierzchnie : 356974
wpisz kraj (koniec – aby zakonczyc): koniec


  Kraj              powierzchnia
Polska                 321685
Ukraina                603700
Niemcy                 356974
```

W pokazanym programie struktura danych ma postać:

```
struct Dane
{    string nazwa;
     double pow;
};
```

Nasza aplikacja posiada dwie funkcje globalne:

```
bool Wprowadz_Dane(Dane & dd);
void Drukuj(const Dane & dd);
```

Pierwsza obsługuje wprowadzanie danych (napisu oraz liczby), druga powoduje wyświetlenie danych struktury na ekranie. Struktura tworzy element umieszczany w wektorze **kraj**:

```
vector<Dane> kraj;
Dane  tmp;
```

```
while( Wprowadz_Dane(tmp) )
  kraj.push_back(tmp);
```

Dopóki funkcja **Wprowadz_Dane()** nie zwróci **false**, metoda **push_back()** będzie umieszczała dane (nazwę kraju i obszar kraju) w kontenerze wektor. Każde wykonanie pętli powoduje dodanie nowego elementu do wektora **kraj.** Nie musimy podawać liczby elementów. Dopóki program posiada dostęp do odpowiedniej ilości pamięci, będzie odpowiednio do potrzeb zwiększał wielkość kontenera.

2.4. Kontenery sekwencyjne - deque

Szablon klasy **deque** (wymawia się, jako „diik" albo, jako „dek") znajduje się w pliku <deque> i reprezentuje kolejkę o dwóch końcach. Klasa **deque** jest bardzo pożytecznym zasobnikiem gdyż łączy w sobie cechy kontenerów **vector** i **list**. Termin „deque" jest skrótem „kolejki dwukierunkowej" (ang. *double–ended queue*). W swoim działaniu **deque** jest bardzo podobnym kontenerem do wektora. Zasadnicza różnica polega na tym, że wstawianie i usuwanie elementów z początku kontenera **deque** jest operacją o stałej złożoności, podczas gdy dla wektora są to operacje o złożoności liniowej.

Tak jak w przypadku wektora dostęp do wybranych elementów w **deque** jest szybki (stały czas). Wstawianie i usuwanie elementów w środku kontenera **deque** jest podobne do wykonywania tych operacji w kontenerze wektor (czas liniowy) a nawet trochę wolniejsze, ponieważ kontener **deque** ma bardziej złożoną strukturę niż wektor. Kontenera **deque** powinno się używać zamiast kontenera wektor, gdy planowane jest częste wstawianie i usuwanie elementów na początku i końcu kontenera oraz potrzebny jest odczyt wszystkich elementów. Klasa deque stosuje obsługę iteratorów o dostępie bezpośrednim, wszystkie algorytmy STL mogą być stosowane do działania na obiektach **deque.** Najczęściej kontener **deque** jest stosowany przy realizacji kolejki elementów FIFO („pierwszy wszedł, pierwszy wyszedł', ang. *first-in-first-out*). Kolejka **deque** nie udostępnia funkcji służących do zarządzania pojemnością takich jak **capacity()** i **reserve().** Wymienione funkcje są dostępne w klasie **vector**. W przeciwieństwie do kontenera **vector**, kolejka **deque** udostępnia funkcję **push_front()** oraz **pop_front()** do wstawiania i usuwania pierwszego elementu. Pozostałe operacje są takie same jak dla klasy **vector**. Należy mieć na uwadze, że wstawienie lub usunięcie elementów może spowodować realokację, co prowadzi zazwyczaj do unieważnienia wszystkich wskaźników, referencji i iteratorów w kolejce **deque**. Wyjątkiem jest operacja wstawiania elementów na początku lub na końcu kolejki **deque.**
Specyfikacja szablonu dla klasy **deque** ma postać:

```
template <class T, class Allocator = allocator<T> >  class deque;
```

W tej specyfikacji **T** oznacza typ, który będzie przechowywany w kontenerze,

Allocator określa alokator, domyślnym alokatorem jest alokator standardowy. Klasa **deque** ma następujące konstruktory (Borland Builder v.6.0):

```
explicit deque(const Allocator& alloc = Allocator());
```

Konstruktor tworzy kolejkę **deque** długości zerowej (pusta kolejka). Alokator **alloc** służy do zarządzania pamięcią.

```
explicit deque(size_type n);
```

Konstruktor tworzy kolejkę długości **n**, zawiera **n** kopi domyślnych wartości dla typu **T**. Wymagane jest, aby typ **T** miał domyślny konstruktor. Kolejka **deque** wykorzysta alokator **Allocator()** do zarządzania pamięcią.

```
deque(size_type n, const T& value, const Allocator& alloc =Allocator());
```

Konstruktor tworzy kolejkę długości **n**, zawiera **n** kopii **value**. Kolejka **deque** użyje alokatora **alloc** do zarządzania pamięcią.

```
deque(const deque<T, Allocator>& x);
```

Konstruktor tworzy kopię **x**.

```
template <class InputIterator>
deque(InputIterator first, InputIterator last, const Allocator& alloc =
      Allocator());
```

Konstruktor tworzy kolejkę **deque** długości **last – first**, Kolejka zostanie wypełniona wszystkimi wartościami otrzymanymi z dereferencji iteratora **InputIterators** w przedziale [**first, last**]. Kolejka **deque** użyje alokatora **alloc** do zarządzania pamięcią.
Destruktor ma postać:

```
~deque();
```

Zwalnia całą pamięć przydzieloną kolejce **deque.**

W kolejnym przykładzie przedstawiamy podstawowe cechy klasy **deque**. Klasa **deque** wymaga włączenia pliku nagłówkowego:

```
#include <deque>
```

W instrukcjach:

```
deque < int > w ;
 w.push_front ( 5 );
 w.push_front ( 99 );
 w.push_back ( 777 );
```

najpierw tworzymy obiekt **deque** o nazwie **w**, który może przechowywać wartości typu **int**. Wykorzystane są funkcje **push_front()** oraz **push_back()** do wstawiania elementów na początku i końcu kontenera **deque**. Funkcja **push_front** jest dostępna tylko dla kontenerów **list** i **deque**.

Wydruk 2.6. Kontener **deque**, podstawowe operacje

```
#include <deque>
#include <iostream>
#include <conio.h>
using namespace std;
int main()
{ deque < int > w ;
  w.push_front ( 5 );
  w.push_front ( 99 );
  w.push_back ( 777 );
  cout << "deque w zawiera liczby :"<< endl;
  for (int i =0; i < w.size(); i++)
      cout << w[i] << "  ";
  w[1] = 555;
  cout << "\npo zmianie deque w zawiera liczby :"<< endl;
  for (int i =0; i < w.size(); i++)
      cout << w[i] << "  ";
  getche();
  return 0;
}
```

Po uruchomieniu programu mamy następujący wynik:

```
deque w zawiera liczby:
99   5   777
po zmianie deque w zawiera liczby :
99   555   777
```

Przy pomocy operatora indeksu odzyskujemy wartości w każdym elemencie, dzięki czemu możemy wyświetlić liczby przechowywane w kolejce **w**:

```
cout << "deque w zawiera liczby :"<< endl;
  for (int i =0; i < w.size(); i++)
       cout << w[i] << "  ";
```

W kolejnej instrukcji:

```
w[1] = 555;
```

wykorzystujemy operator indeksu do utworzenia *lwartości*. Dzięki temu możemy przyporządkować wartość bezpośrednio do dowolnego elementu obiektu kolejki **deque**.

Dla kontenerów **deque** mamy dostępne metody **begin(), end(), rbegin()** oraz **rend(),** które zwracają iteratory o takim samym znaczeniu jak w innych kolekcjach sekwencyjnych.
Dostępne są także metody związane z rozmiarem: **size(), max size(), resize()** i **empty()** o takim samym znaczeniu jak w innych kolekcjach sekwencyjnych.

W kolejnym przykładzie ilustrujemy obsługę kolejki **deque** i omawiamy inne podstawowe operacje. W programie utworzona jest tablica **tab[ ]** zainicjalizowana wartościami typu **int.** Elementy tablicy **tab[ ]** są umieszczone w kontenerze **deque**. Funkcje **insert()** i **pop_front()** służą do umieszczania elementów w kolejce **w** oraz do usuwania elementów z kolejki.

Wydruk 2.7. Kontener **deque**, podstawowe operacje, **insert(), sort()**

```
#include <deque>
#include <iterator>                    //ostream_iterator
#include <algorithm>                   //copy(), sort()
#include <iostream>
#include <conio.h>

using namespace std;
const int MAX = 10;

int main()
{
 int tab[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
 deque<int> w(tab, tab+5);
 cout << "elementy tablicy tab skopiowane do kolejki w :" << endl;
 ostream_iterator <int> output (cout, "   ");
 copy (w.begin(), w.end(), output);                    // 1 2 3 4 5
 cout << endl << "trzy elementy tab wstawione do w :" << endl;
 w.insert(w.begin()+3, tab+7, tab+10);
 copy (w.begin(), w.end(), output);                    // 1 2 3 8 9 10 4 5
 cout << endl << "pierwszy element usuniety z kolejki w:" << endl;
 w.pop_front();
 copy (w.begin(), w.end(), output);                    // 2 3 8 9 10 4 5
```

```
  cout << endl << "dodany nowy element na poczatek kolejki :" << endl;
  w.push_front(9);
  copy (w.begin(), w.end(), output);                    // 9 2 3 8 9 10 4 5
  cout << endl << "posortowane elementy kolejki w : " <<  endl;
  sort ( w.begin(), w.end() );                          // sortowanie
  copy (w.begin(), w.end(), output);

   getche();
   return 0;
}
```

Wydruk wyników działania programu ma następującą postać:

```
elementy tablicy tab skopiowane do kolejki w :
1   2   3   4   5
trzy elementy tab wstawione do w :
1   2   3   8   9   10   4   5
pierwszy element usuniety z kolejki w :
2   3   8   9   10   4   5
dodany nowy element na początek kolejki :
9   2   3   8   9   10   4   5
posortowane elementy kolejki w :
2   3   4   5   8   9    9    10
```

W programie utworzona została klasyczna tablica **tab[ ]** oraz utworzona została kolejka **deque** o nazwie **w**, do której skopiowano elementy tablicy:

```
int tab[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
deque < int > w ( tab, tab+5);
```

W celu wydrukowania elementów kolejki **w** zastosowano instrukcję:

```
ostream_iterator <int> output (cout, "   ");
```

która tworzy **ostream_iterator** o nazwie **output,** który może być użyty przez **cout** do wysłania na wyjście liczb całkowitych oddzielonych spacjami. Stosowanie iteratora wyjściowego (wymagany jest plik nagłówkowy <iterator>) jest bezpiecznym mechanizmem. Pierwszy argument dla konstruktora określa strumień wyjściowy, a drugi jest napisem określającym znaki rozdzielające dla wartości wyjściowych. Kolejna instrukcja:

```
copy (w.begin(), w.end(), output);
```

wykorzystuje algorytm **copy()** (wymagany jest plik nagłówkowy <algorithm>) do wyprowadzenia całej zawartości obiektu **w** klasy **deque** do standardowego wyjścia.

Algorytm **copy()** kopiuje wartości z zakresu wskazanego przez [**w.begin(), w.end()**). Elementy kopiowane są do miejsca określonego przez iterator wyjściowy (trzeci argument). W pokazanym przykładzie iteratorem wyjściowym jest **ostream_iterator output**, który jest dowiązany do **cout,** więc elementy są skopiowane do standardowego wyjścia.
W instrukcji:

```
w.insert(w.begin()+3, tab+7, tab+10);
```

wykorzystana została metoda **insert().**
Funkcja **insert()** w ogólnej postaci:

```
w.insert(pos, beg, end)
```

wstawia na pozycji iteratora **pos** kopie wszystkich elementów z zakresu [**beg, end**). W naszym przypadku wstawiane są elementy tablicy **tab[ ]** do kolejki **w** na pozycji czwartej. Wstawiane są trzy elementy tablicy: ósmy, dziewiąty i dziesiąty. Elementy kolejki: czwarty i piąty są przesuwane na koniec kolejki.
Kolejne metoda:

```
w.pop_front();
```

usuwa pierwszy element kolejki, a metoda:

```
w.push_front(9);
```

wstawia liczbę 9 na początek kolejki. Na koniec algorytm **sort()** sortuje elementy kolejki **w**:

```
sort ( w.begin(), w.end() );
```

2.5. Kontenery sekwencyjne - list

Zasobnik sekwencyjny **list** jest standardową listą powiązaną, wymaga pliku nagłówkowego <list>. Podobnie jak kontenery **vector** czy **deque** służy do zapisywania ciągu elementów. Istotną cechą kontenera **list** jest to, że nie musi zajmować w pamięci ciągłego obszaru. Podobnie jak w klasycznej liście, każdy element kontenera **list** wskazuje gdzie są elementy następny i poprzedni (wykorzystujemy do tego celu wskaźniki, element pierwszy i ostatni są traktowane inaczej). Taka lista nosi nazwę listy podwójnie powiązanej. Ten fakt umożliwia klasie list obsługę dwukierunkowych iteratorów, dzięki którym można przeglądać kontener do przodu jak i w odwróconym porządku. Wiele algorytmów STL może operować na obiekcie typu **list**. W porównaniu do

kontenera **vector** czy **deque**, dostęp do elementów jest wolny (liniowy). Zaletą kontenera **list** jest szybkość wstawiania i usuwania elementów w dowolnym miejscu (czas stały). Listy stosujemy w przypadku gdy często wstawiamy i usuwamy elementy a rzadko przeglądamy listę. Taki przypadek zachodzi gdy obsługujemy biuro informacyjne dla klientów. Klienci często kontaktują się z operatorem a potem się odłączają, rzadko operator musi przeglądać listę rozmówców.

W przeciwieństwie do kontenera **vector,** kontener **list** nie obsługuje indeksowania ani nie umożliwia dostępu swobodnego. Iteratory wektora działają inaczej niż iteratory listy.

W wektorze po wstawieniu nowego elementu, iterator dalej wskazuje to samo miejsce, co oznacza, że gdy wstawimy nowy element na początku kontenera, iterator pokaże inną wartość (wszystkie elementy są przesuwane o jedno miejsce, aby zrobić miejsce dla nowego elementu). Gdy wstawiany jest nowy element do listy, ta operacja nie powoduje przesunięcia pozostałych elementów, może tylko zmienić informacje wiążące elementy. Iterator wskazujący na dany element dalej wskazuje na ten sam element.
Specyfikacja szablonu dla klasy **list** ma postać:

```
#include <list>
template <class T, class Allocator = allocator<T> >  class list;
```

W tej specyfikacji **T** oznacza typ, który będzie przechowywany w kontenerze, **Allocator** określa alokator, domyślnym alokatorem jest alokator standardowy.

Klasa **list** ma następujące konstruktory (Borland Builder v.6.0):

```
explicit list(const Allocator& alloc = Allocator());
```

Konstruktor tworzy listę **list** długości zerowej (pusta lista). Alokator **alloc** służy do zarządzania pamięcią.

```
explicit list(size_type n);
```

Konstruktor tworzy listę długości **n**, zawiera **n** kopi domyślnych wartości dla typu **T**. Wymagane jest aby typ **T** miał domyślny konstruktor. Lista **list** wykorzysta alokator **Allocator()** do zarządzania pamięcią.

```
list(size_type n, const T& value, const Allocator& alloc =
 Allocator());
```

Konstruktor tworzy listę długości **n**, zawiera **n** kopii value. Lista **list** użyje alokatora **alloc** do zarządzania pamięcią.

```
template <class InputIterator>
list(InputIterator first, InputIterator last, const Allocator& alloc =
 Allocator());
```

Konstruktor tworzy listę **list** długości **last – first**, lista zostanie wypełniona wszystkimi wartościami otrzymanymi z dereferencji iteratora **InputIterators** w przedziale [**first, last**]. Lista **list** użyje alokatora **alloc** do zarządzania pamięcią.

```
list(const list<T, Allocator>& x);
```

Konstruktor tworzy kopię **x**.
Destruktor klasy list ma postać:

```
~list();
```

Zwalnia całą pamięć przydzieloną liście.

W porównaniu do klasy **vector**, klasa **list** posiada dodatkowe funkcje, przedstawione w tabeli 2.8.

Tabela 2.8. Niektóre funkcje składowe kontenera **list**.

Lp.	Funkcja	Opis
1	void splice(iterator i, list<T,Alloc> &x)	Wstawia zawartość listy **x** przed pozycją **i**, lista **x** staje się pustą
2	void push_front(const T &val)	Dodaje kopię **val** na początek listy
3	void pop_front()	Usuwa pierwszy element listy
4	void remove(const T &val)	Usuwa z listy wszystkie wystąpienia **val**
5	void unique()	Usuwa z listy wszystkie elementy powtarzające się
6	void merge(list<T, Alloc> &x)	Scala listę **x** z listą wywołującą, wynikiem jest lista posortowana. Lista **x** staje się pusta
7	void reverse()	Odwraca porządek elementów
8	void sort()	Sortuje listę

Do dodawania i usuwania elementów kontener list posiada następujące metody : **push_back(), pop_back(),** trzy formy **insert(),** dwie formy **erase()** oraz

clear(). W kontenerze **list** obsługiwane są następujące metody związane z rozmiarem: **size()** oraz **empty().** Kontener **list** nie obsługuje metody **resize()** oraz **capacity().**

Kolejny program przedstawia podstawowe operacje, jakie mogą być wykonywane na kontenerze **list.** Omówione zostaną proste operacje wstawiania elementów na początku kontenera: **push_front()** oraz na końcu kontenera – **push_back().** Pokazano użycie funkcji składowej klasy **list** o nazwie **sort()** do ustawiania elementów obiektu **list** w porządku rosnącym.

Wydruk 2.8. Kontener **list,** podstawowe operacje

```
#include <list>
#include <algorithm>
#include <iostream>
#include <conio.h>

using namespace std;
const int R = 4;

void drukuj(double &p) {cout << p << " , "; }

int main ()
{
 list<double> w1, w2;
 double t[R] = { 2.7, 5.5, 7.7, 9.9 };
 w1.push_back(0.5);
 w1.push_front(5.5);
 w1.push_back(55.5);
   cout << " lista w1 pierwotna : " << endl;
   for_each(w1.begin(), w1.end(),drukuj);
 w1.sort();
   cout << "\n lista w1 posortowana : " << endl;
   for_each(w1.begin(), w1.end(),drukuj);
 w2.insert(w2.begin(), t, t+R) ;
   cout << "\n lista w2 pierwotna : " << endl;
   for_each(w2.begin(), w2.end(),drukuj);
 w1.splice(w1.end(), w2);                    // lista w2 po splice() jest pusta!
   cout << "\n lista w1 + w2, splice() : " << endl;
   for_each(w1.begin(), w1.end(),drukuj);
 w2.insert(w2.begin(), t, t+R) ;             // lista w2 po splice() jest pusta!
   cout << "\n lista w2, insert() : " << endl;
   for_each(w2.begin(), w2.end(),drukuj);
 w1.sort();
 w2.sort();
```

```
  w1.merge(w2);                          // listy powinny byc posortowane
    cout << "\n lista w1, merge() -> w1 + w2 : " << endl;
    for_each(w1.begin(), w1.end(),drukuj);
  w1.unique();
    cout << "\n lista w1, unique() : " << endl;
    for_each(w1.begin(), w1.end(),drukuj);

    getche();
    return 0;
}
```

Wynikiem działania programu jest następujący wydruk:

```
 lista w1 pierwotna :
5.5 , 0.5 , 55.5 ,
  lista w1 posortowana :
0.5 , 5.5 , 55.5 ,
  lista w2 pierwotna :
2.7 , 5.5 , 7.7 , 9.9 ,
  lista w1 + w2, splice() :
0.5 , 5.5 , 55.5 , 2.7 , 5.5 , 7.7 , 9.9 ,
  lista w2 insert() :
2.7 , 5.5 , 7.7 , 9.9 ,
  lista w1, merge() -> w1 + w2 :
 0.5 , 2.7 , 2.7 , 5.5 , 5.5 , 5.5 , 7.7 , 7.7 , 9.9 , 9.9 , 55.5 ,
 lista w1, unique() :
0.5 , 2.7 , 5.5 , 7.7 , 9.9 , 55.5 ,
```

W instrukcjach:

```
list < double > w1, w2;
double t[R] = { 2.7, 5.5, 7.7, 9.9 };
w1.push_back(0.5);
w1.push_front(5.5);
w1.push_back(55.5);
```

utworzone zostały dwa obiekty **list** do przechowywania liczb rzeczywistych (typu **double**), zadeklarowana i zainicjalizowana klasyczna tablica **t[ ]** oraz umieszczone trzy wartości w kontenerze **w1** (metoda **push_front()** i **push_back()**). W instrukcji :

```
for_each ( w1.begin(), w1.end(), drukuj );
```

wykorzystano algorytm STL o nazwie **for_each().** Wywołuje on funkcję zdefiniowaną przez użytkownika wobec każdego elementu z podanego zakresu.

W naszym przypadku wywołana jest funkcja **drukuj()** wobec każdego elementu listy **w1** z zakresu **w1.begin()** i **w1.end().** Powoduje to wydruk na ekranie monitora wartości przechowywanych w kontenerze **list**. Kolejna instrukcja:

```
w2.insert(w2.begin(), t, t+R) ;
```

powoduje skopiowanie wartości przechowywanych w tablicy **t[ ]** do kontenera **w2**. Korzystamy z wersji funkcji **insert(),** która używa drugiego i trzeciego argumentu do określenie początkowej i końcowej pozycji w sekwencji wartości (w naszym przypadku z tablicy liczb typu **double**), które powinny zostać wstawione do obiektu **list.** W instrukcji:

```
w1.splice(w1.end(), w2);
```

stosujemy metodę **splice().** Jest to rozbudowana funkcja, posiada trzy wersje. W naszym przykładzie elementy z listy **w2** wstawimy do listy **w1**, przed pozycją iteratora określonego, jako pierwszy argument.
Lista **w2** staje się pusta. W kolejnych instrukcjach:

```
w2.insert(w2.begin(), t, t+R) ;
w1.sort();
w2.sort();
w1.merge(w2);
```

ponownie umieszczamy wartości w liście **w2,** sortujemy **w1** i **w2** (w sensie metody klasy **list**) a następnie wykorzystujemy metodę **merge()** do umieszczenia wartości listy **w2** w liście **w1**. Obiekty klasy **list** powinny być uporządkowane przed przeprowadzeniem tej operacji.
Na koniec w instrukcji:

```
w1.unique();
```

wykorzystujemy metodę klasy **list** do usunięcia powtórzeń elementów w obiekcie klasy list **w1**. Lista powinna być uporządkowana przed wykonaniem tej operacji.

Kolejny program ilustruje działanie kontenera **list** przechowującego elementy typu **string**. Tworzymy listę kompozytorów i sprawdzamy czy wprowadzone z klawiatury nazwisko znajduje się na liście, a jeżeli jest na liście, to sprawdzamy na jakim miejscu. Wykorzystano ogólny algorytm **find().**

Wydruk 2.9. Kontener **list**, podstawowe operacje, algorytm **find()**

```
#include <list>
#include <algorithm>
#include <string>
#include <iostream.h>
#include <conio.h>
using namespace std;
int main()
{string n;
 list<string> kompozytor;
  kompozytor.push_back("Wagner");
  kompozytor.push_back("Bach");
  kompozytor.push_back("Vivaldi");
  kompozytor.push_back("Mozart");
  kompozytor.push_back("Beethoven");
 list<string>::iterator p;
 cout << " Lista kompozytorow :" << endl;
 for (p=kompozytor.begin(); p!=kompozytor.end(); ++p)
    cout << *p << endl;
 cout << "   podaj poszukiwane nazwisko : " ;
 cin >> n;
 p = find(kompozytor.begin(), kompozytor.end(), n);
 if (p == kompozytor.end())
      cout<< "nie znaleziono nazwiska : " << n << endl;
 else if (++p != kompozytor.end())
      cout<< n << " znaleziony przed : "<< *p << endl;
 else
      cout<< n <<" znaleziony na koncu." << endl;
  getche();
  return 0;
}
```

Działanie programu jest następujące:

```
Lista kompozytorow :
Wagner
Bach
Vivaldi
Mozart
Beethoven
      Podaj poszukiwane nazwisko : Vivaldi
Vivaldi znaleziony przed : Mozart
```

W programie deklarujemy listę i wstawiamy elementy:

```
list<string> kompozytor;
kompozytor.push_back("Wagner");
kompozytor.push_back("Bach");
kompozytor.push_back("Vivaldi");
kompozytor.push_back("Mozart");
kompozytor.push_back("Beethoven");
```

Jak zwykle metoda **push_back()** pozwala na bezpieczne wstawienie wszystkich żądanych elementów na listę.
W instrukcjach:

```
list < string > :: iterator p ;
cout << " Lista kompozytorow :" << endl;
for (p = kompozytor.begin(); p! = kompozytor.end(); ++p)
    cout << *p << endl;
```

wykorzystano iterator do obsługi kontenera **list.** Pierwsza pokazana instrukcja definiuje iterator **p**. Przy pomocy iteratorów otrzymujemy najprostszy sposób uzyskiwania dostępu do elementów przechowywanych w kontenerach.
Wszystkie kontenery mają funkcje składowe **begin()** i **end(),** które zwracają iteratory wskazujące na początek i koniec kontenera. Iteratory, tak samo jak wskaźniki można inkrementować, dekrementować, stosować operator dereferencji *. W pętli **for** wykorzystując dereferencje iteratora **p** drukujemy na ekranie monitora nazwiska wprowadzonych na listę kompozytorów.
W instrukcjach:

```
cout << "   podaj poszukiwane nazwisko : " ;
 cin >> n;
 p = find(kompozytor.begin(), kompozytor.end(), n);
 if (p == kompozytor.end())
      cout<< "nie znaleziono nazwiska : " << n << endl;
 else if (++p != kompozytor.end())
      cout<< n << " znaleziony przed : "<< *p << endl;
 else
      cout<< n <<" znaleziony na koncu." << endl;
```

realizujemy zadanie wprowadzenia z klawiatury nazwiska poszukiwanego kompozytora a następnie wykorzystując algorytm STL **find()** sprawdzamy, czy wprowadzone nazwisko znajduje się na liście i na jakiej pozycji. Algorytm **find(),** jak każdy algorytm zdefiniowany w STL operuje na szablonach za pomocą iteratorów. Wszystkie algorytmy są szablonami funkcji, **find()** zadeklarowany jest następująco:

```
template < class InIter, class T >
InIter  find ( InIter start, InIter end, const T &val )
```

Algorytm **find()** przeszukuje kontener w zakresie od **start** do **end,** poszukuje wartości określonej przez **val**. Zwraca iterator do pierwszego wystąpienia elementu lub do **end**, gdy w kontenerze nie występuje szukana wartość.

W kolejnym programie wykorzystano listę **list** do obsługi typów napisowych. Jak pamiętamy klasa **list** bazuje na liście dwukierunkowej, mamy możliwość wstawiania innego obiektu klasy **list** do istniejącego już obiektu klasy list. Wykorzystać do tego celu możemy metodę splice().

Wydruk 2.10. Kontener **list,** podstawowe operacje, metoda **splice()**

```
#include <list>
#include <algorithm>
#include <string>
#include <iostream.h>
#include <conio.h>
using namespace std;

int main()
{string n;
 list<string> kraj, nowe;
  kraj.push_back("Albania");
  kraj.push_back("Andora");
  kraj.push_back("Argentyna");
  kraj.push_back("Cejlon");
  kraj.push_back("Chile");
  kraj.push_back("Chiny");
  kraj.push_back("Czechy");
  nowe.push_back("Belgia");    // nowe panstwa
  nowe.push_back("Birma");
  nowe.push_back("Brazylia");

 list<string>::iterator p;
 cout << " Lista panstw :" << endl;
 for (p=kraj.begin(); p!=kraj.end(); ++p)
     cout << *p << endl;
 cout << "   podaj miejsce wstawiania : " ;
 cin >> n;
 p = find(kraj.begin(), kraj.end(), n);
 kraj.splice(p, nowe);
 cout << " Nowa lista panstw :" << endl;
 for (p=kraj.begin(); p!=kraj.end(); ++p)
     cout << *p << endl;

  getche();
  return o;
}
```

Po uruchomieniu wynik działania tego programu wygląda następująco:

```
 Lista panstw :
Albania
Andora
Argentyna
Cejlon
Chile
Chiny
Czechy
      Podaj miejsce wstawiania : Cejlon
  Nowa lista panstw :
Albania
Andora
Argentyna
Belgia
Birma
Brazylia
Cejlon
Chile
Chiny
Czechy
```

Tworzone są dwa obiekty klasy **list:**

```
list<string> kraj, nowe;
```

metoda **push_back()** służy do dodawania elementów listy do kontenerów **kraj** i **nowe**:

```
kraj.push_back("Albania");
nowe.push_back("Belgia");
```

W naszym programie tworzymy listę państw w porządku alfabetycznym. Na liście **kraj** znajdują się państwa nazwach zaczynających się na literę A i C. Na liście **nowe** umieściliśmy państwa o nazwach zaczynających się na literę B. Chcemy utworzyć jedną listę a nazwy wszystkich państwa umieścić w porządku alfabetycznym. Należy ustalić miejsce, w którym wstawimy obiekt **list nowe** do listy **kraj**.

Przy pomocy iteratora **p** i funkcji **find()** znajdujemy miejsce, w którym wstawimy państwa zaczynające się na literę B. Musimy pamiętać, że iterator **p** wskaże miejsce za ostatnim elementem w określonym przedziale, w naszej sytuacji argumentem funkcji **find()** musi być pierwsze państwo o nazwie zaczynającej się na C (Cejlon):

```
string n;
list < string > :: iterator p ;
cout << "   podaj miejsce wstawiania : " ;
cin >> n;
p = find ( kraj.begin(), kraj.end(), n ) ;
```

Dla iteratorów klasy **list** nie można zastosować przeciążonego operatora + w postaci **p + 1**, co jest w pełni zrozumiałe, natomiast wykonywalna jest operacja **p++** czy **p--** .
W instrukcji :

```
kraj.splice(p, nowe);
```

w obiekcie **kraj** klasy **list** będą wstawione elementy z klasy **nowe** w miejscu wskazanym przez iterator **p**. Elementy z obiektu **nowe** będą usunięte.
Bardzo często zachodzi konieczność manipulowania istniejącymi już listami. Przydatnym jest algorytm STL **copy(),** który należy do grupy algorytmów modyfikujących. Tego typu algorytmy modyfikują wszystkie lub niektóre elementy sekwencji. Niektóre algorytmy z tej grupy modyfikują ciąg wejściowy (mogą nawet usunąć wszystkie elementy), inne pozostawiają ciąg wejściowy niezmieniony. Zawsze należy upewnić się jak będzie wyglądała sekwencja wejściowa i jak wyjściowa po operacji **copy().** Kolejny program ilustruje podstawowe działanie **copy()** w odniesieniu do obiektów klasy **list**.

Wydruk 2.11. Kontener **list**, metoda **copy(), insert(), erase()**

```
#include <list>
#include <iostream.h>
#include <conio.h>
using namespace std;
const int R =5;
int main()
{ list<int>::iterator p;
  int a1[R]= { 1, 3, 5, 7, 9 };
  int a2[R]= { 2, 4, 6, 8, 10 };
  list<int> w1(a1, a1+5);      // inicjalizacja w1 liczbami a1
  list<int> w2(a2, a2+5);      // inicjalizacja w2 liczbami a2
    cout << " lista w1 : ";
    for (p=w1.begin(); p!=w1.end(); ++p) cout << *p << "   ";
    cout << "\n lista w2 : ";
    for (p=w2.begin(); p!=w2.end(); ++p) cout << *p << "   ";
  copy(w1.begin(), w1.end(), w2.begin());    //w2 napisana elementami w1
    cout << "\n lista w2 nadpisana w1 : "<< endl;
    for (p=w2.begin(); p!=w2.end(); ++p) cout <<" "<<*p << "   ";
  w2.erase(w2.begin(), w2.end()) ;     //przywracanie stanu poczatkowego
```

```
 w2.insert(w2.begin(), a2, a2 + R);  //skopiowanie elementow a1[] do w2
  cout << "\n     odtworzona lista w2 :" << endl;
  cout << " lista w2 : ";
  for (p=w2.begin(); p!=w2.end(); ++p) cout << *p << "   ";
 getche();
 return 0;
}
```

Wyjście programu wygląda następująco:

```
lista w1 : 1   3   5   7   9
lista w2 : 2   4   6   8   10
lista w2 nadpisana w1 :
1    3    5    7    9
   odtworzona lista w2 :
lista w2 : 2   4   6   8   10
```

Przy pomocy instrukcji:

```
list<int>::iterator p;
int a1[R]= { 1, 3, 5, 7, 9 };
int a2[R]= { 2, 4, 6, 8, 10 };
list<int> w1(a1, a1+5);       // inicjalizacja w1 liczbami a1
list<int> w2(a2, a2+5);       // inicjalizacja w2 liczbami a2
```

utworzony został iterator **p** oraz zdefiniowane zostały dwie klasyczne tablice **a1** i **a2.** Wykorzystano konstruktor klasy **list** do utworzenia obiektów klasy **list w1** oraz **w2**, zainicjalizowanych odpowiednio elementami tablic **a1** i **a2.** Przypominamy - przeciążony konstruktor klasy **list** w naszym przykładzie stosuje dwa iteratory jako argumenty. Wiemy już, że wskaźniki do tablic mogą być użyte, jako iteratory. W pokazanych instrukcjach tworzone są listy liczb całkowitych i inicjowane zawartością tablic od lokalizacji **a1** do lokalizacji **a1 + R** oraz od **a2** do **a2 + R**. Należy pamiętać, że lokalizacja **a1+R** oraz **a2+R** jest wyłączona.
Instrukcja postaci:

```
for (p=w1.begin(); p!=w1.end(); ++p) cout << *p << "   ";
```

służy do wyświetlenia elementów listy **w1** korzystając z dwóch iteratorów, operacji inkrementacji iteratora **p** oraz jego dereferencji. W instrukcji

```
copy(w1.begin(), w1.end(), w2.begin());
```

wywoływany jest algorytm **copy().** Kopiuje on wszystkie elementy z

pierwszego zakresu do zakresu docelowego. W naszym przykładzie zdefiniowany jest całkowicie jest pierwszy zakres (początek i koniec), dla drugiego zakresu zdefiniowany jest tylko początek. Algorytm **copy()** nadpisuje elementy (w pokazanym przypadku lista **w1** jest listą źródłową, a lista **w2** jest listą docelową, co oznacza, że lista **w2** otrzyma elementy listy **w1**). W dodatku algorytm **copy()** wymaga, aby kolekcja docelowa zawierała wystarczająco dużo elementów do nadpisania. Gdy tak nie jest, działanie algorytmu **copy()** jest niezdefiniowane. Aby uniknąć takich problemów można skorzystać z tzw. iteratorów wstawiających, lub zadbać o wystarczająca pojemność (użyć np. funkcji **resize()**). W kolejnych instrukcjach:

```
w2.erase(w2.begin(), w2.end()) ;   //przywracanie stanu poczatkowego
w2.insert(w2.begin(), a2, a2 + R); //skopiowanie elementow a1[] do w2
```

zastosowano funkcję **erase()** oraz funkcje **insert()** do odtworzenia pierwotnej postaci listy **w2** (przypominamy, że lista **w2** zawierała na początku elementy tablicy **a2**).
Funkcja składowa **erase()** usuwa wszystkie elementy listy z zakresu określonego jej argumentami, w naszym przypadku usunie wszystkie elementy.
Funkcja **insert()** jest dostępna dla każdego kontenera sekwencyjnego, zdefiniowane są jej trzy wersje. W naszym przykładzie ta funkcja używa drugiego i trzeciego argumentu do określenia początkowej i końcowej pozycji w sekwencji (u nas jest to klasyczna tablica) co ma taki skutek, że elementy tablicy **a2** są umieszczone w kontenerze **w2**.
Kolejny program ilustruje zastosowanie tzw. inserterów ogólnych, zwanych czasami wstawiaczami ogólnymi (ang. *general inserter*).

Wydruk 2.12. Kontener **list**, **back_insert(), front_insert(), insert(),**

```
#include <list>
#include <iostream.h>
#include <conio.h>
using namespace std;
const int R =5;

int main()
{ list<int>::iterator p;
  int a1[R]= { 1, 3, 5, 7, 9 };
  int a2[R]= { 2, 4, 6, 8, 10 };
  list<int> w1(a1, a1+5);      // inicjalizacja w1 liczbami a1
  list<int> w2(a2, a2+5);      // inicjalizacja w2 liczbami a2
  copy(w2. begin(), w2.end(), back_inserter(w1));      // back_inserter
    cout << "\n lista w1 = w1 + w2 , back_inserter: " << endl;
    for (p=w1.begin(); p!=w1.end(); ++p) cout << *p << "   ";
  list<int> w3(a1, a1+5);
```

```
list<int> w4(a2, a2+5);
copy(w4. begin(), w4.end(), front_inserter (w3));       // front_inserter
 cout << "\n lista w3 = (w3 +w4), front_insenter" << endl;
 for (p=w3.begin(); p!=w3.end(); ++p) cout << *p << "   ";
list<int> w5(a1, a1+5);
list<int> w6(a2, a2+5);
copy(w6.begin(), w6.end(), inserter (w5,w5.begin()));  // inserter
 cout <<" \n lista  w5 = (w5 + w6), inserter " << endl;
 for (p=w5.begin(); p!=w5.end(); ++p) cout << *p << "   ";

getche();
return 0;
}
```

W programie manipulujemy dwoma sekwencjami liczb całkowitych. Utworzone zostały dwie klasyczne tablice, sześć obiektów klasy **list** (**w1,w2,w3,w4,w5** i **w6**). Tablice wykorzystano, jako sekwencje, z których tworzymy elementy list.

W programie zostały utworzone dwie tablice

```
const int R = 5;
int a1[R] = { 1, 3, 5, 7, 9   };
int a2[R] = { 2, 4, 6, 8, 10 };
```

Wyjście programu wygląda następująco:

```
lista w1 = w1 + w2 , back_inserter:
1   3  5  7  9  2  4  6  8  10
lista w3 = (w3 + w4), front_inserter :
10  8  6  4  2  1  3  5  7  9
lista w5 = (w5 + w6), insertem
2  4  6  8  10  1  3  5  7  9
```

W programie manipulujemy dwoma sekwencjami liczb całkowitych. Utworzone zostały dwie klasyczne tablice, sześć obiektów klasy **list** (**w1,w2,w3,w4,w5** i **w6**). Tablice wykorzystano, jako sekwencje, z których tworzymy elementy list.
W instrukcjach:

```
list<int> w1(a1, a1+5);      // inicjalizacja w1 liczbami a1
list<int> w2(a2, a2+5);      // inicjalizacja w2 liczbami a2
```

tworzona są listy **w1** oraz **w2.** Sekwencje elementów w listach są następujące:

```
w1 = { 1, 3, 5, 7, 9   };
w2 = { 2, 4, 6, 8, 10 };
```

Operacja

```
copy(w2. begin(), w2.end(), back_inserter(w1));
```

produkuje nową sekwencję w kontenerze **w1**, algorytm **copy()** pobiera elementy listy **w2** z zakresu (początek, koniec) i umieszcza je w kontenerze **w1** na końcu (**w2** jest źródłem, **w1** jest kontenerem docelowym):

```
w1 = {1  3  5  7  9  2  4  6  8  10 }
```

W kolejnych instrukcjach tworzone są listy **w3** i **w4** z elementami:

```
w3 = { 1, 3, 5, 7, 9   };
w4  = { 2, 4, 6, 8, 10 };
```

Operacja

```
copy(w4. begin(), w4.end(), front_inserter (w3));
```

produkuje nową sekwencję w kontenerze **w3**, algorytm **copy()** pobiera elementy listy **w4** z zakresu(początek, koniec) i umieszcza je w kontenerze **w3** na początku (**w4** jest źródłem, **w3** jest kontenerem docelowym), elementy umieszczane są w odwrotnej kolejności:

```
w3 = { 10  8  6  4  2  1  3  5  7  9
```

W kolejnych instrukcjach tworzone są listy **w5** i **w6** z elementami:

```
w5 = { 1, 3, 5, 7, 9   };
w6 = { 2, 4, 6, 8, 10 };
```

Operacja

```
copy(w6.begin(), w6.end(), inserter (w5,w5.begin()));
```

produkuje nową sekwencję w kontenerze **w5**, algorytm **copy()** pobiera elementy listy **w6** z zakresu(początek, koniec) i umieszcza je w kontenerze **w5** na początku (**w6** jest źródłem, **w5** jest kontenerem docelowym), elementy umieszczane są w tej samej kolejności jak były umieszczone w kontenerze źródłowym:

```
w5 = { 2  4  6  8  10  1  3  5  7  9 }
```

Wykorzystana w algorytmie **copy()** funkcja składowa **inserter()** inicjalizowana jest dwiema wartościami: kontenerem oraz pozycją. Wstawiacz ogólny (nazywany czasami iteratorem wstawiającym ogólnym albo iteratorem wstawiającym lub wstawiaczem) wywołuje funkcję składową **insert()** z podaną pozycją, jako argumentem. Wstawiacz ogólny dostępny jest dla wszystkich typów kontenerów, ponieważ każdy kontener ma funkcję składową **insert().** W naszym przykładzie w zapisie:

```
inserter ( w5, w5.begin() ) ;
```

w5 oznacza kontener docelowy a **w5.begin()** miejsce wstawiania (w tym przypadku początek kontenera). Należy zwrócić uwagę na różne działania inserterów.
W tabeli 2.9 podano opis działania inserterów wstawiających.

Tabela 2.9. Iteratory wstawiające

Lp.	Insertem	Opis
1	back_inserter(zasobnik)	Dołącza w tej samej kolejności korzystając z **push_back()**
2	front_inserter(zasobnik)	Wstawia na początek w odwrotnej kolejności, korzystając z **push_front()**
3	inserter(kontener,poz)	Wstawia na pozycji **poz** (w tej samej kolejności) korzystając z **insert()**

W tabeli 2.10 przedstawiono możliwości wykonywania wybranych operacji na kontenerach.

Tabela 2.10. Wybrane dozwolone operacje na kontenerach

	Operacja	Funkcja	vector	deque	list	inne kontenery
1	Wstawianie na końcu	push_back	tak	tak	tak	nie
2	Usuwanie na końcu	pop_back	tak	tak	tak	nie
3	Wstawianie na początku	push_front	nie	tak	tak	nie
4	Usuwanie na	pop_front	nie	tak	tak	nie

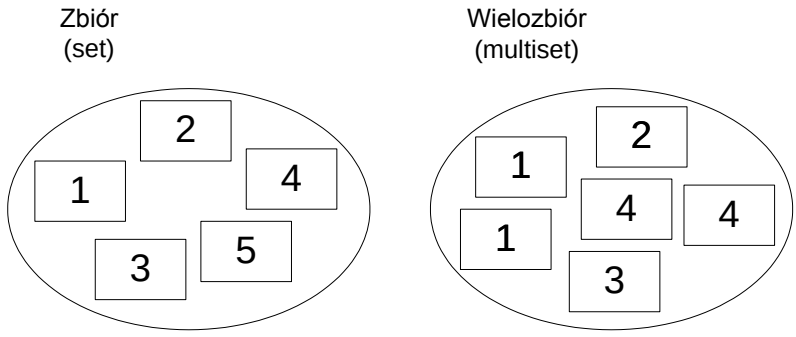
	początku					
5	Wstawianie gdziekolwiek	insert	tak	tak	tak	tak
6	Usuwanie gdziekolwiek	erase	tak	tak	tak	tak
7	Sortowanie	sort	tak	tak	tak	tak

2.6. Kontenery asocjacyjne

Kontenery asocjacyjne (znane także, jako kontenery skojarzeniowe, kontenery stowarzyszone lub zasobniki skojarzeniowe) umożliwiają bezpośredni dostęp do elementów kontenerów przez klucze (zwanymi także kluczami wyszukiwawczymi). Elementy nie są przechowywane w konfiguracji liniowej. Kontenery asocjacyjne zapewniają odwzorowanie kluczy na wartości. W kontenerach asocjacyjnych czas operacji wstawiania, usuwania i wyszukiwania jest porównywalny.
Mamy cztery typy kontenerów asocjacyjnych:

- set
- multiset
- map
- multimap

W praktyce bardzo często tworzymy struktury danych. W firmach przechowywane są rekordy pracowników: nazwisko, rok urodzenia, miejsce urodzenia, telefon domowy, itp. Do wyszukania w bazie pracowników odpowiedniego pracownika dobrze jest mieć ustalony klucz (może to być unikalny numer pracownika albo np. PESEL). Dzięki kluczowi szybko odnajdujemy interesującą nas strukturę i dzięki temu mamy szybki dostęp do konkretnych danych o pracowniku. Praktyczne znaczenie kontenerów asocjacyjnych polega na tym, że umożliwiają bardzo szybki (można by rzec - błyskawiczny) dostęp do elementów.



Rys. 2.2 Schematyczne przedstawienie zbiorów i wielozbiorów

2.7. Kontenery asocjacyjne - set

Kontener **set** (zbiór) najczęściej używany jest do zapamiętywania i odzyskiwania kluczy, przy czym w kontenerze **set** sama wartość jest kluczem. Kontener nie dopuszcza do przechowywania powtarzających się wartości, dane przechowywane są w postaci uporządkowanej. Klasa **set** obsługuje iteratory dwukierunkowe, (ale nie iteratory o dostępie bezpośrednim), **set** nie posiada operatora [ ].
Specyfikacja szablonu **set** ma postać:

```
#include <set>
template <class Key, class Compare = less<Key>,
class Allocator = allocator<Key> >
class set ;
```

W pokazanym wzorcu klasy set<Key, Compare, Allocator> tylko pierwszy argument musi występować. Elementami zbioru mogą być dowolne typy **Key**, które umożliwiają operacje przypisania, kopiowania i porównywania zgodnie z kryterium sortowania. Drugi argument jest opcjonalny, definiuje kryterium sortowania i jeżeli nie zostanie wymieniony to domyślnie będzie rodzaju **less** (korzysta z operatora <). Trzeci argument także może być pominięty, definiuje on model pamięci, opcjonalnym jest model **allocator**. Uporządkowanie w zbiorze musi być antysymetryczne, przechodnie i niezwrotne. Tak jak wszystkie asocjacyjne klasy kontenerowe, zbiory są implementowane, jako zrównoważone drzewa binarne (a mówiąc dokładniej, jako drzewa czerwono-czarne, ang. *red-black trees*).

Klasa **set** ma następujące konstruktory:

```
explicit
set(const Compare& comp = Compare(), const Allocator& alloc =
    Allocator());
```

Kreuje pusty zbiór elementów. Jeżeli funkcyjny obiekt **comp** nie jest wyszczególniony, domyślnie jest wykorzystany **less** (<). Allokator **alloc** jest wykorzystywany do zarządzania pamięcią.

```
template <class InputIterator>
set(InputIterator first, InputIterator last,
const Compare& comp = Compare()
const Allocator& alloc = Allocator());
```

Kreuje zbiór elementów o długości **last - first**, przechowuje wszystkie elementy otrzymane po dereferencji **InputIterators** w przedziale [**first, last**). Jeżeli funkcyjny obiekt **comp** nie jest wyszczególniony, domyślnie jest wykorzystany

less (<). Allokator **alloc** jest wykorzystywany do zarządzania pamięcią.

set(const set<Key, Compare, Allocator>& x);

Konstruktor kopiujący, kreuje kopię **x**.

Destruktor klasy **set** ma postać:

~set();

Zwalnia całą pamięć przydzieloną zbiorowi.

W zbiorach realizowane jest automatyczne sortowanie. Wprowadza to istotne ograniczenia – nie można bezpośrednio zmieniać wartości elementów. Mamy możliwości zmiany elementów zbioru, wymaga to dość skomplikowanych zabiegów. Na zbiorach STL możemy wykonać wiele operacji znanych dla matematycznych zbiorów, takich jak suma, przecięcie czy różnica. Suma dwóch zbiorów to zbiór zawierający elementy dwóch zbiorów z wyłączeniem powtórzeń. Przecięcie dwóch zbiorów to zbiór, który zawiera ich wspólne elementy. Różnica dwóch zbiorów to zbiór elementów pierwszego zbioru, które nie pojawiły się w drugim zbiorze. W bibliotece STL znajdują się algorytmy do obsługi wymienionych operacji na zbiorach takie jak na przykład **set_union(), set_intersection()** czy **set_difference().** W kolejnym programie demonstrujemy operacje wykonywane na kontenerze **set**.

Wydruk 2.13. Kontener **set, insert(), erase(), set_union(),**

```
#include <set>
#include <algorithm>
 #include <iostream.h>
using namespace std;
const int R =5;
int main()
{ set<int> s1,s2, s3, s4, s5;       // deklarowane sa zbiory
  set <int> :: iterator p;
  insert_iterator <set <int> > w(s5, s5.begin());
  int a1[R]= { 1, 3, 5, 7, 9 };
  int a2[R]= { 2, 4, 6, 8, 10 };
  s1.insert(2);       s1.insert(4);
  s1.insert(9);       s1.insert(1);
   cout <<"ins.  s1 = ";
   for (p=s1.begin(); p!=s1.end(); ++p) cout << *p << "   ";
  s2 = s1;
   cout << "\n copy s2 = ";
   for (p=s2.begin(); p!=s2.end(); ++p) cout << *p << "   ";
   cout << "\n a1  s3 = ";
```

```
 s3.insert(a1,a1+R);
  for (p=s3.begin(); p!=s3.end(); ++p) cout << *p << "   ";
  cout << "\n a2   s4 = ";
 s4.insert(a2,a2+R);
  for (p=s4.begin(); p!=s4.end(); ++p) cout << *p << "   ";
  cout << "\na1+a2,s5 = " ;
 set_union(a1,a1+R, a2,a2+R, w);
  for (p=s5.begin(); p!=s5.end(); ++p) cout << *p << "   ";
  cout << "\nerase s5 = " ;;
 s5.erase(6);
  for (p=s5.begin(); p!=s5.end(); ++p) cout << *p << "   ";
  cout << endl;
 return 0;
}
```

Wyjście programu ma następującą postać:

```
 ins.    s1 = 1   2   4   9
  copy s2 = 1   2   4   9
 a1       s3 = 1   3   5   7   9
 a2       s4 = 2   4   6   8   10
a1+a2,s5 = 1   2   3   4   5   6   7   8   9   10
erase s5 =  1   2   3   4   5   7   8   9   10
```

W programie musimy dołączyć plik nagłówkowy <set>, aby można było używać kontenerów **set.**
W instrukcjach:

```
set<int> s1,s2, s3, s4, s5;
set <int> :: iterator p;
```

tworzone są obiekty klasy **set,** kontenery **s1,s2,s3,s4** i **s5** oraz deklarowany jest iterator **p**. Zbiory przechowują wartości typu **int** i porządkowane są w porządku rosnącym (obiekt funkcji **less<int>,** domyślny).
W kolejnej instrukcji:

```
insert_iterator <set <int> > w(s5, s5.begin());
```

utworzyliśmy iterator **insert_iterator** dla kontenera **s5**. Deklaracja tego iteratora posiada dodatkowy argument konstruktora, który pozwala określić miejsce wstawiania elementów. Potrzeba deklaracji typu kontenera wynika z tego, że iterator musi używać odpowiednich metod kontenera. Opisany iterator należy do grupy adaptatorów iteratorów. Są to specjalne iteratory umożliwiające algorytmom na działanie w trybie odwrotnym, w trybie wstawiania oraz współpracę ze strumieniami. W pokazanym przykładzie użyliśmy iteratora wstawiającego. Iteratory wstawiające (zwane czasami

wstawiaczami) przekształcają operacje przypisania nowej wartości w operacje wstawiania tej wartości. W ten sposób dzięki wstawiaczom algorytmy mogą wstawiać elementy, zamiast je nadpisywać.
Wszystkie iteratory wstawiające należą do kategorii iteratorów wyjściowych. W tabeli 2.11 pokazano rodzaje iteratorów wstawiających.

Tabela 2.11. Rodzaje iteratorów wstawiających.

Iterator	Klasa	Wywoływana funkcja	Kontenery
Końcowy	back_insert_iterator	push_back(v)	vector, deque,list
Początkowy	front_insert_iterator	push_front(v)	deque,list
Ogólny	insert_iterator	insert(pos,v)	wszystkie

W instrukcjach typu:

```
s1.insert(2);
```

korzystamy z funkcji **insert()**, aby dodać wartość do obiektu **s1.** W pokazanym przykładzie wartość 2 będzie umieszczona w kontenerze **s1**.
W instrukcji:

```
s2 = s1;
```

mamy do czynienia z instrukcją przypisania zbiorów. Ta instrukcja powoduje przypisanie wszystkich elementów zbioru **s1** do kontenera **s2.** Obydwa kontenery muszą być tego samego typu, w szczególności taki sam musi być typ kryterium porównania. Jeżeli mamy zadeklarowaną i zainicjalizowaną klasyczną tablicę:

```
int a1[R]= { 1, 3, 5, 7, 9 };
```

to możemy wykorzystać ją do wstawienia wszystkich jej elementów z zakresu [**first,end**) do kontenera **s3** przy pomocy jednej z trzech wersji funkcji **insert()**:

```
s3.insert(a1,a1+R);
```

W kolejnej instrukcji:

```
s5.erase(6);
```

usunęliśmy element z kontenera **set** korzystając z jednej z trzech wersji funkcji **erase().** W tym przypadku funkcja usuwa wyspecyfikowany element (liczbę 6) ze zbioru **s5**. Nie wolno podejmować próby usuwania elementu z pustego

kontenera.
W instrukcji:

```
set_union(a1,a1+R, a2,a2+R, w);
```

wykorzystaliśmy algorytm **set_union(),** aby utworzyć sumę dwóch zbiorów, zbiór pierwszy zawiera elementy tablicy **a1[ ],** zbiór drugi zawiera elementy tablicy **a2[ ],** suma zbiorów umieszczona jest w kontenerze s5, wykorzystaliśmy utworzony uprzednio iterator postaci:

```
insert_iterator <set <int> > w(s5, s5.begin());
```

Funkcja **set_union()** przyjmuje pięć argumentów w postaci iteratorów. Dwa pierwsze określają zakres w pierwszym zbiorze, dwa następne określają zakres w drugim zbiorze, a ostatni iterator to iterator wyjścia, który określa miejsce, do którego przekopiowany zostanie zbiór wyjściowy. W naszym przypadku musi być określona także nazwa docelowego kontenera. Można to wykonać tworząc anonimowy obiekt **insert_iterator().** Konstruktor tego iteratora pobiera dwa argumenty: nazwę kontenera docelowego i iterator wskazujący pozycję wstawiania.

W operowaniu kontenerami asocjacyjnymi bardzo przydatna jest klasa narzędziowa **pair().** Zdefiniowana jest ona w pliku nagłówkowym <utility>. Typ **pair** używany jest wszędzie tam, gdzie zachodzi potrzeba traktowania dwóch wartości, jako pojedynczego elementu.

Typ **pair** zadeklarowany został w STL, jako struktura, żeby wszystkie jego składowe były publiczne:

```
template <class T1, class T2>
 struct pair {
      typedef T1 first_type;
      typedef T2 second_type;
      T1 first;
      T2 second;
      pair();
      pair (const T1&, const T2&);
      template <class V, class U>
      pair (const pair <V, U>& p);
      ~pair();
};
```

Konstruktor domyślny tworzy parę wartości z wybranych typów. W ten sposób deklaracja:

```
pair < int, double > p
```

powoduje inicjalizację wartości pary **p** za pomocą konstruktorów **int()** oraz **double(),** co w rezultacie daje wartości zerowe.

Jeżeli istnieją obiekty klasy **pair(),** możemy uzyskać dostęp do jej składników poprzez pola **first** oraz **second**. Użycie obiektu **pair()** pokazane jest w kolejnym programie, parę o składowych typu **int** oraz **string** tworzą numer telefonu i nazwisko .

Wydruk 2.14. Klasa **pair()**

```
#include <utility>
#include <iostream.h>
#include <conio.h>
using namespace std;

int main()
{ pair <int, string> a1(5373105,"Kowalski");
  pair <int, string> a2(5265153,"Zawalski");
  cout << a1.second <<", tel.   " << a1.first<<endl;
  cout << a2.second <<", tel.   " << a2.first<<endl;

 getche();
 return 0;
}
```

Po uruchomieniu tego programu mamy następujący wydruk:

```
Kowalski, tel.   5373105
Zawalski, tel.   5265153
```

Obiekt struktury **pair** może być bardzo skomplikowany, tworzy się go dla konkretnych typów, (np. dla kontenerów asocjacyjnych), czasami te obiekty są niezbędne do wykonania poszczególnych operacji na kontenerach. W szczególności utworzona para dla kontenera **set** w postaci:

```
pair <set<int> :: iterator, bool> stan;
```

deklaruje parę składająca się z iteratora dla set<int> i wartości typu **bool**. Jeżeli np. obiekt **stan** będzie przechowywał wynik wywołania do kontenera **set** funkcji **insert(),** to w wyniku otrzymamy parę **stan,** która zawiera iterator **stan.first** wskazujący na wartość typu **int** w obiekcie **set** oraz wartość **bool** (stan**.second**), która jest prawdą (**true**), jeżeli wartość została wstawiona lub fałszem (**false**), gdy wartość nie została wstawiona (taka możliwość istnieje, jeżeli w zbiorze istnieje już wstawiana wartość).

Nie jest sprawą trywialną ustalenie pozycji w kontenerach asocjacyjnych. Istnieje specjalna funkcja pomocnicza dla iteratorów - **distance(),** która zwraca

odległość pomiędzy dwoma iteratorami. Pobiera ona dwa argumenty – iteratory wejściowe **pos1** oraz **pos2** i zwraca odległość pomiędzy nimi. Obydwa iteratory muszą odnosić się do elementów tego samego kontenera. Dla iteratorów dostępu swobodnego funkcja **distance()** zwraca po prostu wynik wyrażenia **pos2 – pos1.** W przypadku pozostałych kategorii iteratorów, iterator **pos1** jest inkrementowany tak długo, aż osiągnie pozycję iteratora **pos2**, a zwracana jest liczba tych inkrementacji. W kolejnym programie zastosujemy obiekt typu **pair** oraz funkcje składową **distance()** do ustalenia pozycji, na jakiej umieszczamy wstawianą wartość do zbioru.

Wydruk 2.15. Kontener **set, insert(), distance(), pair()**

```
#include <set>
#include <iostream.h>
#include <conio.h>
using namespace std;

int main()
{ set<int> s1;
  set <int> :: iterator p;
  pair <set<int> :: iterator, bool> stan;
 s1.insert(2);
 s1.insert(4);
 s1.insert(9);
 s1.insert(1);
   cout <<" s1 = ";
   for (p=s1.begin(); p!=s1.end(); ++p) cout << *p << "   ";
   cout <<"\n   wstawiono 7 do zbioru s1\n s1 = ";
 stan = s1.insert(7);
   for (p=s1.begin(); p!=s1.end(); ++p) cout << *p << "   ";
   cout << "\nna pozycji = " << distance(s1.begin(), stan.first) +1;
 getche();
 return 0;
    }
```

Wynik działania programu ma postać:

```
s1 = 1   2   4   9
   wstawiono 7 do zbioru s1
s1 = 1   2   4   7   9
na pozycji = 4
```

W przedstawionym programie tworzony jest zbiór o nazwie **s1** i inicjalizowany wartościami typu **int.**

W instrukcji:

```
pair <set<int> :: iterator, bool> stan;
```

deklarujemy parę składającą się z iteratora i wartości typu **bool**.
W kolejnej instrukcji:

```
stan = s1.insert(7);
```

używamy funkcji **insert()** do umieszczenia wartości 7 w obiekcie **set.**
Zwracana para **stan** zawiera iterator i wartość typu **bool**.
W instrukcji:

```
cout << "\nna pozycji = " << distance(s1.begin(), stan.first) +1;
```

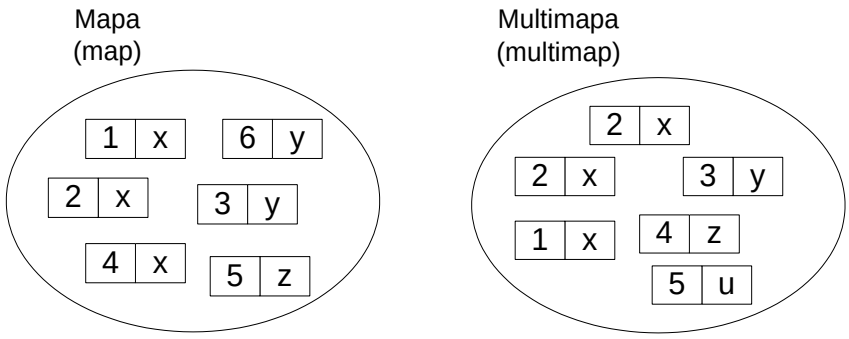
wykorzystano funkcję **distance()** do ustalenia pozycji wstawianej wartości 7 w zbiorze **s1**. Wynik wyświetlony jest na ekranie monitora.

2.8. Kontenery asocjacyjne - multiset

Kontener **multiset** służy do szybkiego zapamiętywania i odzyskiwania kluczy, ale w przeciwieństwie do kontenera **set** dopuszcza duplikowanie kluczy. Podobnie jak w zbiorze **set**, elementy są uporządkowane (porządek rosnący jest domyślnym). Kontener **multiset** obsługuje wszystkie operacje dostępne w **set**.

2.9. Kontenery asocjacyjne - map

Kontener **map** uważany jest za jeden z bardziej przydatnych kontenerów. Przechowuje on pary klucz – wartość. Operacje wstawiania, wyszukiwania i usuwania korzystają z wartości klucza. Nazwa **map** związana jest z funkcjonalnością kontenera – odwzorowuje (mapuje) klucze na wartości. Duplikaty kluczy nie są możliwe w obiekcie **map**, (ale są dopuszczalne w kontenerze **multimap**). Jest to mapowanie jeden - do jednego. Kontener **map** przechowuje elementy w postaci uporządkowanej względem kluczy. Istotną zaletą **map** jest umożliwienie wyszukiwania wartości na podstawie klucza. Bardzo często tworzymy spis znajomych oraz ich numerów telefonów. Możemy utworzyć mapę, służącą do przechowywania nazwisk i numerów telefonów, nazwisko będzie kluczem.



Rys. 2.3 Schematyczne przedstawienie **map** i **multimap**

Specyfikacja szablonu **map** ma postać:

```
#include <map>
template <class Key, class T, class Compare = less<Key>
        class Allocator = allocator<pair<const Key, T>> >  class map;
```

Zgodnie ze specyfikacją mapa map <Key, T, Compare, Allocator> daje szybki dostęp do przechowywanych wartości typu **T**, które są indeksowane unikalnym kluczem typu **Key**. **Comp** jest funkcją porównującą dwa klucze. Domyślnie stosowana jest funkcja **less()** (operator <). Ponieważ elementami **map** są pary klucz-wartość, **map** ma dwukierunkowy iterator, który wskazuje na parę <const Key x, T y> , gdzie **x** jest kluczem a **y** jest przechowywaną wartością powiązaną z tym kluczem. Definicja mapy zawiera definicje typu (**typedef**) tej pary, **value_type** oznacza pair< Key, T >.

Para klucz - wartość przechowywana jest w mapie, jako obiekt para (**pair**). W strukturze **pair**, wartość **first** zawiera klucz, wartość **second** zawiera wartość z nim związaną.
Klasa **map** ma następujące konstruktory:

```
explicit map(const Compare& comp = Compare(),
                const Allocator& alloc = Allocator());
```

Kreuje pusty zbiór elementów. Jeżeli funkcyjny obiekt **comp** nie jest wyszczególniony, domyślnie jest wykorzystany **less** (<). Allokator **alloc** jest wykorzystywany do zarządzania pamięcią.

```
template <class InputIterator>
map(InputIterator first, InputIterator last,
      const Compare& comp = Compare(),
      const Allocator& alloc = Allocator());
```

Kreuje zbiór elementów, przechowuje wszystkie elementy w przedziale [**first, last**). Utworzenie nowej mapy jest możliwe tylko w przypadku, gdy iteratory **first** i **last** zwracają wartości typu pary pair<classKey, class Value> i wszystkie wartości **Key** w zakresie [**first, last**) są unikalne. Jeżeli funkcyjny obiekt **comp** nie jest wyszczególniony, domyślnie jest wykorzystany **less** (<). Allokator **alloc** jest wykorzystywany do zarządzania pamięcią.

map(const map<Key,T,Compare,Allocator>& x);

Konstruktor kopiujący, kreuje kopię **x**, utworzona zostaje nowa mapa przez skopiowanie wszystkich par kluczy i wartości z **x.**

Destruktor klasy **map** ma postać:

~map();

Zwalnia całą przydzieloną mapie pamięć.

Do wstawiania elementów do mapy możemy skorzystać z trzech wersji funkcji **insert().** Opis tych funkcji przedstawiony jest w tabeli 2.12.

Tabela 2.12. Operacje wstawiania elementów do kontenera map

Lp.	Operacja	Opis
1	c.insert(v)	Wstawia kopię elementu **v**
2	c.insert(pos,v)	Wstawia kopię elementu **v** i zwraca pozycję **pos**
3	c.insert(beg,end)	Wstawia kopię wszystkich elementów z zakresu [**beg,end**)

Należy pamiętać, że elementy są dla kontenera **map** parami klucz-wartość. Istnieją cztery sposoby wstawiania wartości do mapy:

- użycie składowej **value_type**
- użycie struktury **pair<>**
- użycie funkcji **make_pair()**
- wykorzystanie operatora **[ ]**

W programie pokazujemy sposoby wstawiania elementów do kontenera **map,** wykorzystując cztery wymienione techniki. W programie tworzymy notatnik z nazwiskami i telefonami.

Wydruk 2.16. Kontener **map, insert(), pair(), make_pair()**

```
#include <map>
#include <iostream>
#include<iomanip>       //setw()
#include<string>
#include <conio.h>
using namespace std;

int main()
{ typedef map<string, int> msi;
 typedef pair<string, int> pa;
 map<string,int>:: iterator p;
 string naz;
 msi k_tel;
 k_tel.insert(msi::value_type("Kowalski", 5264455));   //value_type
 k_tel.insert(msi::value_type("Zawalski", 5264466));
  cout << "podaj nazwisko : ";
  cin >> naz;
  p = k_tel.find(naz);
  cout <<" pan "<< naz << " ma nr tel. " <<p->second;
 k_tel.insert(pa("Lipski", 5264477));                    //pair<>
 k_tel.insert(make_pair("Bielski", 5264488));            //make_pair()
 k_tel["Rogalski"] = 5264499;              //operator[ ]
 cout << "\nNotatnik telefoniczny " << endl;
 for(p = k_tel.begin(); p != k_tel.end(); ++p)
  cout << setw(10)<<p->first << "  tel.  " << p->second << endl;
 getche();
 return 0;
  }
```

Wyjście programu wygląda następująco:

```
podaj nazwisko : Kowalski
   pan Kowalski ma nr tel. 5264455
Notatnik telefoniczny
   Bielski  tel. 5264488
Kowalski  tel. 5264455
    Lipski  tel. 5264477
Rogalski  tel. 5264499
Zawalski  tel. 5264466
```

W instrukcjach:

```
typedef map < string, int > msi ;
typedef pair < string, int > pa ;
```

tworzymy synonimy (aliasy) dla zdefiniowanych typów danych - map i **pair**. Jest to często stosowane w STL uproszczenie, aby nie powtarzać długich deklaracji. W kolejnych instrukcjach:

```
map < string, in t> :: iterator p ;
 msi k_tel;
```

tworzony jest iterator **p** do obsługi mapy oraz tworzony jest kontener **map** o nazwie **k_tel.** Aby wstawić parę klucz-wartość możemy skorzystać z kilku technik. W pierwszym przykładzie korzystamy ze składowej **value_type**. Sposób użycia tej metody pokazany jest w instrukcjach:

```
k_tel.insert(msi::value_type("Kowalski", 5264455));    //value_type
k_tel.insert(msi::value_type("Zawalski", 5264466));
```

Kolejna metoda umieszczania elementów w kontenerze **map** pokazana jest w instrukcji:

```
k_tel.insert(pa("Lipski", 5264477));          //pair<>
```

Argumentem funkcji **insert()** jest obiekt **pair** z danymi. Pokazana instrukcja jest skondensowana, w bardziej czytelnej formie ma postać:

```
pair<string, int> pa("Lipski", 5264477);
k_tel.insert(pa) ;
```

Można to samo można wykonać tworząc anonimowy obiekt **pair** i wstawić go do kontenera:

```
k_tel.insert(pair<string, int> ("Lipski", 5264477)) ;
```

Jednym z wygodniejszych sposobów wstawiania elementów do kontenera **map** jest wykorzystanie funkcji **make_pair(),** tak jak to zostało wykonane w kolejnej instrukcji programu:

```
k_tel.insert(make_pair("Bielski", 5264488));    //make_pair()
```

Funkcja **make_pair()** tworzy obiekt typu **pair** zawierający dwie wartości przekazywane, jako argumenty.

Ostatnim sposobem wstawiania elementów do kontenera **map** jest wykorzystanie operatora [ ], co w praktyce sprowadza się do użycia indeksu. Kontenery asocjacyjne nie zapewniają normalnie możliwości uzyskania bezpośredniego dostępu do elementów.

Pamiętamy, że to odpowiednie iteratory umożliwiają dostęp do elementów. W przypadku **map** mamy wyjątek. Mapy, które nie są stałe, udostępniają operator

indeksowy. Indeksem w takim przypadku jest klucz służący do identyfikacji elementu (w praktyce to oznacza, że indeks może być dowolnego typu!). Opisana metoda jest prosta, ale oprócz zalet ma też wady, dlatego należy ostrożnie wybierać ten sposób umieszczania danych. Instrukcja programu:

```
k_tel [ "Rogalski" ] = 5264499 ;        //operator[ ]
```

ilustruje technikę wykorzystania indeksu mapy do wstawiania pary klucz-wartość do kontenera **map**. Metoda **find():**

```
cin >> naz;
p = k_tel.find(naz);
```

pozwala nam na wyszukanie wartości, jeżeli podamy odpowiedni klucz. Funkcja **find()** zwraca iterator wskazujący odpowiedni element lub, jeżeli nie zostanie podany prawidłowy klucz, wskazany będzie koniec mapy. Po odszukaniu klucza, związana z nim wartość jest umieszczona w składowej **second** obiektu **pair:**

```
cout <<" pan "<< naz << " ma nr tel. " << p->second;
```

Na koniec przy pomocy instrukcji **for:**

```
for (p = k_tel.begin(); p != k_tel.end(); ++p)
    cout << setw(10)<< p -> first << "  tel.  " << p -> second << endl;
```

wykonamy pełny wydruk klucz-wartość elementów kontenera **map**, korzystając ze składowych obiektu **pair** – **first** i **second**.

2.10. Kontenery asocjacyjne - multimap

Kontener asocjacyjny multimapa (ang. *multimap*), który działa podobnie jak kontener mapa, jest także użytecznym zasobnikiem. Przechowuje pary klucz – wartość, ale klucze mogą się powtarzać. Elementy kontenera multimapa są automatycznie sortowane, gdy określimy kryterium sortowania wobec bieżącego klucza. Multimapa posiada dwukierunkowy iterator.
Specyfikacja szablonu ma postać:

```
#include <map>
template <class Key, class T, class Compare = less<Key>,
    class Allocator = allocator<pair<const Key, T>> > class multimap;
```

Zgodnie ze specyfikacją multimapa multimap <Key ,T, Compare, Allocator> daje szybki dostęp do przechowywanych wartości typu **T**, które są indeksowane kluczami typu **Key**. Domyślną operacją porównywania kluczy jest **less** (obiekt

funkcyjny **less** sortuje elementy przy użyciu operatora <). Multimapa używa dwukierunkowego iteratora, który wskazuje na parę pair<const Key x, T y>, gdzie **x** jest kluczem a **y** jest przechowywaną wartością zwiazaną z tym kluczem. Definicja multimapy zawiera definicję typu (**typedef**) tej pary o nazwie **value_type**. W przypadku **map** i **multimap** jest to struktura pair<const typ_klucza, typ_wartosci>.

Klasa **multimap** ma następujące konstruktory:

```
explicit multimap(const Compare& comp = Compare(),
                  const Allocator& alloc = Allocator());
```

Kreuje pusty zbiór elementów, użyta jest opcjonalna relacja **comp** do uporządkowania kluczy oraz alokatora **alloc** do zarządzania pamięcią.

```
template <class InputIterator>
multimap(InputIterator first, InputIterator last,
         const Compare& comp = Compare()
         const Allocator& alloc = Allocator());
```

Kreuje zbiór elementów, przechowuje wszystkie elementy w przedziale [**first, last**). Utworzenie nowej multimapy jest możliwe tylko wtedy, gdy iteratory **first** i **last** zwracają wartości typu pary pair<class Key, class T>.

```
multimap(const multimap<Key, T, Compare, Allocator>& x);
```

Konstruktor kopiujacy, kreuje nową multimapę kopiując wszystkie pary kluczy i wartości z **x.**

Destruktor klasy ma postać:

```
~multimap();
```

Zwalnia całą przydzieloną multimapie pamięć.

Należy zauważyć, że w pliku <map> typy **map** i **multimap** zdefiniowane są, jako wzorce klas w przestrzeni nazw **std**.

Do wstawiania elementów do multimapy korzystamy z identycznego zestawu operacji, jaki został zaprezentowany dla **map** (tabela 2.12).

Kolejny program ilustruje wykorzystanie kontenera **multimap.** Do wstawiania elementów do kontenera **multimap** wykorzystano metodę **insert()** z użyciem funkcji **make_pair().** Do programu musimy dołączyć plik nagłówkowy <map> aby można było używać klasy **multimap**.

Wydruk 2.17. Kontener **multimap, insert(), make_pair()**

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{ typedef multimap<int, string> AM ;
  AM lilije;
  lilije.insert(make_pair(6,"pana"));
  lilije.insert(make_pair(2,"to"));
  lilije.insert(make_pair(4,"pani"));
  lilije.insert(make_pair(1,"Zbrodnia"));
  lilije.insert(make_pair(3,"nieslychana"));
  lilije.insert(make_pair(4,"(panna)"));
  lilije.insert(make_pair(5,"zabija"));
  AM :: iterator n;
  for(n = lilije.begin(); n != lilije.end(); ++n)
     cout << n -> second << ' ';
  cout << endl;
   cin.get();
   return 0;
}
```

W wyniku działania programu zostaje wyświetlony tekst:

Zbrodnia to nieslychana pani (panna) zabija pana

Jest to początek znanej ballady Adama Mickiewicza zatytułowanej Lelije z roku 1820, dodatkowy napis w nawiasie (***panna***) wygenerował nasz program w celach dydaktycznych.
W instrukcji:

typedef multimap<int, string> AM ;

wykorzystano wyrażenie **typedef** do definicji typu **multimap** (synonim **AM**), gdzie typem klucza jest **int**, typem skojarzonej wartości jest **string**, a elementy (przez domniemanie) są uporządkowane rosnąco.
Taką konstrukcję stosujemy zazwyczaj aby zastąpić skomplikowaną nazwę typu prostszym symbolem. Nie tworzymy osobnego typu a jedynie synonim. W instrukcjach:

```
AM lilije;
lilije.insert(make_pair(6,"pana"));
```

tworzymy obiekt klasy **multimap** o nazwie **lilije** (taki tytuł nosi ballada A. Mickiewicza) oraz wykorzystując funkcję **insert()** dodajemy nowe pary klucz-wartość do obiektu **multimap**. W tej konkretnej instrukcji klucz 6 jest skojarzony z napisem „pana". Ponieważ elementy kontenera są parami, do ich wstawiania wykorzystano wygodną funkcję **make_pair().** Funkcja **make_pair()** tworzy obiekty typu **pair**, które reprezentują pary wartości. W kolejnych instrukcjach:

```
AM :: iterator n;
for (n = lilije.begin( ); n ! = lilije.end( ); ++n )
   cout << n -> second << ' ';
```

tworzymy iterator kontenera. Program wyświetli wszystkie elementy kontenera w pętli **for** przy pomocy iteratora kontenera **n**.

Wewnątrz pętli iterator zostaje zainicjalizowany pierwszym elementem kontenera, a następnie przesuwa się po kolejnych elementach do momentu, gdy osiągnie pozycję ostatniego elementu. Operator inkrementacji kontenera jest tak zdefiniowany, że bierze pod uwagę drzewiastą strukturę kontenera. Ponieważ mamy obiekty **pair**, możemy uzyskać dostęp do jego składników używając składowych **first** oraz **second**. W naszym przykładzie żądamy wyświetlenia składowej **second** (czyli wartości typu **string** skojarzonej z kluczem **int**).

Kolejny przykład pokazuje użycie multimapy do notowania temperatury dni. Elementami multimapy są pary, w których kluczem jest temperatura a wartością jest nazwa dnia tygodnia. W programie zmieniono porządek – klucze są przechowywane od największego do najmniejszego, pokazano wykorzystanie metod **count()** oraz **find().** Metoda **insert()** wykorzystuje składową **value_type** do wstawiania par do kontenera.

Wydruk 2.18. Kontener **multimap, insert(), value_type(), count()**

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{ typedef multimap<int, string, greater<int> > tem;
 tem rap;
 rap.insert(tem::value_type(25,"monday"));
  rap.insert(tem::value_type(21,"tuesday"));
  rap.insert(tem::value_type(19,"wednesday"));
  rap.insert(tem::value_type(20,"thursday"));
  rap.insert(tem::value_type(22,"friday"));
  rap.insert(tem::value_type(21,"saturday"));
  rap.insert(tem::value_type(20,"sunday"));
```

```
 tem :: iterator n;
  for ( n = rap.begin(); n != rap.end(); ++n)
      cout << n->first << "     " << n->second <<endl;
 int ile = 21;
 cout <<"dni z temperatura = " << ile << " stopni to:" <<endl;
 for ( n = rap.begin(); n != rap.end(); ++n)
     if (n->first == ile) cout << '\t' <<n->second << endl;
 ile = 20;
 cout <<"dni z temperatura = " << ile << " stopni bylo ";
 cout << rap.count(20) << endl;
 ile = 19;
 n = rap.find(ile);
 cout <<"dzien z temperatura = " << ile << " stopni to: " << n->second;
cin.get();
return 0;
}
```

Wyjście programu wygląda następująco:

```
25 monday
22 friday
21 tuesday
21 saturday
20 thursday
20 sunday
19 wednesday
dni z temperatura 21 stopni to:
    tuesday
    saturday
dni z temperatura = 20 stopni było 2
dzien z temperatura = 19 stopni to: wednesday
```

Instrukcje:

```
typedef multimap<int, string, greater<int> > tem;
 tem rap;
```

tworzą synonim typu **multimap** oraz obiekt **rap** typu **multimap**. Obiekt funkcyjny **greater<int>** sortuje elementy porównując je przy pomocy operatora > (domyślnie kontener **multimap** wykorzystuje operator <). Dzięki temu otrzymujemy porządek malejący w kontenerze. Do wstawiania par do kontenera wykorzystano metodę **insert()** z funkcją składową **value_type():**

```
rap.insert(tem::value_type(25,"monday"));
```

Do obsługi kontenera wygodnie jest mieć zdefiniowany iterator:

tem :: iterator n;

Mapy i multimapy udostępniają funkcje wyszukiwania, opisane są w tabeli 2.13.

Tabela 2.13. Operacje wyszukiwania w **map** i **multimap**

Lp.	Operacja	Opis
1	count(k)	Zwraca liczbę elementów o kluczu **k**
2	find(k)	Zwraca pozycję pierwszego elementu o kluczu **k**
3	lower_bound(k)	Zwraca pierwszą pozycję, na której zostałby wstawiony element o kluczu **k**
4	upper_bound(k)	Zwraca ostatnią pozycję, na której zostałby wstawiony element o kluczu **k**
5	equal_range(k)	Zwraca pierwszą i ostatnią pozycję, na której zostałby wstawiony element o kluczu **k** (zakres elementów)

W pętli **for** korzystając z iteratora **n** wyświetlamy zawartość kontenera:

```
for ( n = rap.begin(); n != rap.end(); ++n)
    cout << n->first << "      " << n->second <<endl;
```

Funkcje wyszukiwania zwracają pozycje iteratora. Gdy nie zostanie znaleziona pozycja iteratora odpowiadająca kluczowi, zostanie zwrócona wartość funkcji **end().** Oznacza to, że nie możemy użyć wprost funkcji np. **find()** do znalezienia elementu o określonej wartości. Zagadnienie to można rozwiązać na kilka sposobów, my wykorzystaliśmy pętlę **for** oraz iterator **n**.
Ten fragment programu ma postać:

```
int ile = 21;
cout <<"dni z temperatura = " << ile << " stopni to:" <<endl;
for ( n = rap.begin(); n != rap.end(); ++n)
    if (n->first == ile) cout << '\t' <<n->second << endl;
```

Podobnie zrealizowane były inne wyszukiwania:

```
ile = 20;
```

```
cout <<"dni z temperatura = " << ile << " stopni bylo ";
cout << rap.count(20) << endl;
ile = 19;
n = rap.find(ile);
cout <<"dzien z temperatura = "<< ile <<" stopni to: "<< n->second;
```

Wbrew pozorom multimapy mają szerokie zastosowania. Typowym przykładem użycia multimap są wszelkiego typu słowniki. W kolejnym programie demonstrujemy wykorzystanie multimapy, jako słownika wyrazów obcych.

Wydruk 2.19. multimapa, słownik, **lower_bound(), upper_bound()**

```
#include <iostream>
#include<iomanip>
#include <string>
#include <map>
using namespace std;

int main()
{
 typedef multimap<string, string> wyrazy_obce;
 wyrazy_obce sl;
 sl.insert(make_pair("faramuszka", "polewka"));
 sl.insert(make_pair("faramuszka", "drobnostka"));
 sl.insert(make_pair("faramuszka", "bagatela"));
 sl.insert(make_pair("granat", "owoc azjatycki"));
 sl.insert(make_pair("granat", "kolor ciemnoniebieski"));
 sl.insert(make_pair("granat", "pocisk"));
 sl.insert(make_pair("szrapnel", "pocisk"));
 sl.insert(make_pair("granat", "rodzaj mineralu"));
 sl.insert(make_pair("bokser", "sportowiec"));
 sl.insert(make_pair("bokser", "rasa psa"));
 sl.insert(make_pair("bokser", "typ silnika"));
 sl.insert(make_pair("bufon", "lichy komik"));
 sl.insert(make_pair("bufon", "zarozumialec"));
 wyrazy_obce:: iterator n;
 cout.setf(ios::left,ios::adjustfield);
 cout << "   " << setw(15) << "wyraz obcy" << "    znaczenie"<<endl;
 cout << endl;
 for( n = sl.begin(); n != sl.end(); ++n)
  cout <<"   " << setw(15) << n->first << n->second << endl;
  string w("bokser");
 cout << endl <<w << " znaczy:" <<endl;
 for ( n = sl.lower_bound(w); n != sl.upper_bound(w); ++n)
  cout <<"   " << setw(15) << n->second << endl;
```

```
  w = "pocisk";
  cout << endl <<w << " to:" <<endl;
  for ( n = sl.begin(); n != sl.end(); ++n)
    if (n->second == w) cout <<"   " << setw(15) << n->first << endl;

  cin.get();
  return 0;
}
```

Pokazany program wyprodukuje następujący komunikat:

```
wyraz obcy          znaczenie
  bokser            sportowiec
  bokser            rasa psa
  bokser            typ silnika
  bufon             lichy komik
  bufon             zarozumialec
  faramuszka        polewka
  faramuszka        drobnostka
  faramuszka        bagatela
  granat            owoc azjatycki
  granat            kolor ciemnoniebieski
  granat            pocisk
  granat            rodzaj mineralu
  szrapnel          pocisk

bokser znaczy:
  sportowiec
  rasa psa
  typ silnika

pocisk to:
  granat
  szrapnel
```

W programie definiujemy typ **multimap**, typem klucza jest **string**, typem skojarzonej wartości jest także **string**, elementy są uporządkowane rosnąco:

```
typedef multimap<string, string> wyrazy_obce;
```

Dalej tworzony jest pusty słownik:

```
wyrazy_obce sl;
```

Do kontenera (naszego słownika) wstawimy w przypadkowej kolejności elementy, korzystając z metody **insert(),** tworząc pary korzystamy z funkcji **make_pair():**

```
sl.insert(make_pair("faramuszka", "polewka"));
```

Mamy przygotowany słownik. Aby wydrukować słownik wykorzystamy iterator:

```
wyrazy_obce:: iterator n;
```

Wydruk elementów obsługuje pętla **for**, korzystamy z manipulatorów formatowania, wydrukowane będą wszystkie elementy słownika:

```
cout.setf(ios::left, ios::adjustfield);
cout << "   " << setw(15) << "wyraz obcy" << "    znaczenie"<<endl;
cout << endl;
for( n = sl.begin(); n != sl.end(); ++n)
  cout <<"   " << setw(15) << n->first << n->second << endl;
```

Korzystanie ze słownika jest stosunkowo proste. Aby wypisać wszystkie wartości dla klucza „bokser” wykorzystamy funkcje **lower_bound(klucz)** i **upper_bound(klucz):**

```
string w("bokser");
cout << endl << w << " znaczy:" <<endl;
for ( n = sl.lower_bound(w); n != sl.upper_bound(w); ++n)
  cout <<"   " << setw(15) << n->second << endl;
```

Wypisywanie wszystkich kluczy dla ustalonej wartość (w naszym przypadku „pocisk”) jest trochę trudniejsze, musimy skorzystać z warunku **if**:

```
w = "pocisk";
cout << endl <<w << " to:" <<endl;
for ( n = sl.begin(); n != sl.end(); ++n)
  if (n->second == w) cout <<"   " << setw(15) << n->first << endl;
```

ROZDZIAŁ 3

KONTENERY I KLASY SPECJALNE

3.1. Wstęp 80
3.2. Adaptator kontenerów stack 80
3.3. Adaptator kontenerów queue 84
3.4. Adaptator kontenerów priority_queue 86
3.5. Klasa bitset 89
3.6. Klasa vector<bool> 92
3.7. Klasa complex 94

3.1. Wstęp

STL zawiera kontenery, klasy i algorytmy ogólnego przeznaczenia stosowane do różnorodnych operacji. Biblioteka zawiera także kontenery, klasy i algorytmy służące do wykonywania specjalistycznych operacji. Istnieje na przykład kontener **bitset** służący do obsługiwania operacji wykonywanych na bitach czy klasa **complex**, służąca do obsługi liczb zespolonych. Biblioteka standardowa zawiera także kontenery zbudowane z podstawowych klas kontenerowych. Są to tzw. adaptatory kontenerów, w tej grupie mamy trzy rodzaje kontenerów: stosy, kolejki i kolejki priorytetowe. Zwracamy uwagę, że w języku polskim nazewnictwo elementów STL nie jest do końca ustalone. W literaturze polskiej angielski termin „*container adaptors*" ma następujące tłumaczenia:

- łącznik zasobników (H.Deitel, P.Deitel)
- kontenery adaptacyjne (J.Liberty, S.Rao, L.Jones)
- adapter (N.Solter, S.Kleper)
- adaptatory kontenerów (N. Josuttis)

W naszym podręczniku będzie stosowali termin „adaptatory kontenerów". Wymienione trzy adaptatory kontenerów nie są zaliczane do kontenerów pierwszej kategorii, ponieważ nie dostarczają rzeczywistej struktury danych, w której elementy mogą być obsługiwane i nie obsługują iteratorów. Mamy trzy typy takich adaptatorów:

- stos (**stack**)
- kolejka (**queue**)
- kolejka priorytetowa (**priority_queue**)

Zgodnie z definicją podana przez A.Stepanowa: „*Adaptors are template classes that provide interface mapping*", adaptatory są klasami, które w oparciu o inne klasy kontenerowe pozwalają tworzyć konstrukcje o nowych możliwościach. **Adaptator kontenerów stack**

Stos (ang. *stack*) jest liniową strukturą danych. Umożliwia wstawianie danych i usuwanie ich z jednego końca struktury danych. Stos nazywany jest strukturą LIFO (ang. *last in, first out*). Stos zdefiniowany jest operacjami, które zmieniają lub sprawdzają jego stan:

- opróżnienie stosu
- sprawdzenie czy stos jest pusty
- sprawdzenie czy stos jest zapełniony
- umieszczenie elementu na stosie (operacja **push**)
- zdjęcie elementu ze stosu (operacja **pop**)

Klasa **stack** może być implementowana z dowolnym kontenerem

sekwencyjnym: **vector, list** i **deque**. Klasa **stack** jest klasą wzorcową pozwalająca na wstawianie i usuwanie elementów na górze stosu. Nie mamy żadnych możliwości operowania elementami wewnątrz stosu. W programie, aby utworzyć strukturę stos należy włączyć plik nagłówkowy <stack>.

Należy pamiętać, że w implementacji klasy **stack** parametr **deque** jest parametrem domyślnym, co oznacza, że tym parametrem może być także klasa **vector** lub **list**. Deklaracja klasy **stack** ma postać:

```
namespace std {
template <class T, class Container = deque<T> >   class stack;
}
```

Do dyspozycji mamy następujące metody:

push() – umieszcza element na stosie
top() – zwraca kolejny element stosu
pop() – usuwa element ze stosu
size() – zwraca bieżącą liczbę elementów
empty() – określa czy stos jest pusty
comparison() – porównuje dwa stosy

W programie demonstrujemy wykorzystanie klasy stack<>. Pokazano wykorzystanie metod klasy.

Wydruk 3.1. Użycia adaptatora stack<>, domyślny kontener **deque**

```
#include<iostream>
#include<stack>
#include <conio.h>
using namespace std;
int main()
{ stack<int> st;
 st.push(11);
 st.push(12);
 st.push(-13);
 st.push(-14);
 cout << "na stosie mamy " << st.size() << " elementy"<<endl;
 cout << "usuwamy element " << st.top() << endl;
 st.pop();
 cout << "na stosie mamy " << st.size()<< " elementy"<<endl;
 st.pop();
 st.empty() ? cout << "stos pusty " : cout << "stos nie pusty";
 cout << "\nusuwamy pozostale elementy";
 while(st.size() != 0)
   st.pop();
 cout << "\n teraz ";
```

```
 st.empty() ? cout <<" stos pusty " : cout << "stos nie pusty";
getche();
return o;
}
```

W wyniku działania programu otrzymujemy komunikat:

```
na stosie mamy 4 elementy
usuwamy element -14
na stosie mamy 3 elementy
stos nie pusty
usuwamy pozostale elementy
teraz stos pusty
```

W linii:

```
stack<int> st;
```

inicjujemy stos liczb całkowitych (typ **int**). Domyślnym kontenerem jest kontener **deque**. W instrukcji:

```
st.push(11);
```

metoda klasy **stack** wykorzystuje metodę **push()** do umieszczenia liczby całkowitej 11 na wierzchołku stosu. Ta metoda jest implementowana przez wywołanie metody **push_back()** z podległego kontenera, w naszym przypadku jest to **deque**. Kolejne wywołanie metody **push()** powoduje umieszczenie nowej wartości na wierzchołku stosu. Po wykonaniu czterech operacji **push()** kolejność wprowadzonych na stos liczb jest następująca: -14, -13, 12, 11.
W instrukcjach:

```
st.pop();
cout << "na stosie mamy " << st.size()<< " elementy"<<endl;
```

możemy, korzystając z metody **pop()** usunąć element z wierzchołka stosu, a za pomocą metody **size()** sprawdzić ile elementów umieszczono na stosie.
Instrukcja:

```
st.top()
```

korzystając z metody **top()** zwraca wartość elementu umieszczonego na wierzchołku stosu.
W instrukcji:

```
st.empty() ? cout <<" stos pusty " : cout << "stos nie pusty";
```

korzystając z metody **empty()** sprawdzamy czy stos jest pusty.
Możemy wykorzystać inny niż domyślny kontener do tworzenia stosu. Kolejny przykład pokazuje tworzenie stosu w kontenerze **vector**.

Wydruk 3.2. Przykład użycia adaptatora stack<>, kontener **vector**

```
#include<iostream>
#include<stack>
#include<vector>
#include<string>
#include <conio.h>
using namespace std;

int main()
{ stack <string, vector<string> > st;
  st.push("Wacek");
  st.push("Robert");
  st.push("Ewa");
  st.push("Zofia");
  cout << " na stosie pierwsza jest " << st.top();
  int n = st.size();
  cout << "\n usuwamy imiona ze stosu" ;
  for (int i = 0; i < n-1; i++) st.pop();
  cout << "\n na stosie zostal " << st.top() << endl;

 getche();
 return 0;
}
```

Wynikiem działania programu jest komunikat:

```
na stosie pierwsza jest Zofia
usuwamy imiona ze stosu
na stosie zostal Wacek
```

W liniach:

```
stack <string, vector<string> > st;
st.push("Wacek");
```

tworzymy stos, który używa kontenera **vector, jako** podległej struktury danych, do umieszczenie elementów klasy **string** (musimy włączyć pliki
< vector> oraz <string>). Metoda **push()** służy do umieszczania elementów na stosie.

3.3. Adaptator kontenerów queue

Kolejka (ang. *queue*) umożliwia wstawianie na końcu i usuwanie z początku podległych struktur danych. Jest to po prostu lista oczekujących. W odróżnieniu od stosu kolejka jest strukturą, która musi wykonywać operacje na początku i na końcu struktury danych. Kolejka nazywana jest także strukturą FIFO (ang. *first in, first out*).

Deklaracja klasy **queue** ma postać:

```
namespace std {
template <class T, class Container = deque<T> >   class queue;
}
```

Domyślnym kontenerem jest kontener **deque,** ale można także wykorzystać kontener **list.**

Do dyspozycji mamy następujące metody:

push() – umieszcza element w kolejce
front() – zwraca element pierwszy kolejki
back() – zwraca ostatni element z kolejki
pop() – usuwa element z kolejki
size() – zwraca bieżącą liczbę elementów
empty() – określa czy kolejka jest pusta
comparison() – porównuje dwie kolejki

Klasa **queue** to klasa wzorcowa, która do działania wymaga dołączenia pliku nagłówkowego <queue>. Przykład użycia klasy **queue** pokazano na listingu 3.3.

Wydruk 3.3. Przykład użycia adaptatora queue<>, kontener **list**

```
#include<iostream>
#include<queue>
#include<list>
#include <conio.h>

using namespace std;

int main()
{ queue<double, list<double> > st;
  st.push(11.0);
  st.push(12.0);
  st.push(-13.0);
```

```
 st.push(-14.0);
 cout << "w kolejce mamy " << st.size() << " elementy"<<endl;
 cout << "pierwszy element to " << st.front() << endl;
 cout << "ostatni element to " << st.back() << endl;
 cout << "teraz usuwamy 1 element" <<endl;
 st.pop();
 cout << "teraz w kolejce mamy " << st.size() << " elementy"<<endl;
 cout << "pierwszy element to " << st.front() << endl;
 cout << "ostatni element to " << st.back() << endl;
 st.empty() ? cout << "kolejka pust " : cout << "kolejka nie pusta";
 cout << "\nusuwamy pozostale elementy";
 while(st.size() != 0)
   st.pop();
 cout << "\nteraz ";
 st.empty() ? cout <<" kolejka pusta " : cout << "kolejka nie pusta";
getche();
return 0;
}
```

Wynikiem działania programu jest komunikat:

```
w kolejce mamy 4 elementy
pierwszy element to 11
ostatni element to -14
teraz usuwamy 1 element
teraz w kolejce mamy 3 elementy
pierwszy element to 12
ostatni element to -14
kolejka nie pusta
usuwamy pozostale elementy
teraz kolejka pusta
```

W liniach:

```
queue<double, list<double> > st;
st.push(11.0);
```

ustanowiono obiekt **queue** o nazwie **st**, w którym określono, że kontenerem wykorzystanym przez **queue** będzie kontener **list.** Przy pomocy metody **push()** umieszczamy liczby zmiennoprzecinkowe (typu **double**) w kolejce. Pozostałe metody działają bardzo podobnie jak pokazano w przykładzie z klasą **stack**. Należy pamiętać, że klasa **queue** nie może korzystać z kontenera **vector.**

Kolejny przykład ilustruje tworzenie adaptatora **queue**, wykorzystującego kontener **deque** do obsługi danych typu **string**.

Wydruk 3.4. adaptator queue<>, kontener **deque**, dane typu **string**

```
#include<iostream>
#include<queue>
#include <string>
#include <conio.h>
using namespace std;
int main()
{ queue <string> st;
  st.push("Pan Kowalski");
  st.push("Pan Zielinski");
  st.push("Pani Kwiatek");
  st.push("Pani Solska");
  cout << "w kolejce mamy " << st.size() << " osoby"<<endl;
  cout << "dostal dokumenty " << st.front() << endl;
  st.pop();
  cout << "dostal dokumenty " << st.front() << endl;
  cout << "ostatni klient to " << st.back() << endl;
 getche();
 return 0;
}
```

Wynikiem działania programu jest komunikat:

```
w kolejce mamy 4 osoby
dostal dokumenty Pan Kowalski
dostal dokumenty Pan Zielinski
ostatni klient to Pani Solska
```

W liniach:

```
queue <string> st;
st.push("Pan Kowalski");
```

tworzymy kolejkę, domyślnym kontenerem jest **deque,** przy pomocy metody **push()** umieszczamy elementy w kolejce. Metoda **back()** zwraca ostatni element kolejki.

3.4. Adaptator kontenerów priority_queue

Kolejka priorytetowa (ang. *priority queue*) różni się od zwykłej kolejki możliwością obsłużenia, jako pierwszego elementu o największej wartości lub elementu uznanego za największy przez predykat dwuargumentowy. Klasa **priority_queue** jest implementowana ze strukturami danych **vector** i **deque**, domyślnie stosowany jest kontener **vector**. W momencie dodawania elementów do kolejki priorytetowej są one sortowane w porządku priorytetowym, to znaczy element o najwyższym priorytecie będzie tak ustawiony, aby był

pierwszym usuniętym. Zwykle sortowanie realizowane jest techniką sortowania stosu (ang. *heapsort*), domyślnie dane są uszeregowane od największej wartości do najmniejszej. Porównywanie domyślne elementów jest przeprowadzane przy pomocy obiektu funkcyjnego less<T>.

Aby utworzyć kolejkę priorytetową należy do programu dołączyć plik nagłówkowy <queue> (klasa **priority_queue** jest zdefiniowana w tym samym pliku, co klasa **queue**). Deklaracja klasy **priority_queue** ma postać:

```
namespace std {
template < class T,
          class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
          class priority_queue;
}
```

Do dyspozycji mamy następujące metody:

push() – umieszcza element w kolejce
top() – zwraca element pierwszy kolejki
pop() – usuwa element z kolejki
size() – zwraca bieżącą liczbę elementów
empty() –określa czy kolejka jest pusta

Kolejny przykład ilustruje tworzenie kolejki priorytetowej do obsługi danych typu **string.** Pokazane są podstawowe operacje.

Wydruk 3.5. adaptator priority_queue<>, dane typu **string**

```
#include<iostream>
#include<queue>
#include <string>
#include <conio.h>
using namespace std;
int main()
{ priority_queue <string> st;
  st.push("Kowalski");
  st.push("Zielinski");
  st.push("Kwiatek");
  st.push("Solska");
  st.push("Biernacki");
  cout << "w kolejce mamy " << st.size() << " osob"<<endl;
  cout << "obywatel na poczatku: " << st.top() << endl;
  cout <<"pierwszy klient zalatwiony";
  st.pop();
  cout << "\nobywatel na poczatku: " << st.top() << endl;
 getche();
 return 0;
}
```

W wyniku działania programu mamy następujący komunikat:

```
w kolejce mamy 5 osob
obywatel na początku: Zielinski
pierwszy klient załatwiony
obywatel na początku: Solska
```

W liniach:

```
priority_queue <string> st;
st.push("Kowalski");
```

tworzymy obiekt **priority_queue** o nazwie **st** i przy pomocy metody **push()** wstawiamy zmienne łańcuchowe. W obiekcie elementy są posegregowane, od wartości największej do najmniejszej. W naszym przypadku zmienna „**Zielinski**" jest na czele kolejki, ponieważ w kodzie ASCII litera „**Z**" ma przyporządkowaną wartość (dziesiętną) 90 (litera „**A**" ma kod 65). Instrukcja:

```
st.pop();
```

usuwa pierwszy element z kolejki priorytetowej, teraz zmienna „**Solska**" znajduje się na czele kolejki. Zauważmy, że wstawianie elementów do kolejki priorytetowej odbywa się w kolejności przypadkowej, wprowadzane dane są sortowane automatycznie. Jak już wspomniano, domyślny predykat można zmienić na dowolny inny. W kolejnym przykładzie zastosujemy obiekt funkcyjny greater<int>.

Wydruk 3.6. adaptator priority_queue<>, obiekt funkcyjny

```
#include<iostream>
#include<queue>
#include <conio.h>
using namespace std;
int main()
{ vector<int> v;
 typedef priority_queue <int, vector<int>,   less<int> > pless;
 typedef priority_queue <int, vector<int>, greater<int> > pgreater;
  v.push_back(3);
  v.push_back(7);
  v.push_back(1);
  v.push_back(5);
  pless v1(v.begin(),v.end());
  cout << "element na poczatku (less): " << v1.top() << endl;
  pgreater v2(v.begin(),v.end());
  cout << "element na poczatku (greater): " << v2.top() << endl;
 getche();
 return 0;
}
```

Wynikiem działania programu jest komunikat:

```
element na początku (less): 7
element na początku (greater): 1
```

W liniach:

```
typedef priority_queue <int, vector<int>,    less<int> > pless;
typedef priority_queue <int, vector<int>, greater<int> > pgreater;
```

używamy **typedef** do stworzenia nowego zestawu kilku liczb całkowitych umieszczonych w kontenerze **priority_queue,** uporządkowanych w porządku rosnącym (**pless**) i malejącym (**pgreater**).
Przy pomocy metody **push_back():**

```
v.push_back(3);
```

umieszczamy elementy w wektorze pomocniczym **v**. Elementami umieszczonymi w wektorze **v** inicjalizujemy kontenery **priority_queue**: **v1** i **v2**. W pierwszym kontenerze mamy porządek rosnący (predykat less<int>), w drugim kontenerze **v2** mamy porządek malejący (predykat greater<int>).

3.5. Klasa bitset

W języku C/C++ mamy możliwość manipulowani bitami w bajtach. Mamy do dyspozycji szereg operatorów bitowych, najczęściej tablicę bitów tworzymy z wykorzystaniem zmiennych typu **int** lub **long**. W STL dysponujemy odrębną klasą o nazwie **bitset** wyspecjalizowaną w obsłudze bitów. Kontenery **bitset** są bardzo przydatne w programowaniu systemów wbudowanych. Kontener **bitset** nie jest kontenerem pierwszego rodzaju: ma stałą wielkość, nie obsługuje iteratorów. Formalnie **bitset** jest sekwencją bitów o stałej długości. Możemy sami ustalać wielkość kontenera. Obiekt **bitset** tworzony jest domyślnie z zer i jedynek. Gdy bit ma wartość 1 to mówimy, że jest „ustawiony”, w przeciwnym wypadku mówimy, że nie jest ustawiony albo jest wyzerowany. W celu skorzystania z klasy **bitset** należy dołączyć do programu plik nagłówkowy <bitset>. Klasa **bitset** zdefiniowana jest w postaci klasy wzorcowej:

```
namespace std {
        template <size_t Bits>
        class bitset;
}
```

W klasie mamy następujące metody niezmieniające wartości kontenera bitset:

size() - zwraca liczbę bitów
count() - zwraca liczbę ustawionych bitów
any() - określa czy został ustawiony jakikolwiek bit
none() - określa czy nie został ustawiony jakikolwiek bit
test(p) - określa czy bit na pozycji **p** jest ustawiony

W klasie mamy metody zmieniające wartości kontenera **bitset**:

set() - ustawia bity
reset() - zeruje bity (albo wszystkie, albo na podanej pozycji)
flip() - odwraca bity
to_ulong() - zwraca wartość całkowitą reprezentowaną przez bity
to_string() - zwraca łańcuch znakowy

W klasie są również zdefiniowane różnego typu operatory:

<< - w strumieniu wyjściowym wstawia sekwencje bitową
>> - wstawia ciąg bitowy do obiektu **bitset**
& - bitowa operacja AND
| - bitowa operacja OR
^ - bitowa operacja XOR
~ - bitowa operacja NOT
>>= - bitowe przesuniecie w prawo
<<= - bitowe przesuniecie w lewo
[n] - umożliwia dostęp do bitu na pozycji **n**

Prostą obsługę bitów przy pomocy kontenera **bitset** ilustruje kolejny program.

Wydruk 3.7. Kontener bitset, metody **test()** i **set()**

```
#include<iostream>
#include<bitset>
#include <conio.h>
using namespace std;

int main()
{ const int r = 8;
  bitset < r > v;
  cout << "podaj sekwencje bitow: ";
  cin >> v;
  if (v.test(3))
     cout <<"bit 3 ustawiony\n";
  else
     cout <<"bit 3 nie ustawiony\n";
  cout <<  v << endl;
```

```
 cout << "teraz bit trzeci ustawiamy\n";
 v.set(3);
 cout << v << endl;

getche();
return 0;
}
```

Wynik działania tego programu jest następujący:

```
podaj sekwencje bitow: 11110000
bit 3 nie ustawiony
11110000
teraz bit trzeci ustawiony
11111000
```

Należy pamiętać o przyjętej konwencji numerowania bitów w sekwencji, bity numerujemy od strony prawej do lewej, skrajny prawy bit ma pozycję zero, w naszym przypadku skrajny bit z lewej strony ma pozycję 7.
W liniach:

```
const int r = 8;
bitset < r > v;
```

tworzymy obiekt **bitset** do przechowywania **r** bitów (w naszym przypadku jest to wartość 8). Dzięki instrukcji:

```
cin >> v;
```

wprowadzamy z klawiatury 8 wartości bitów do obiektu **v**.

Korzystając z metody **test()** sprawdzamy, czy bit na wyspecyfikowanej pozycji jest ustawiony (tzn. czy ma wartość 1) a przy pomocy metody **set()** ustawiamy bit na wyspecyfikowanej pozycji. Kolejny przykład ilustruje wykorzystanie operatora bitowego **AND** oraz wykorzystanie metody **to_ulong()** do konwersji sekwencji binarnej na wartość całkowitą.

Wydruk 3.8. Kontener bitset, metoda **to_ulong()** i operator &

```
#include<iostream>
#include<bitset>
#include <conio.h>
using namespace std;

int main()
{ const int r = 10;
 string s1 = "0011001100";
```

```
 string s2 = "0000111100";
 bitset <r> v1(s1), v2(s2);
 bitset <r> v3(v1 & v2);
 cout << "v1: " << v1 << endl;
 cout << "v2: " << v2 << endl;
 cout <<"wynik operacji v1 & v2 : \n";
 cout << "v3: " << v3 <<endl;
 cout << "wartosc liczbowa v3: " << v3.to_ulong()<< endl;
getche();
return 0;
}
```

Wynik działania programu jest następujący:

```
v1: 0011001100
v2: 0000111100
wynik operacji v1 & v2 :
v3: 0000001100
wartość liczbowa v3: 12
```

W liniach:

```
string s1 = "0011001100";
string s2 = "0000111100";
bitset < r > v1(s1), v2(s2);
```

sekwencje bitów zapisane w łańcuchach **s1** i **s2** umieszczane są obiektach **bitset v1** i **v2**, które mogą pomieścić 10 bitów.
Na sekwencji bitów zapisanych w **v1** i **v2** wykonujemy operacje bitowego **AND** a wynik zapisany jest w obiekcie **v3**:

```
bitset <r> v3(v1 & v2);
```

Korzystając z metody **to_ulong()** przekształcamy sekwencje bitów zapisanych w **v3** w wartość liczbową (system dziesiętny):

```
cout << "wartosc liczbowa v3: " << v3.to_ulong()<< endl;
```

3.6. Klasa vector<bool>

Klasa **bitset** nie ma możliwości dynamicznej zmiany wielkości obiektu. Ciekawym rozwiązaniem jest umieszczona w STL specjalizacja klasy **vector** przeznaczona do przechowywania danych typu **bool.** Specjalizacja vector<bool> ma możliwość dynamicznej zmiany wielkości obiektu (tak jak każdy obiekt klasy **vector**). Należy jednak pamiętać, że tego typu specjalizacja ma ograniczenia, kontener vector<bool> nie jest kontenerem pierwszego

rodzaju. Specjalizacja klasy vector<bool> dostarcza kilku nowych operacji. Te dodatkowe operacje to:

c.flip() - neguje wszystkie elementy kontenera
c[n].flip() - neguje element o indeksie **n**
c[n] = v - przypisuje wartość v do elementu o indeksie **n**
c[n] = c[m] - przypisuje wartość elementu o indeksie **n** elementowi o indeksie **m**

Utworzenie obiektu klasy vector<bool> wymaga dołączenia do programu pliku nagłówkowego <vector>. W programie przedstawiono sposób obsługi obiektu vector<bool>.

Wydruk 3.9. kontener vector<bool>

```
#include<iostream>
#include<vector>
#include <conio.h>
using namespace std;
int main()
{ vector<bool> v(4);
  size_t i;
  v[0] = true;
  v[1] = true;
  v[2] = false;
  v[3] = false;
  cout << "sekwencja nr 1:  ";
  for (i=0; i<v.size(); ++i)
    cout << v[i] << "   ";
  v.push_back(true);
  v.push_back(false);
  cout << "\nsekwencja nr 2:  ";
  for (i=0; i < v.size(); ++i)
    cout << v[i] << "   ";
  vector<bool> v1(6, true);
  cout << "\nsekwencja nr 3:  ";
  for (i=0; i < v1.size(); ++i)
    cout << v1[i] << "   ";
 getche();
 return 0;
}
```

Wynikiem działania tego programu jest komunikat:

```
sekwencja nr 1:  1   1   0   0
sekwencja nr 2:  1   1   0   0   1   0
```

```
sekwencja nr 3:  1   1   1   1   1   1
```

Ustanowienie obiektu vector<bool>, w którym umieścimy cztery elementy ma postać:

```
vector<bool> v(4);
```

Przypisanie wartości poszczególnym elementom wykonujemy przy pomocy operatora indeksu [ ] :

```
v[0] = true;
```

Możemy także umieszczać wartości przy pomocy klasycznej metody kontenera **vector**:

```
v.push_back(true);
```

Przypominamy, że metoda ta umieszcza wartość na końcu kontenera. Jak to jest pokazane na wydruku sekwencji nr 2, kontener zwiększył (dynamicznie) swój rozmiar.
Wydruk zawartości kontenera realizujemy korzystając z indeksu:

```
for (i=0; i<v.size(); ++i)
   cout << v[i] << "   ";
```

W wielu podręcznikach twierdzi się, że klasa **bitset** jest bardziej wydajna niż vector<bool>, wobec tego, gdy nie potrzebujemy obiektów o rozmiarze dynamicznym to powinniśmy używać klasy **bitset.**

3.7. Klasa complex

STL zawiera dość rozbudowaną bibliotekę klas numerycznych. W obliczeniach naukowych oraz inżynierskich bardzo często wygodnie jest posługiwać się liczbami zespolonymi. Biblioteka standardowa posiada klasę **complex**, dzięki której możemy wykonywać obliczenia z wykorzystaniem liczb zespolonych. Standard C99 wprowadził do języka C obsługę liczb zespolonych (plik nagłówkowy <complex.h>). W C++ do obsługi liczb zespolonych musi być włączony plik <complex>. Obsługa liczb zespolonych w nowej bibliotece jest zdecydowanie inna niż w starej.
Liczba zespolona **z** to liczba postaci (jest to tzw. postać algebraiczna):

$$z = a + bi$$

gdzie **i** oznacza tak zwaną jednostkę urojoną, **a** i **b** są liczbami rzeczywistymi. W tym zapisie **a** jest częścią rzeczywistą (składową) liczby zespolonej a **b** jest częścią (składową) urojoną liczby zespolonej. Jednostka zespolona **i** jest

pierwiastkiem kwadratowym z liczby -1, to znaczy spełnia równanie

$$i^2 = -1$$

W przypadku, gdy **b = 0**, liczba z reprezentuje zbiór liczb rzeczywistych. Należy pamiętać, że liczby zespolone, oprócz przedstawienia w postaci algebraicznej, można przedstawiać także w postaci wykładniczej i trygonometrycznej. Liczba zespolona może mieć interpretację geometryczną we współrzędnych prostokątnych (składowe: **a, b**) lub we współrzędnych biegunowych (składowe: r, ϕ). Na liczbach zespolonych określone są operacje arytmetyczne oraz różnorodne funkcje. W pliku nagłówkowym <complex> klasa **complex** jest zdefiniowana następująco:

```
namespace std
 { template  <class T>
   class complex;
 }
```

Konstruktory klasy i operatory przypisania pokazane są w tabeli 3.1.

Tabela 3.1. Konstruktory i operatory przypisania klasy **complex**

wyrażenie	opis
complex z	Tworzy liczbę zespoloną (0, 0i)
complex z(3)	Tworzy liczbę zespoloną (3, 0i)
complex z(3,4)	Tworzy liczbę zespoloną (3, 4i)
complex z1(z2)	Tworzy liczbę zespoloną z1, która jest kopią z2
polar (3)	Współrzędne biegunowe: (3, 0), moduł równa się 3, faza równa jest 0
polar (5, 0.927)	Współrzędne biegunowe: (5, 0.927), moduł równa się 5, kąt fazy równa jest 0.975. W postaci współrzędnych kartezjańskich: (3, 4i)
conj (z)	Liczba zespolona sprzężona z liczbą zespoloną z
z1 = z2	Przypisanie z2 do z1
z1 += z2	Dodanie z1 do z2
z1 -= z2	Odejmowanie z2 od z1
z1 *= z2	Mnożenie z1 przez z2
z1 /= z2	Dzielenie z1 przez z2

W tej klasie mamy trzy specjalizacje ze względu na typ:

- float
- double
- long double

Cztery podstawowe działania arytmetyczne na liczbach zespolonych mają następującą postać:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

$$(a + bi) \times (c + di) = (ac - bd) + (ad + bc)i$$

$$\frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{(a + bi)(c - di)}{\|c + di\|}$$

Liczbę zespoloną można przedstawić w postaci modułowo-argumentowej (ta postać nosi też nazwę postaci trygonometrycznej):

$$z = r(\cos\theta + i\sin\theta)$$

gdzie liczbę **r** nazywamy modułem liczby zespolonej **z**. Kąt θ (mierzony w radianach) nazywamy argumentem liczby **z.**
Reprezentacja wykładnicza liczby zespolonej ma postać:

$$re^{i\theta} = r\cos\theta + (r\sin\theta)i$$

W prezentowanym programie demonstrujemy najprostsze wykorzystanie klasy **complex** do utworzenia liczb zespolonych i wykonania prostych operacji na tych liczbach.

Wydruk 3.10. Liczby zespolone, podstawowe operacje

```
#include <iostream>
#include<complex>
#include<conio.h>
using namespace std;

int main()
{ typedef complex<float> cx;          // typ liczby zespolonej
 cx z1(1.0,2.0);                      //wspolrzedne kartezjanskie
 cx z2(3.0, 4.0);
 cx z3;
  cout << "z1: "<<z1<<endl;
```

```
  cout << "z2: "<<z2<<endl;
  cout << "czesc rzeczywista z2: " << real(z2)<<endl;
 z3 = z1+z2;
  cout << "suma z1 i z2, z3: "<<z3<<endl;
 cx z4(polar(5.0,0.9273));                    //wspolrzedne biegunowe
  cout << "wspol. biegunowe, z4: "<<z4<<endl;
  cout << "modul z z4: "<<abs(z4)<<endl;
  cout << "norma z z4: "<<norm(z4)<<endl;

 getche();
 return 0;
}
```

Wynikiem programu jest następujący komunikat:

```
z1: (1,2)
z2: (3,4)
czesc rzeczywista z2:  3
suma z1 i z2, z3: (4,6)
wspol. biegunowe, z4: (2.99998, 4.00001)
modul z z4: 5
norma z z4: 25
```

W liniach:

```
typedef complex<float> cx;
cx z1(1.0,2.0);
```

deklarujemy i definiujemy liczbę zespolona **z1**. Wybieramy postać algebraiczną, część rzeczywista ma wartość 1.0, część urojona ma wartość 2.0. Liczba zespolona **z1** będzie reprezentowana typem **float**.
Wprowadziliśmy skrót **cx** do oznaczenia konkretnego typu liczby zespolonej.

Równoważną deklarację możemy zapisać następująco:

```
complex<float> z11(1.0, 2.0);
```

Postać trygonometryczną liczby zespolonej (we współrzędnych biegunowych albo polarnych) deklarujemy następująco:

```
cx z4(polar(5.0,0.9273));
```

w tej postaci moduł liczby zespolonej ma wartość 5.0, a argument (zwany też kątem fazy) ma wartość 0.9273 (wartość jest wyrażona w radianach).

Przy pomocy prostej instrukcji:

```
cout << "z1: "<<z1<<endl;
```

wyświetlamy liczbę zespoloną:

```
z1: (1,2)
```

Widzimy, że w nawiasach okrągłych podane są wartości części rzeczywistej i części urojonej. Można spowodować wyświetlenie np. tylko części rzeczywistej:

```
cout << "czesc rzeczywista z2: " << real(z2)<<endl;
```

Funkcja **real()** zwraca wartość rzeczywistą, podobnie funkcja **imag()** zwraca część urojoną. Operację dodawania dwóch liczb zespolonych realizuje przeciążony operator dodawania:

```
z3 = z1+z2;
```

Funkcje **abs()** oraz **norm()** zwracają odpowiednio moduł liczby zespolonej i kwadrat modułu liczby zespolonej:

```
cout << "modul z z4: "<<abs(z4)<<endl;
cout << "norma z z4: "<<norm(z4)<<endl;
```

Do dyspozycji mamy także funkcję **arg(),** która zwraca wartość kąta, gdy liczba zespolona przedstawiona jest we współrzędnych biegunowych.

Liczby zespolone mogą być elementami kontenerów. Poniżej demonstrujemy tworzenie kontenera **vector** przechowującego liczby zespolone. Dane wprowadzamy z klawiatury. Korzystamy z udostępnionych w klasie **complex** operatorów wejścia-wyjścia >> oraz <<. Operator wejściowy odczytuje liczby zespolone w postaci:

```
(część rzeczywista, część urojona)
```

Operator wyjściowy zapisuje liczby zespolone w postaci:

```
(część rzeczywista, część urojona)
```

Operacje we/wy dla liczb zespolonych ilustruje kolejny program.

Wydruk 3.11. Liczby zespolone, kontener **vector,** operacje we/wy

```
#include <iostream>
#include<vector>
#include<complex>
#include<conio.h>
using namespace std;
int main()
{complex<float>  rm;
 vector < complex<float> >  w1;
 vector < complex<float> > :: const_iterator p;
   //format czytania:  (a,b) enter
 cout <<"podaj 3 liczby\n";
 for (int i=0; i<3; ++i)
    {cin >> rm;
     w1.push_back(rm);
    }
 cout <<"w kontenerze mamy : " << endl;
 for (p=w1.begin(); p<w1.end(); ++p)
   cout << *p << "   ";
 cout <<"\n real  imaginary ";
 for (p=w1.begin(); p<w1.end(); ++p)
  { rm = *p;
    cout <<"\n   " << real(rm)<< "   "<< imag(rm)<<"i";
  }
 getche();
 return 0;
}
```

W wyniku działania programu mamy następujący komunikat:

```
podaj 3 liczby
(1,2)
(3,5)
(7,9)
w kontenerze mamy :
(1,2)          (3,5)          (7,9)
  real    imaginary
    1            2i
    3            5i
    7            9i
```

W instrukcjach:

```
complex<float>  rm;
vector < complex<float> >  w1;
vector < complex<float> > :: const_iterator p;
```

deklarujemy liczbę zespoloną **rm,** kontener **w1,** który będzie przechowywał liczby zespolone oraz iterator **p** do obsługi tego kontenera.
Wykorzystujemy metodę **push_back** do wprowadzania elementów, liczby wprowadzamy z klawiatury, musimy pamiętać o nawiasach okrągłych, przy wprowadzaniu danych.
Wprowadzanie danych ma postać:

```
for (int i=0; i<3; ++i)
  {cin >> rm;
   w1.push_back(rm);
  }
```

Przy pomocy operatora wyjściowego << wyświetlamy liczby zespolone zapisane w kontenerze:

```
for (p=w1.begin(); p<w1.end(); ++p)
    cout << *p << "   ";
```

Pokazany przykład potwierdza fakt, że wykorzystanie kontenerów do przechowywania liczb zespolonych nie jest kłopotliwe.

Obsługę liczb zespolonych z wykorzystaniem klasy **complex** i funkcji pokażemy na przykładzie rozwiązywania równań kwadratowych. Gdy mamy równanie kwadratowe postaci:

$$ax^2 + bx + c = 0$$

to pierwiastki równania kwadratowego można obliczyć przy pomocy wzoru

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

W ogólności, w zależności od tak zwanego wyróżnika wielomianu, rozwiązanie będzie w dziedzinie liczb zespolonych. W programie demonstrujemy rozwiązanie równania kwadratowego z wykorzystaniem liczb zespolonych.

Wydruk 3.12. Liczby zespolone, równanie kwadratowe

```
#include <iostream>
#include<complex>
#include<conio.h>
using namespace std;

int main()
{float a=1, b = 2, c=2;
 complex<float> delta = (b*b - 4*a*c);
 complex<float> x1,x2,d,bb;
 d = sqrt(delta);
 bb = complex<float>(b);
 cout << "rozwiazania dla rownania: "<<endl;
 cout << a<<"x**2+"<<b<<"x+ "<<c <<"=0" <<endl;
 x1 = (- bb+= d)/= 2*a;
 x2 = ( -bb-= d)/= 2*a;
 cout <<"\n x1 = " << real(x1)<< " ,  "<< imag(x1)<<"i";
 cout <<"\n x2 = " << real(x2)<< " ,  "<< imag(x2)<<"i";
  getche();
  return 0;
}
```

Wynikiem tego programu jest komunikat:

rozwiązania dla równania:
1x**2 + 2x + 2 = 0

x1 = -1 , 1i
x2 = -1 , -1i

Jeżeli nasze równanie będzie miało postać:

$$x^2 - 4 = 0$$

to program wyświetli następujący komunikat:

rozwiązania dla równania:
1x**2 + 0x + -4 = 0

x1 = 2 , 0i
x2 = -2 , 0i

W praktyce często żądamy wprowadzania danych z klawiatury.

Kolejny pokazany alternatywny program rozwiązujący równania kwadratowe pobiera współczynniki wielomianu z klawiatury, sprawdza, poprawność współczynników i podaje rozwiązanie.

Wydruk 3.13. Liczby zespolone, równanie kwadratowe,

```
#include <iostream>
#include <complex>
#include<conio.h>

using namespace std;

int main()
{
  complex<double> A, B, C, D;
   cout << "A = ";
   cin >> A;
   if (abs(A)<1E-3)
    {cout << "niedobra wartosc A" << endl;
      return 1;
    }
   cout << "B = ";
   cin >> B;
   cout << "C = ";
   cin >> C;
    A *= 2;
   D = B*B-A*C*2.0;
   if (abs(D)<1E-3)
      cout << "x = " << (-B/A).real();
   else
    { cout << "x1 = " << (-B+sqrt(D))/A << endl;
      cout << "x2 = " << (-B-sqrt(D))/A << endl;
    }

   getche();
   return 0;
}
```

W klasie **complex** są zdefiniowane funkcje wykonujące operacje na liczbach zespolonych. Lista tych funkcji pokazana jest w tabeli 3.2

Tabela 3.2. Funkcje zdefiniowane w klasie **complex**, a- liczba rzeczywista, z, z1 – liczba zespolona

funkcja	Opis
exp(z)	Funkcja wykładnicza, e^z
log(z)	Logarytm naturalny z liczby z
log10(z)	Logarytm dziesiętny z liczby z
sqrt(z)	Pierwiastek kwadratowy z liczby z
pow(z,a)	Funkcja potęgowa, z^a
pow(z1,z2)	Funkcja potęgowa, $z1^{z2}$
pow(a,z)	Funkcja potęgowa, a^z
sin(z)	Sinus liczby z
cos(z)	Cosinus liczby z
tan(z)	Tangens liczby z
sinh(z)	Sinus hiperboliczny liczby z
cosh(z)	Cosinus hiperboliczny liczby z
tanh(z)	Tangens hiperboliczny liczby z

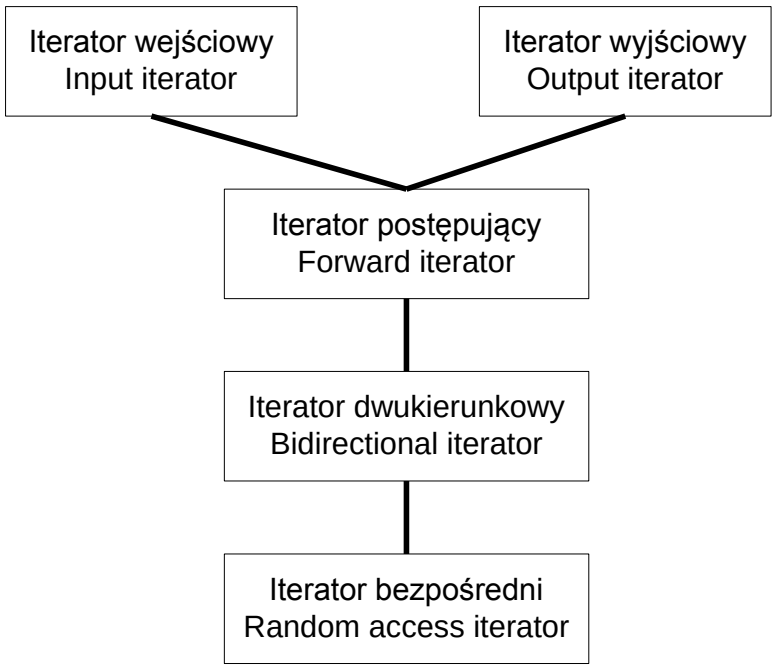
ROZDZIAŁ 4

ITERATORY

4.1. Wstęp 106
4.2. Iteratory wejściowe 112
4.3. Iteratory wyjściowe 113
4.4. Iteratory postępujące 113
4.5. Iteratory dwukierunkowe 114
4.6. Iteratory dostępu swobodnego 115
4.7. Dodatkowe operacje na iteratorach 117

4.1. Wstęp

Zgodnie z definicją Stepanova **iteratory** są uogólnieniem wskaźników języka C++, dzięki którym programista może pracować z różnymi kontenerami zgodnie z jednolitymi zasadami. Dzięki iteratorom można poruszać się po zawartości kontenera w podobny sposób jak dzięki wskaźnikom przemieszczamy się w obrębie tablicy. Ponieważ iteratory zachowują się bardzo podobnie do wskaźników, możliwa jest ich inkrementacja i dekrementacja. Pamiętamy, że w języku C++ mając wskaźnik **wsk** do elementu tablicy, dzięki operacji: **wsk + 1,** otrzymamy wskaźnik do następnego elementu tablicy. Podobnie działają iteratory. Wszystkie kontenery mają zdefiniowane własne iteratory. Iteratory deklaruje się za pomocą typu **iterator** zdefiniowanego w różnych kontenerach. W zasadzie, aby operować iteratorami nie musimy włączać odrębnego pliku nagłówkowego. Jedynie, gdy programista chce wykorzystać specjalne iteratory należy włączyć plik nagłówkowy o nazwie <iterator>. Istnieje pięć różnych typów iteratorów. Hierarchia iteratorów pokazana jest na rysunku 4.1, krótkie charakterystyki pokazane są w tabeli 4.1.



Rys. 4.1. Klasyfikacja iteratorów.

Każdy kontener posiada własne definicje iteratorów, ale wiele typów operacji przeprowadzonych z wykorzystaniem iteratorów daje jednolite wyniki dla wszystkich kontenerów. Na przykład operator dereferencji * pozwala na wykonanie dereferencji iteratora, dzięki czemu możemy wykorzystać wartość elementu wskazywanego przez iterator. Podobnie inkrementacja iteratora ++ zwraca iterator do następnego elementu kontenera.

W tabeli 4.1. pokazano typy iteratorów i ich krótkie charakterystyki. Ze względu na nieustalone polskie tłumaczenia nazw poszczególnych iteratorów, pokazano także nazwy angielskie.

Tablica 4.1. Iteratory STL

Typ iteratora	charakterystyka
Iterator wejściowy *(input iterator)*	Umożliwia odczytanie danych wejściowych (z kontenera, klawiatury, itp.)
Iterator wyjściowy *(output iterator)*	Umożliwia zapis danych (niekonieczny jest odczyt danych)
Iterator postępujący *(forward iterator)*	Jest to ulepszona wersja iteratorów we/wy, umożliwia odczyt i zapis danych. Używa jedynie operatora ++ do poruszania się po kontenerze, co pozwala jedynie na ruch do przodu i zapewnia odczyt tylko po jednym elemencie
Iterator dwukierunkowy *(bidirectional iterator)*	Ma te same funkcje, co iterator postępujący, ale zapewnia przeglądanie kontenera w przód i wstecz
Iterator dostępu swobodnego *(random access iterator)*	Jest to ulepszona wersja iteratora dwukierunkowego, umożliwia odczyt i zapis z dostępem swobodnym (udostępnia operatory do wykonywania arytmetyki iteratorów, co jest analogiczne do arytmetyki wskaźników)

W prostym przykładzie stosowania iteratorów wykorzystamy fakt, że istnieje związek pomiędzy iteratorem i kontenerem **vector**, analogiczny do związku pomiędzy wskaźnikiem i tablicą, jaki istnieje w C++. Jak wiadomo do elementu tablicy możemy odwołać się poprzez indeks albo wskaźnik. Podobnie zachowuje się iterator – umożliwia dostęp do elementu wektora. Program pokazany na wydruku 4.1 pokazuje analogie występujące pomiędzy iteratorami i wskaźnikami.

Wydruk 4.1. Przykład użycia iteratora, kontener **vector ,** typ **int**

```
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;

int main()
{
 const int R = 5;
 int *wsk;                                  //deklaracja wskaznika
 int tab[R] = {1,2,3,4,5};                  //klasyczna tablica
 cout << "wskazniki i tablice" << endl;
 for (wsk=tab; wsk<tab+R; wsk++)
    cout << *wsk << endl;
 vector <int> dane ;                        // kontener vector
 dane.push_back(10);
 dane.push_back(20);
 dane.push_back(30);
 dane.push_back(40);
 dane.push_back(50);
 vector <int> :: iterator p1;               //deklaracja iteratora
 cout << " iterator wektora" << endl;
 for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << endl;
 vector <int> :: reverse_iterator p2;      //deklaracja iteratora odwrotnego
 cout << " iterator odwrotny wektora" << endl;
  for (p2 = dane.rbegin(); p2!= dane.rend(); ++p2)
    cout << *p2 << endl;

  getche();
  return 0;
}
```

Następujący fragment programu:

```
const int R = 5;
 int *wsk;                                  //deklaracja wskaznika
 int tab[R] = {1,2,3,4,5};                  //klasyczna tablica
 cout << "wskazniki i tablice" << endl;
 for (wsk=tab; wsk<tab+R; wsk++)
     cout << *wsk << endl;
```

tworzy zainicjalizowaną tablicę **tab**, wykorzystując arytmetykę wskaźnikową w pętli **for** wyświetla wartości kolejnych elementów tablicy **tab**, korzystając z

operatora dereferencji *.

Kolejno deklarujemy kontener **vector** o nazwie **dane**, w którym przy pomocy metody **push_back()** umieszczamy kolejno wartości całkowite:

```
vector <int> dane ;                    // kontener vector
dane.push_back(10);
……
```

Tak, jak w przypadku wskaźników, iterator także musi być zadeklarowany. Deklaracja iteratora o nazwie **p1** działającego na kontenerze **vector** ma postać:

```
vector <int> :: iterator p1;           //deklaracja iteratora
```

Taka deklaracja jest ważna ponieważ każdy kontener definiuje typ o nazwie iterator, który jest charakterystyczny dla danego kontenera. Przy pomocy pętli **for** wykorzystując operator dereferencji iteratora **p1** wyświetlane są wprowadzone do kontenera wartości:

```
for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << endl;
```

W wyrażeniach pętli **for** wykorzystano metody **begin()** oraz **end().** Metoda **begin()** zwraca iterator (pozycję) do pierwszego elementu umieszczonego w kontenerze **dane**. Inkrementując iterator:

```
++p1
```

uzyskujemy dostęp do kolejnych elementów kontenera **vector**. Należy zwrócić uwagę na wyrażenie warunkowe, wykorzystane do sprawdzenia warunku osiągnięcia końca kontenera:

```
p1!= dane.end();
```

Przy pomocy metody **end()** sprawdzamy, czy iterator znajduje się poza końcem sekwencji elementów umieszczonych w kontenerze. Gdy iterator osiągnie to miejsce, pętla kończy działanie.

W programie 4.1 pokazano także działania metod **rbegin()** i **rend(),** dzięki którym możemy przeglądać kontener w odwrotnym kierunku (wartości elementów kontenera pokazane będą w odwrotnej kolejności, pierwsza pokazana będzie ostatnia wartość kontenera), wykorzystaliśmy tzw. iterator odwrotny:

```
vector <int> :: reverse_iterator p2;   //deklaracja iteratora odwrotnego
 cout << " iterator odwrotny wektora" << endl;
```

```
for (p2 = dane.rbegin(); p2!= dane.rend(); ++p2)
    cout << *p2 << endl;
```

Jak już wspominaliśmy, każdy typ kontenera posiada własny typ iteratora. Stosując na przykład kontener **list,** musimy odpowiednio zadeklarować iterator tego kontenera. To zagadnienie ilustruje kolejny przykład (wydruk 4.2).

Wydruk 4.2. Przykład użycia iteratora, kontener **list,** typ **int**

```
#include <iostream>
#include <list>
#include <conio.h>
using namespace std;
int main()
{list<int> dane ;              // kontener list
 dane.push_front(10);
 dane.push_front(20);
 dane.push_front(30);
 dane.push_front(40);
 dane.push_front(50);
 list <int> :: iterator p1;        //deklaracja iteratora
 cout << " iterator listy" << endl;
 for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << endl;
 getche();
 return 0;
}
```

W programie tworzymy kontener typu **list**, w którym przechowywane będą wartości typu **int**. Wstawianie elementu odbywa się przy pomocy metody **push_front()** obiektu **list**. Kolejność wprowadzanych liczb widoczna jest na wydruku programu 4.2. Po uruchomieniu programu otrzymujemy następujący wynik:

```
 iterator listy
50
40
30
20
10
```

Ta kolejności liczb jest oczywista, metoda **push_front()** wstawia kolejny element ***przed*** jej poprzednikiem. Utworzenie iteratora listy **p1** oraz wydrukowanie elementów listy realizuje następujący fragment programu:

```
list <int> :: iterator p1;          //deklaracja iteratora
cout << " iterator listy" << endl;
```

```
for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << endl;
```

Należy pamiętać, jak to dowcipnie napisali N. Solter i S. Kleper (C++, zaawansowane programowanie) " iteratory są niemal tak bezpieczne jak wskaźniki, czyli są niezwykle niebezpieczne". Przykładem może być niepoprawne użycie metody **end().** Pamiętamy, że iterator zwracany przez **end()** wskazuje na miejsce poza ostatnim elementem kontenera. Dereferencja tego elementu jest nieokreślona, co może spowodować przerwanie działania programu.

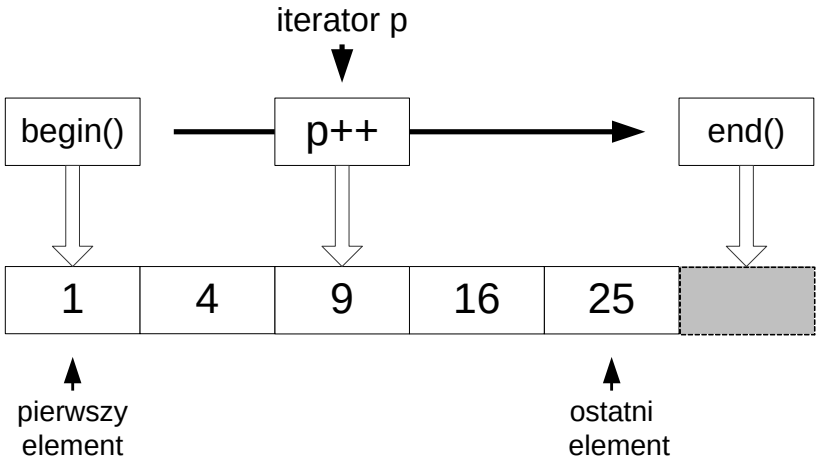
Potrzeba istnienia kilku kategorii iteratorów wynika ze względów praktycznych – każdy kontener ma inne wymagania względem obsługi pozycji elementów w kontenerze. W tabeli 4.2 pokazano typy iteratorów związane z konkretnym kontenerem.

Tabela 4.2. Typy iteratorów

kontener	Typ iteratora
vector	Iterator dostępu swobodnego
list	Iterator dwukierunkowy
deque	Iterator dostępu swobodnego
set	Iterator dwukierunkowy
multiset	Iterator dwukierunkowy
map	Iterator dwukierunkowy
multimap	Iterator dwukierunkowy

Iterator jest obiektem, dzięki któremu możemy wykonywać operacje przemieszczania się od jednego elementu do innego w kontenerze. Z praktycznego punktu widzenia, iterator reprezentuje określoną pozycję elementu w kontenerze. Klasy kontenerowe udostępniają wiele funkcji, dzięki którym można wykonywać operacje związane z przemieszczaniem się iteratora po elementach. Wydaje się, że dwie funkcje są najczęściej wykorzystywane:

- **begin()** – zwraca iterator reprezentujący początek elementów w kontenerze (pozycje pierwszego elementu w sekwencji)
- **end()** – zwraca iterator, który określa pozycję potrzebną do ustalenia pozycji ostatniego elementu w sekwencji, jest to pozycja za ostatnim elementem w kolekcji (rys. 4.2).



Rys. 4.2 Pozycje wskazywane przez metody **begin()** oraz **end()**

4.2. Iteratory wejściowe

Iteratory wejściowe (ang. *input iterator*) to iteratory, które umożliwiają odczytywanie w programie wartości elementów kontenera. Dzięki iteratorom wejściowym możemy wykorzystać dereferencję do pobrania wartości elementu kontenera, bez możliwości zmiany wartości. Do przemieszczania się po elementach kontenera korzystamy najczęściej z operatora ++ . Gdy ustawimy iterator wejściowy na pierwszy element sekwencji, to inkrementacja iteratora pozwoli nam osiągać kolejno wszystkie elementy, aż do osiągnięcia pozycji za ostatnim elementem. Iterator wejściowy jest dość prymitywny, jego czysta postać służy do odczytywania danych z wejścia standardowego (jest to najczęściej klawiatura). Inne typy iteratorów mają znacznie rozszerzoną listę dozwolonych operacji. Liczba dozwolonych operacji wykonywanych na iteratorach wejściowych jest ograniczona, pokazana jest w tabeli 4.3.

Tabela 4.3. Dozwolone operacja iteratorów wejściowych

Operacja	Opis
*p	Dostęp do aktualnego elementu z możliwością odczytu
p->obiekt	Dostęp do składowej aktualnego elementu z możliwością odczytu
++p	Przesunięcie do przodu, zwraca nową pozycję
p++	Przesunięcie do przodu, zwraca dotychczasową pozycję
p1 == p2	Sprawdza, czy dwa iteratory są równe
p1 != p2	Sprawdza, czy dwa iteratory są różne
TYPE(p)	Kopiuje iterator

Operator wejściowy można jedynie zwiększać (nie ma możliwości jego zmniejszenia). Każdy element sekwencji można odczytać tylko raz przy pomocy operatora *. Zaleca się by korzystać postaci przedrostkowej (++p) operatora inkrementacji, ponieważ jest to znacznie szybsze od postacie przyrostkowej (p++). Iteratory wejściowe udostępnia biblioteka ***istream***.

4.3. Iteratory wyjściowe

Są to proste iteratory od obsługi danych wyjściowych. Przy pomocy operatora ++ mamy możliwości poruszania się do przodu przeglądając sekwencje i zapisać każdy z elementów tylko raz za pomocą operatora *. Nie ma możliwości użycia iteratora wyjściowego do dwukrotnej iteracji po tym samym zakresie. Nie ma gwarancji, że przy próbie ponownego zapisania nowej wartości na tej samej pozycji zostanie nadpisana stara wartość.

Operator wyjściowy najczęściej jest używany do zapisu na urządzeniu wyjścia standardowego (najczęściej jest to ekran monitora). Iteratory wyjściowe są udostępniane przez bibliotekę ***ostream***.

Tabela 4.4. Dozwolone operacja iteratorów wyjściowych

operacja	Opis
*p = dane	Zapisuje wartości danych w miejscu wskazanym przez iterator
p->obiekt	Dostęp do aktualnego elementu z możliwością odczytu
++p	Przesunięcie do przodu, zwraca nową pozycję
p++	Przesunięcie do przodu, zwraca dotychczasową pozycję
TYPE(p)	Kopiuje iterator

4.4. Iteratory postępujące

Iteratory postępujące (ang. *forward iterator*) łączą cechy iteratorów wejściowych i wyjściowych. Iteratory tego typu mogą przetwarzać dany element w kontenerze wiele razy. Dzięki iteratorowi postępującemu i wykorzystaniu operatora ++ można przemieszczać się w przód przeglądając sekwencje elementów kontenera.

W tabeli 4.5 pokazano dozwolone operacje iteratorów postępujących.

Tabela 4.5. Dozwolone operacja iteratorów postępujących

Operacja	Opis
*p	Dostęp do aktualnego elementu z możliwością odczytu
p->obiekt	Dostęp do składowej aktualnego elementu z możliwością odczytu
++p	Przesuniecie do przodu, zwraca nowa pozycję
p++	Przesuniecie do przodu, zwraca dotychczasowa pozycję
p1 == p2	Sprawdza, czy dwa iteratory są równe
p1 != p2	Sprawdza, czy dwa iteratory są różne
TYPE()	Tworzy iterator
TYPE(p)	Kopiuje iterator
p1 = p2	Przypisuje iterator

Iterator postępujący umożliwia równocześnie odczyt i zapis danych, należy w takim przypadku wprowadzić deklarację:

```
int * p1       //odczyt i zapis, dane typu int
```

Gdy chcemy mieć możliwość jedynie odczytu danych to musimy zastosować następującą deklarację:

```
const int * p2     // tylko odczyt, dane typu int
```

4.5. Iteratory dwukierunkowe

Iteratory dwukierunkowe (ang. *bidirectional iterator*) są modyfikacją iteratorów postępujących. Iteratory tego typu mogą przetwarzać dany element w kontenerze wiele razy przemieszczając się zarówno do przodu jak i wstecz. Ten typ iteratora udostępniony jest przez kontenery **list, map** i **set**.
W tabeli 4.6 pokazano dozwolone operacje iteratorów dwukierunkowych.

Tabela 4.6. Dozwolone operacja iteratorów dwukierunkowych

Operacja	Opis
*p	Dostęp do aktualnego elementu z możliwością odczytu
p->obiekt	Dostęp do aktualnego elementu z możliwością odczytu
++p	Przesunięcie do przodu, zwraca nową pozycję
p++	Przesunięcie do przodu, zwraca dotychczasową pozycję

--p	Przesunięcie wstecz, zwraca nową pozycję
p--	Przesunięcie wstecz, zwraca aktualną pozycję
p1 == p2	Sprawdza, czy dwa iteratory są równe
p1 != p2	Sprawdza, czy dwa iteratory są różne
TYPE()	Tworzy iterator
TYPE(p)	Kopiuje iterator
p1 = p2	Przypisuje iterator

4.6. Iteratory dostępu swobodnego

Najbardziej uniwersalnym iteratorem jest iterator dostępu swobodnego (ang. *random access iterator*). Zawierają wszystkie operacje zdefiniowane dla iteratorów dwukierunkowych a także mają możliwość realizacji dostępu swobodnego (mogą przesuwać się w sekwencji o dowolną ilość pozycji). Wykorzystują tzw. arytmetykę iteratorów, analogicznie jak w języku C++ wykorzystywana jest arytmetyka wskaźnikowa. W tabeli 4.7 pokazano dodatkowe dozwolone operacje iteratorów dostępu swobodnego.

Tabela 4.7. Dodatkowe operacje iteratorów dostępu swobodnego

Operacja	Opis
p[n]	Umożliwia dostęp do elementu o indeksie **n**
p += n	Przesuwa się o n elementów do przodu (wstecz, gdy **n** jest ujemne)
p -= n	Przesuwa się o **n** elementów wstecz (do przodu, gdy **n** jest ujemne)
p + n	Zwraca iterator elementu oddalonego o **n** pozycji
n + p	Zwraca iterator elementu oddalonego o **n** pozycji
p - n	Zwraca iterator elementu oddalonego o **n** pozycji wstecz
p1 - p2	Zwraca odległość między iteratorami
p1 < p2	Ustala, czy **p1** występuje przed **p2**
p1 > p2	Ustala, czy **p1** występuje po **p2**
p 1<= p2	Ustala, czy **p1** nie występuje po **p2**
p1 >= p2	Ustala, czy **p1** nie występuje przed **p2**

Iteratory dostępu swobodnego udostępniane są przez kontenery **vector** i **deque**. W kolejnym przykładzie (wydruk 4.3) demonstrujemy wykorzystanie iteratora dostępu swobodnego.

Wydruk 4.3. Przykład użycia iteratora, kontener **deque ,** typ **int**

```
#include <iostream>
#include <deque>
#include <conio.h>
```

```
using namespace std;
int main()
{deque<int> dane ;                          // kontener deque
 dane.push_front(60);
 dane.push_front(50);
 dane.push_front(40);
 dane.push_front(30);
 dane.push_front(20);
 dane.push_front(10);
 cout << "dystans = " << dane.end() - dane.begin() << endl;
 deque <int> :: iterator p1;                //deklaracja iteratora
 cout << " iterator, elementy: " << endl;
 for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << "  ";
 cout << endl;
 cout << " operator [ ], elementy: " << endl;
 for (int i =0; i < dane.size(); ++i)
    cout << dane.begin()[i]  << "  ";
 cout << endl;
 cout << " iterator, co drugi element: " << endl;
 for (p1 = dane.begin(); p1< dane.end()-1; p1 +=2)
    cout << *p1 << "  ";
 getche();
 return 0;
}
```

Po uruchomieniu programu mamy następujące wyniki:

```
dystans = 6
 iterator, elementy:
10  20  30  40  50 60
 operator [ ], elementy:
10  20  30  40  50 60
 iterator, co drugi element:
10  30  50
```

W tym przykładzie pokazano klasyczną metodę wykorzystania iteratora, pokazano metodę dostępu do elementów kontenera wykorzystując indeksy, pokazano także metodę operowania wybranymi elementami kontenera.

Przy pomocy metody **push_front()** umieszczamy elementy w kontenerze **deque**:

```
deque<int> dane ;                  // kontener deque
dane.push_front(60);
```

Obliczenie odległości między iteratorem początkowym i końcowym realizowane jest w instrukcji:

```
cout << "dystans = " << dane.end() - dane.begin() << endl;
```

Należy zauważyć, że pokazanych operacji z iteratorami dostępu swobodnego nie można zastosować dla obsługi kontenerów **list, set** i **map**.
Wydruk elementów kontenera realizowany jest przy pomocy iteratora dostępu swobodnego w następujący sposób:

```
for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << "  ";
```

Aby otrzymać ten sam wynik można wykorzystać indeksy do obsługi elementów kontenera:

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

Do wyprowadzenia wartości, co drugiego elementu kontenera wykorzystujemy operator += (i oczywiście arytmetykę iteratorów) w następujący sposób:

```
for (p1 = dane.begin(); p1< dane.end()-1; p1 +=2)
   cout << *p1 << "  ";
```

Należy jednak pamiętać o dużym niebezpieczeństwie, które występuje w tego typu realizacji przeglądania elementów kontenera. Może się, bowiem zdarzyć tak, że kolejne przesunięcie iteratora dostępu swobodnego znajdzie się za pozycją **end()** naszej kolekcji. Spowoduje to niekontrolowane zachowanie się programu i zazwyczaj skończy się jego przerwaniem.

4.7. Dodatkowe operacje na iteratorach

W bibliotece STL są zaimplementowane trzy dodatkowe funkcje operujące na iteratorach:

- advance()
- distance()
- iter_swap()

Użycie tych funkcji wymaga dołączenia pliku <iterator>. Występujący w prototypach funkcji specyfikator **Dist** oznacza typ.
Funkcja **advance(),** której prototyp ma postać:

```
void advance(InputIterator& p, Dist, n)
```

umożliwia przesunięcie iteratora **p** o **n** elementów. Argument **n** może być dodatni lub ujemny.
Funkcja **distance(),** której prototyp ma postać:

Dist distance(InputIterator p1, Input Iterator p2)

zwraca odległość pomiędzy iteratorami **p1** i **p2**.
Kolejną funkcją pomocniczą jest funkcja **iter_swap().** Jej prototyp ma postać:

void iter_swap(ForwardIterator1 p1, ForwardIterator2 p2)

Funkcja **iter_swap()** wykorzystywana jest do zamiany dwóch iteratorów. Iteratory nie muszą być tego samego typu.

W prostym przykładzie pokażemy zastosowanie omówionych trzech funkcji pomocniczych operujących na iteratorach.

Wydruk 4.4. Przykład użycia **advance(),distance()** oraz **iter_swap(),**

```
#include <iostream>
#include <deque>
#include <conio.h>
using namespace std;

int main()
{deque<int> dane ;              // kontener deque
 dane.push_front(60);
 dane.push_front(50);
 dane.push_front(40);
 dane.push_front(30);
 dane.push_front(20);
 dane.push_front(10);
 deque <int> :: iterator p1,p2,p3;        //deklaracja iteratora
 cout << " wszystkie elementy: " << endl;
 for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << "  ";
 cout << "\ndystans = " << dane.end() - dane.begin() << endl;
 p2 = dane.begin() ;
 cout << "pierwszy element = " << *p2 << endl;
 advance(p2, 4);
 cout << " element po przesunieciu o 4 pozycje = " << *p2 << endl;
 p3 = find(dane.begin(), dane.end(), 40);
 cout << "wyszukano element  " << *p3 << endl;
 deque<int>::difference_type dist = 0;
 distance(dane.begin(), p3, dist);
 cout << "odleglosc pomiedzy elementami = " << dist<< endl;
```

```
 cout << " wybrano co drugi element: " << endl;
 for (p1 = dane.begin(); p1< dane.end()-1; p1 +=2)
    cout << *p1 << "  ";
 cout << "\nzamiana 2 elementow"<< endl;
 iter_swap(dane.begin(),dane.begin()+1);
 for (p1 = dane.begin(); p1!= dane.end(); ++p1)
    cout << *p1 << "  ";

  getche();
  return 0;
}
```

Wynik działania programu ma postać:

```
wszystkie elementy:
10  20  30  40  50  60
dystans = 6
pierwszy element = 10
element po przesunieciu o 4 pozycje = 50
wyszukano element 40
odległość pomiedzy elementami = 3
wybrano co drugi element:
10  30  50
zamiana 2 elementow:
20  10  30  40  50  60
```

Działanie funkcji **advance()** pokazane jest w następującym fragmencie programu:

```
p2 = dane.begin() ;
cout << "pierwszy element = " << *p2 << endl;
advance(p2, 4);
cout << " element po przesunieciu o 4 pozycje = " << *p2 << endl;
```

Na początku iterator **p2** wskazuje na pierwszy element sekwencji (funkcja **begin()**), co potwierdza pierwszy wydruk ***p2**. Następnie wywołujemy funkcję **advance()** z parametrami (**p2**, 4). Oznacza to, że żądamy przesunięcia iteratora **p2** o cztery miejsca. Kolejny wydruk ***p** potwierdza, że **p2** wskazuje na piąty element w naszej kolekcji, jego wartość jest równa 50.

Obliczanie odległości pomiędzy iteratorami przy użyciu funkcji **distance()** realizuje następujący fragment programu:

```
p3 = find(dane.begin(), dane.end(), 40);
cout << "wyszukano element  " << *p3 << endl;
deque<int>::difference_type dist = 0;
distance(dane.begin(), p3, dist);
cout << "odleglosc pomiedzy elementami = " << dist<< endl;
```

Funkcja **find()** wyszukuje w naszej kolekcji elementu o wartości 40, zwraca pozycję tego elementu, przypisując ją iteratorowi **p3**. Ta pozycja wykorzystana jest w funkcji **distance()** do wyliczenia odległości pomiędzy tym iteratorem i początkiem kolekcji (funkcja **begin()**). Odległość elementu o wartości 40 od elementu pierwszego kolekcji równa jest 3.
Sposób zamiany dwóch elementów kolekcji pokazany jest w instrukcji:

```
iter_swap(dane.begin(),dane.begin()+1);
```

Funkcja **iter_swap()** pobiera dwa argumenty – iteratory (pozycje) dwóch elementów, które mają się wymienić położeniem. W naszym przykładzie zamieniamy miejscami element pierwszy (**begin()**) oraz drugi (**begin()+1**). W wyniku element drugi staje się elementem pierwszym kolekcji, a element pierwszy staje się drugim.

ROZDZIAŁ 5

ALGORYTMY

5.1. Wstęp 122
5.2. Algorytmy niemodyfikujące 127
5.3. Algorytmy modyfikujące 135
5.4. Algorytmy usuwające 138
5.5. Algorytmy mutujące 144
5.6. Algorytmy sortujące 148
5.7. Algorytmy przeznaczone dla zakresów posortowanych 155
5.8. Algorytmy numeryczne 160

5.1. Wstęp

Według wielu specjalistów najważniejszymi elementami STL są kontenery, iteratory i algorytmy. Cenną zaletą STL jest fakt, że zawiera ona wiele typowych algorytmów. W praktyce programistycznej bardzo często sortujemy dane, wyszukujemy odpowiednie elementy, itp. Wykorzystując algorytmy STL programista oszczędza mnóstwo czasu. Algorytmy STL są odseparowane od kontenerów, wykorzystują iteratory do operowania na elementach kontenera. Dzięki takiej koncepcji algorytmy wykonują swoje działania na wielu kontenerach, bez względu na typ danych. STL zawiera znaczą ilość algorytmów. Doskonały przegląd wszystkich algorytmów STL można znaleźć w podręczniku „C++, biblioteka standardowa, podręcznik programisty" autorstwa N. Josuttisa. W zależności od implementacji STL zawiera od 60 do 70 standardowych algorytmów. Systematyzacja algorytmów STL nie jest prosta. Według A. Stepanova (1996 rok) mamy cztery kategorie algorytmów:

- Algorytmy mutujące (ang. *mutating sequence operations*)
- Algorytmy niemutujące (ang. *non-mutating sequence operations*)
- Algorytmy sortujące i relacyjne (ang. *sorting and related operations*)
- Algorytmy numeryczne (ang. *generalized numeric operations*)

N. Josuttis (2003 rok) wprowadził następującą klasyfikację:

- Algorytmy niemodyfikujące (ang. *nonmodifying algorithms*)
- Algorytmy modyfikujące (ang. *nonmodifying algorithms*)
- Algorytmy usuwające (ang. *removing algorithms*)
- Algorytmy mutujące (ang. *mutating algorithms*)
- Algorytmy sortujące (ang. *sorting algorithms*)
- Algorytmy przeznaczone dla zakresów posortowanych (ang. *sorted range algorithms*)
- Algorytmy numeryczne (ang. *numeric algorithms*)

Należy pamiętać, że użycie algorytmów wymaga dołączenia pliku nagłówkowego z definicjami algorytmów:

```
#include <algorithm>
```

Niektóre algorytmy numeryczne wymagają dołączenia pliku nagłówkowego numerycznego:

```
#include <numeric>
```

Gdy używane będą obiekty funkcyjne (niektóre algorytmy wymagają tego) należy włączyć także odpowiedni plik z definicjami obiektów funkcyjnych i

adaptatorów funkcji:

#include <functional>

Zasadniczo algorytmy STL wykonują operacje na kontenerach wykorzystując iteratory. Kilka algorytmów może działać nie wymagając elementów zapisanych w kontenerach, może działać na wartościach. Przykładem takiego algorytmu jest np. **swap**, który akceptuje dwie wartości i zamienia je miejscami. Podobnie algorytmy **min** oraz **max** także działać mogą bezpośrednio na wartościach. W pokazanym programie demonstrujemy wykorzystanie algorytmu **next_permutation()** do utworzenia wszystkich permutacji liczb wprowadzonego z klawiatury.

Wydruk 5.1. Przykład użycia algorytmu **next_permutation()**

```
#include <iostream>
#include<vector>
#include <algorithm>
#include <conio.h>
using namespace std;

int main()
{vector <int> v;
 vector <int> :: iterator p;
 v.push_back(1);
 v.push_back(2);
 v.push_back(3);
 p = v.begin();
 cout << "wejsciowa sekwencja:" << endl;
 while(p!=v.end())
  {cout << *p << "  ";
   p++ ; }
 cout << endl;
 cout << "permutacje:" << endl;
 while (next_permutation(v.begin(), v.end()))
  { p = v.begin();
    while(p!=v.end())
     { cout << *p << "  ";
       p++ ;}
    cout << endl;
  }
 getche();
 return 0;
}
```

Wynikiem uruchomienia programu jest następujący komunikat:

```
wejsciowa sekwencja:
1 2 3
permutacje:
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

W STL mamy dwa algorytmy tworzące permutację elementów - **next_permutation()** i **prev_permutation().** Jeżeli mamy zbiór liczb całkowitych 1, 2 i 3, to możemy utworzyć sześć permutacji : 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1. Algorytm **next_permutation**, jako argumenty przyjmuje zakres [**pierwszy, ostatni**] i przekształca wprowadzony zbiór pierwotny w kolejną permutację, jeżeli jest to możliwe. Gdy utworzenie nowej permutacji jest możliwe, algorytm zwraca wartość **true.** Gdy utworzenie kolejnej permutacji nie jest możliwe (wyczerpano wszystkie możliwości), algorytm zwraca wartość **false**. W naszym przykładowym programie, następujący fragment tworzy wszystkie dozwolone permutacje:

```
while (next_permutation(v.begin(), v.end()))
 { p = v.begin();
       while(p!=v.end())
            { cout << *p << "  ";
                p++ ;}
   cout << endl;
 }
```

W kolejnym przykładzie demonstrujemy kolejne użyteczne algorytmy wyszukiwania elementu w zbiorze (algorytm **find()** oraz **binary_search()**) i algorytm **sort()** służący do porządkowania elementów w sekwencji.

Wydruk 5.2. Algorytm **find(), sort()** i **binary_search()**

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <conio.h>
using namespace std;
int main()
{ int tab[6] = {1,13, 2, 4, 27,100 };
  vector <int> v(tab, tab+6);
  vector <int> :: iterator p1;
  cout << "zbior wejsciowy : "<< endl;
```

```
    for (p1=v.begin(); p1 != v.end(); ++p1)
        cout << *p1 << "  ";
    cout << endl;
    p1 = find(v.begin(), v.end(), 2);
    cout << "wartosc 2 na pozycji: " << (p1 - v.begin()+1) << endl;
    if (binary_search(v.begin(), v.end(), 27))
        cout << "wartosc 27 jest w zbiorze";
    else
        cout << "wartosci tej nie ma w zbiorze" << endl;
    cout <<endl;
    sort(v.begin(),v.end());
    cout << "zbior posortowany:"<< endl;
    for (p1=v.begin(); p1 != v.end(); ++p1)
        cout << *p1 << "  ";
    cout << endl;
     getche();
     return 0;
}
```

Po uruchomieniu tego programu mamy następujący komunikat:

```
zbior wejściowy:
1  13   2   4   27  100
wartosc 2 na pozycji: 3
wartosc 27 jest w zbiorze
zbior posortowany:
1   2  4  13  27  100
```

W programie zainicjalizowano wektor **v** tablicą **tab**:

```
int tab[6] = {1,13, 2, 4, 27,100 };
vector <int> v(tab, tab+6);
```

przeciążony konstruktor **vector**, korzystając z dwóch iteratorów (pamiętamy, że wskaźniki do tablicy mogą być użyte, jako iteratory) tworzy wektor **v** zainicjowany elementami tablicy **tab** od lokalizacji **tab** do lokalizacji mniejszej niż **tab + 6**. W programie definiujemy iterator **p1**:

```
vector <int> :: iterator p1;
```

Wykorzystamy algorytm **find()** do sprawdzenia czy wyspecyfikowany element jest w zbiorze elementów wektora **v**:

```
p1 = find(v.begin(), v.end(), 2);
cout << "wartosc 2 na pozycji: " << (p1 - v.begin()+1) << endl;
```

W pokazanej instrukcji, algorytm **find(p1,p2,w)** zwraca pozycję pierwszego wystąpienia elementu o wartości **w** z zakresu **p1** i **p2** w kontenerze **v**. Gdy wyspecyfikowana wartość nie będzie znaleziona zostanie zwrócona pozycja **p2** (koniec sekwencji). Do stwierdzenia faktu, czy wyspecyfikowana wartość znajduje się w zbiorze elementów można wykorzystać algorytm **binary_search().** Formalnie ten algorytm wymaga uporządkowanego zbioru (można to osiągnąć korzystając z algorytmu **sort()**), ale w większości implementacji, algorytm **binary_search()** działa poprawnie, tak jak to pokazano w naszym przykładzie. Rekomenduje się wykorzystanie algorytmu **binary_search()**, ponieważ działa bardzo szybko. Jeżeli szukany element jest w zbiorze, to algorytm **binary_search** zwraca wartość typu **bool** (**true**), w przeciwnym przypadku zwróci wartość **false**.

```
if (binary_search(v.begin(), v.end(), 27))
    cout << "wartosc 27 jest w zbiorze";
else
    cout << "wartosci tej nie ma w zbiorze" << endl;
```

Do posortowania elementów w kontenerze wykorzystaliśmy algorytm **sort(p1,p2),** który ustawia elementy w zakresie od **p1** do **p2** (ale z wyłączeniem **p2**) w obiekcie **v** w porządku rosnącym:

```
sort(v.begin(),v.end());
```

Często zachodzi potrzeba losowego ustawienia elementów. W takim przypadku można wykorzystać algorytm numeryczny **random_shuffle().** Działanie tego algorytmu pokazano w kolejnym przykładzie.

Wydruk 5.3. Przykład użycia algorytmu **random_shuffle()**

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>                    //wymagane dla ostream_iterator
#include <conio.h>
using namespace std;
int main()
{ int tab[6] = {1,2,4,13,27,100 };
  vector <int> v(tab, tab+6);
  ostream_iterator <int> output (cout, "  ");
  cout << "elementy wektora: "<< endl;
  copy(v.begin(), v.end(), output);
  random_shuffle(v.begin(), v.end());
  cout << "\nzbior losowy : "<< endl;
  copy(v.begin(), v.end(), output);
   getche();
   return 0;
}
```

Po uruchomieniu programu mamy następujący wynik:

```
elementy wektora:
1 2 4 13 27 100
zbior losowy:
2  100 13  4  1  27
```

W programie wektor **v** inicjalizowany został elementami tablicy **tab**:

```
int tab[6] = {1,2,4,13,27,100 };
vector <int> v(tab, tab+6);
```

W programie wykorzystaliśmy elegancki sposób wyprowadzania danych na ekran monitora:

```
ostream_iterator <int> output (cout, "  ");
cout << "elementy wektora: "<< endl;
copy(v.begin(), v.end(), output);
```

W instrukcji:

```
ostream_iterator <int> output (cout, "  ");
```

zadeklarowany jest iterator strumieniowy wyjściowy o nazwie **output**, dzięki któremu dane typu **int** są wysyłane na wyjście, dane są rozdzielane spacjami.
Instrukcja:

```
copy(v.begin(), v.end(), output);
```

wykorzystuje algorytm **copy(p1,p2)** w wyspecyfikowanym zakresie [**p1, p2**) do wyprowadzania wszystkich elementów wektora **v** do standardowego wyjścia.
W instrukcji:

```
random_shuffle(v.begin(), v.end());
```

algorytm **random_shuffle(p1,p2)** ustawia przypadkowo elementy wektora **v** z zakresu [**p1, p2**).

Algorytmy niemodyfikujące

Algorytmy niemodyfikujące (ang. *nonmodifying algorithms*) realizują operacje na elementach kontenera nie zmieniając ich wartości ani kolejności. Istnieje kilka grup algorytmów niemodyfikujących:

- Algorytmy wyszukujące
- Algorytmy porównujące
- Algorytmy iterujące

Algorytmy niemodyfikujące wraz z ich krótkimi opisami przedstawione są w tabeli 5.1.

Tabela 5.1 Algorytmy niemodyfikujące
(b – początek zakresu, b1-początek drugiego zakresu, e – koniec zakresu, up – predykat unarny, bp – predykat binarny, v - wartość)

Algorytm	działanie
adjacent_find(b,e)	Szuka dwóch sąsiednich, równych elementów z zakresu (b,e)
adjacent_find(b,e,bp)	Szuka dwóch sąsiednich, równych elementów z zakresu (b,e) dla którego **bp** ma wartość **true**
count(b,e,v)	Zlicza elementy o wartości **v**
count_if(b,e,up)	Zlicza elementy dla których **up** ma wartość **true**
equel(b,e, b1)	Sprawdza czy elementy pierwszego zakresu są równe elementom drugiego zakresu od pozycji **b1**
equel(b,e, b1,bp)	Sprawdza czy dla elementów pierwszego zakresu oraz dla elementów drugiego zakresu od pozycji **b1, bp** ma wartość **true**
find(b,e,v)	Zwraca pozycję pierwszego pasującego elementu o wartości **v**
find_if(b,e,up)	Zwraca pozycję pierwszego elementu gdy **up** ma wartość **true**
find_end(b,e,b1,e1)	Zwraca pozycję pierwszego elementu ostatniego podzakresu (b1,e1) występującego w zakresie (b,e)
find_end(b,e,b1,e1,bp)	Zwraca pozycję pierwszego elementu ostatniego podzakresu (b1,e1) występującego w zakresie (b,e), gdy **bp** ma wartość **true**
find_first_of(b,e,b1,e1)	Zwraca pozycję pierwszego elementu z zakresu (b,e) który występującego w zakresie (b1,e1)
find_first_of(b,e,b1,e1,bp)	Zwraca pozycję pierwszego elementu z zakresu (b,e) dla którego wywołanie elementu z zakresu (b1,e1) dla **bp** ma wartość **true**
for_each(b,e,up)	Umożliwia dostęp do każdego elementu kolekcji i wykonanie operacji **up**

lexicographical_ compare(b,e,b1,e1)	Sprawdza, czy elementy z zakresu (b,e) są leksykograficznie mniejsze od elementów z zakresu (b1,e1)
lexicographical_ compare(b,e,b1,e1, bp)	Porównuje leksykograficzne elementy z obu zakresów przy użyciu predykatu **bp**
max_element(b,e)	Wyszukuje i zwraca pozycję elementu o największej wartości
max_element(b,e,bp)	Wyszukuje i zwraca pozycje elementu o największej wartości gdy **bp** ma wartość **true**
min_element(b,e)	Wyszukuje i zwraca pozycje elementu o najmniejszej wartości
min_element(b,e,bp)	Wyszukuje i zwraca pozycje elementu o najmniejszej wartości gdy **bp** ma wartość **true**
mismatch(b,e,b1)	Wyszukiwanie pierwszych dwóch odpowiadających sobie elementów, które nie są równe
mismatch(b,e,b1,bp)	Wyszukiwanie pierwszych dwóch odpowiadających sobie elementów, dla których **bp** ma wartość **false**
search(b,e,b1,e1)	Zwraca pozycję pierwszego elementu z pierwszego zbioru od którego zaczyna sie zgodność elementów z drugiego zbioru
search(b,e,b1,e1,bp)	Zwraca pozycję pierwszego elementu z pierwszego zbioru od którego zaczyna sie zgodność elementów z drugiego zbioru oraz **bp** ma wartość **true**
searach_n(b,e,n,v)	Wyszukuje pierwszych **n** elementów o wartości **v**
searach_n(b,e,n,v,bp)	Wyszukuje pierwszych **n** elementów o wartości **v** dla których **bp** ma wartość **true**

Wykorzystanie algorytmów STL znacznie ułatwia pisanie programów. Omówimy prosty program wykorzystujący algorytm **count(b,e,v).** Algorytm **count(b,e,v)** wyszukuje w zbiorze wszystkie elementy o podanej wartości. Wyszukiwanie wartości może być przeprowadzone dla wszystkich elementów zbioru a także dla wybranego zakresu. Zakres przeszukiwania określony jest przedziałem [**b, e**], poszukiwaną wartość określa argument **v.**

Wydruk 5.4. Przykład użycia algorytmu **count(b,e,v).**

```
#include <iostream>
#include<conio.h>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{int n;
 int ar[6] = { 2,8,8,15,18, 21};
 vector<int> war(ar,ar+6);
 n = count(war.begin(), war.end(), 8);
 cout << "wartosc 8 wystepuje "<< n << " razy"<<endl;

  getche();
  return 0;
}
```

Wynik działania programu jest następujący:

wartosc 8 wystepuje 2 razy

W kontenerze **vector** umieszczamy zbiór elementów:

{ 2,8,8,15,18, 21}

Umieszczenie elementów w kontenerze **vector** realizują instrukcje:

```
int ar[6] = { 2,8,8,15,18, 21};   //deklaracja tablicy z inicjalizacją
vector<int> war(ar,ar+6);          //elementy ar kopiowane do war
```

Algorytm **count()** zapisany w postaci:

n = count(war.begin(), war.end(), 8);

w zakresie wyznaczonym przez [**war.begin(), war.end()**], szuka wartości **v** = 8. Podany zakres oznacza, że przeszukiwany jest cały zbiór. Algorytm **count()** zwraca ilość wystąpień podanej wartości, w naszym przykładzie - liczby 8.

Następny przykład pokazuje działanie algorytmu **count_if(b,e,up).** Algorytm **count_if()** szuka w podanym zakresie **[b, e]** elementów spełniających warunek podany w predykacie **up**. Zadaniem algorytmu **count_if()** jest wyszukanie w podanym zbiorze liczb wszystkich liczb podzielnych przez 3 i podanie ich ilości.

Wydruk 5.5. Przykład użycia algorytmu **count_if(b,e,pb).**

```
#include <iostream>
#include<conio.h>
#include <vector>
#include <algorithm>

bool warunek(int x)
{ if ((x%3) == 0) return true;
                return false;
}

using namespace std;

int main()
{int n;
 int ar[6] = {2,8,8,15,18,21};
 vector<int> war(ar,ar+6);
 n = count_if(war.begin(), war.end(), warunek);
 cout << "mamy "<< n << " liczby podzielne przez 3"<<endl;

  getche();
  return 0;
}
```

Wynik działania programu:

mamy 4 liczby podzielne przez 3

Dla zbioru liczb:

{2,8,8,15,18,21}

warunek wyszukiwania zdefiniowany jest następująco:

```
bool warunek(int x)
{ if ( ( x%3 ) == 0)   return true;
                        return false;
}
```

Algorytm zapisany w postaci:

```
n = count_if(war.begin(), war.end(), warunek);
```

w zakresie **[war.begin(), war.end()]** szuka liczb spełniających predykat o nazwie **warunek,** to znaczy wyszukuje liczb podzielnych przez 3 oraz zlicza

ilość ich wystąpień.
Dobrym rozwiązaniem programistycznym jest wykorzystywanie algorytmu **for_each(),** gdyż jest to bardzo elastyczny algorytm pozwalający wykonywać operacje na każdym elemencie kolekcji. Poniżej pokazany jest program, w którym zamiast pętli **for** lub **while,** wykorzystany jest algorytm **for_each().** W programie w kontenerze **list** umieszczamy nazwy mebli. Program ma wyświetlić wprowadzone napisy.

Wydruk 5.6. Przykład użycia algorytmu **for_each(b,e,up).**

```
#include <iostream>
#include<conio.h>
#include <string>
#include <list>
#include <algorithm>
using namespace std;

void drukuj (string& napis)
 {  cout << napis << endl;
 }

int main ()
{
  list<string> meble;
  meble.push_back("kanapa");
  meble.push_back("stolik");
  meble.push_back("szafa");
  meble.push_front("fotel");
  for_each (meble.begin(), meble.end(), drukuj);
  getche();
  return o;
}
```

W wyniku uruchomienia programu mamy komunikat:

```
fotel
kanapa
stolik
szafa
```

Algorytm **for_each()** pobiera kolejno każdy element zbioru określonego iteratorami wejściowymi **[meble.begin(), meble.end()]** i traktuje je jak argument funkcji **drukuj().** Zadaniem tej funkcji jest wyświetlenie napisu. W ten sposób wyświetlamy na ekranie monitora wprowadzone wszystkie napisy do kontenera **list.** Należy pamiętać, że algorytm **for_each()** jest także zaliczany do

algorytmów modyfikujących, ponieważ dopuszcza wykonanie żądanej operacji na swoim argumencie. Argument taki musi być przekazany przez referencję.
Algorytm **find_if(b,e,up)** wyszukuje zakres dla elementu, dla którego zdefiniowany predykat unarny **up** zwraca wartość prawda. Zakres wyznaczają iteratory wejściowe **[b, e].**

Zastosowanie kolejnego algorytmu **find_if()** ilustruje dość ciekawy program 5.7. Mamy zapisane informacje o obrazach dwóch wspaniałych malarzy – H. Boscha oraz P. Bruegela. Każdy rekord zawiera nazwisko malarza, datę, tytuł obrazu oraz miasto, w którym znajduje się wymieniony obraz. Po wprowadzeniu daty, ukazuje się rekord związany z tą datą. Gdy wprowadzonej daty nie ma na liście, ukazuje się komunikat, że nic nie wiadomo o pracach malarzy w tym roku. Omawiany program ilustruje technikę przetwarzania tekstu.

Wydruk 5.7. Przykład użycia algorytmu **find_if()(b,e,up).**

```
#include <iostream>
#include <conio.h>
#include <string>
#include <list>
#include <algorithm>
using namespace std;
string s;
struct sztuka
 { bool operator () (string& zapis)
    { return zapis.substr(10,4)== s;
    }
 };
int main ()
{ list<string> obraz;
 cout << "podaj rok : ";
 cin >> s;
// pozycja          0123456789012345678901234567890123456789012
 obraz.push_back("H.Bosch   1475  Leczenie glupoty          Madryt   ");
 obraz.push_back("H.Bosch   1505  Kuszenie sw. Antoniego    Lizbona ");
 obraz.push_back("H.Bosch   1500  Woz z sianem             Madryt  ");
 obraz.push_back("H.Bosch   1495  Statek glopcow           Paryz    ");
 obraz.push_back("P.Bruegel 1557  Upadek Ikara            Bruksela");
 obraz.push_back("P.Bruegel 1559  Przyslowia niderlandzkie  Berlin    ");
 obraz.push_back("P.Bruegel 1562  Triumf smierci           Madryt   ");
 obraz.push_back("P.Bruegel 1563  Wieza Babel             Wieden  ");
 list<string>::iterator p =
                find_if (obraz.begin(),obraz.end(),sztuka());
 if (p==obraz.end())
     cout << "nie ma tego faktu" << endl;
 else
     cout << *p << endl;
```

```
  getche();
  return 0;
}
```

Po uruchomieniu programu mamy następujący wydruk:

```
podaj rok : 1495
H.Bosch   1495  Statek glopcow          Paryz
```

Dane w postaci napisu są wprowadzone w ustalony sposób. Poszczególne informacje zapisane są w ustalonych pozycjach. Data zajmuje cztery pozycje, zaczyna się od pozycji dziesiątej:

```
// pozycja            01234567890123456789012345678901234567890123
  obraz.push_back("H.Bosch   1475  Leczenie glupoty              Madryt   ");
```

Algorytm **find_if()** przegląda wszystkie elementy kontenera **list** i sprawdza, czy nasz predykat jednoargumentowy:

```
struct sztuka
{ bool operator () (string& zapis)
     { return zapis.substr(10,4)== s;
      }
};
```

ma wartość **true**. Występujące w tej strukturze wyrażenie:

```
zapis.substr(10,4)== s;
```

zawiera funkcję :

```
substr(10,4)
```

Jest to metoda klasy **string**. Prototyp tej funkcji ma postać:

```
string string:: substr ( size_type idx, size_type len) const.
```

Zwraca ona fragment łańcucha znakowego zawierającego **len** znaków łańcucha ***this**, fragment zaczyna się od pozycji **idx**. W pokazanym przykładzie fragment zawiera cztery znaki i zaczyna się od dziesiątej pozycji. Algorytm zwraca pozycję elementu, dla którego nasz predykat jest prawdziwy:

```
list<string>::iterator p =  find_if (obraz.begin(),obraz.end(),sztuka());
```

i w przypadku sukcesu wyświetla ten element:

```
{ cout << *p << endl;
```

5.3. Algorytmy modyfikujące

Algorytmy modyfikujące (ang. *modifying algorithms*), jak wskazuje nazwa mogą modyfikować elementy umieszczone w kontenerze. Lista algorytmów modyfikujących pokazana jest w tabeli 5.2

Tabela 5.2 Algorytmy modyfikujące
(b – początek zakresu, b1-początek drugiego zakresu, b2-początek trzeciego zakresu, e – koniec pierwszego zakresu, e1 – koniec drugiego zakresu, up – predykat unarny, bp – predykat binarny, v – wartość, v1 – wartość, n – ilość, g- operacja)

Algorytm	działanie
copy(b,e,b1)	Kopiuje elementy z zakresu [b,e] do zakresu [b1, b1+(e-b)]
copy_backward(b,e,b1)	Kopiuje elementy z zakresu [b,e] do zakresu [b1, b1-(e-b)]
fill(b,e,v)	Ustawia wartość każdego elementu kolekcji wartością **v**
fill_n(b,n,v)	Ustawia wartość n elementom kolekcji wartością **v**
generate(b,e,g)	Ustawia wartość każdego elementu kolekcji wartością wygenerowaną przez obiekt funkcyjny **g**
generate_n(b,n,g)	Ustawia wartość **n** elementom kolekcji wartością wygenerowaną przez obiekt funkcyjny **g**
merge(b,e,b1,e1,b3)	Scala elementy z posortowanych zakresów i umieszcza je w zakresie od **b3**
merge(b,e,b1,e1,b3, g)	Scala elementy z posortowanych zakresów i umieszcza je w zakresie od **b3**, kryterium sortowania ustala predykat **g**
replace(b,e,v,v1)	Zastępuje każde wystąpienie wartości **v**, wartością **v1** w zakresie [b,e]
replace_if(b,e,g,v,v1)	Zastępuje każde wystąpienie wartości v, wartością **v1** w zakresie

	[b,e], gdy spełniony jest predykat **g**
replace_copy(b,e,b1,v,v1)	Kopiuje elementy z zakresu [b,e] do zakresu od **b1**, zastępując wartość **v**, wartością **v1**
replace_copy_if(b,e,b1,g,v1)	Kopiuje elementy z zakresu [b,e] do zakresu od **b1**, zastępując wartość **v**, wartością **v1**, gdy spełniony jest predykat **g**
swap_ranges(b,e,b1)	Zamienia miejscami elementy z zakresu [b,e] od pozycji **b1**
transform(b,e,b1,up)	Przekształca elementy z zakresu [b,e] zgodnie z predykatem **up** i umieszcza je w zakresie docelowym, zaczynając od **b1**
transform(b,e,b1,bp)	Przekształca elementy z zakresu [b,e] zgodnie z predykatem **bp** i umieszcza je w zakresie docelowym, zaczynając od **b1**

W poniższym programie demonstrujemy wykorzystanie algorytmów modyfikujących.

Wydruk 5.8. Algorytmy **generate(), generate_n(), replace_if(), copy()**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>                    //ostream_iterator
bool parzysta (int &n2)                //predykat unarny
  { return ( ( n2%2) == 0) ; }
using namespace std;
int main()
{ int i;
  vector <int> k1(6), k2(6);
  cout<< "\n algorytm generate() :" << endl;
  generate(k1.begin(), k1.end(), rand);
  for (i=0; i<k1.size(); i++)
      cout << k1[i]<< ' ';
  cout<< "\n algorytm replace_if() :" << endl;
  replace_if(k1.begin(),k1.end(),parzysta,999);
  copy (k1.begin(),k1.end(),
      ostream_iterator<int> (cout, "  "));
  cout<< "\nalgorytm generate_n() :" << endl;
  generate_n(k2.begin(),3, rand);
  for (i=0; i<k2.size(); i++)
      cout << k2[i]<< ' ';
```

```
  cout<< "\n algorytm copy() :" << endl;
  copy (k2.begin(),k2.end(),
      ostream_iterator<int> (cout, "  "));
  return 0;
}
```

Wynikiem uruchomienia programu jest komunikat:

```
algorytm generate() :
130  10982  1090  11656  7117  17595
algorytm replace_if()  :
999  999 999  999  7117  17595
algorytm generate_n() :
6415  22948  31126  0  0  0
algorytm copy() :
6415  22948  31126  0  0  0
```

W programie tworzymy dwa kontenery **k1** i **k2**, w każdym znaleźć się może sześć elementów:

```
vector <int> k1(6), k2(6);
```

W instrukcji:

```
generate(k1.begin(), k1.end(), rand);
```

algorytm **generate()** wykorzystuje funkcję **rand(),** do generowania sześciu liczb losowych i umieszcza je w wektorze **k1**. Prototyp funkcji **rand()** ma postać:

```
int rand(void);
```

Funkcja **rand()** generuje liczby pseudolosowe całkowite z zakresu od 0 do **RAND_MAX**. Okres generatora wynosi 2^{32}.
W instrukcji:

```
replace_if(k1.begin(),k1.end(),parzysta,999);
```

algorytm **replace_if()** przegląda elementy kontenera **k1** z zakresu **[k1.begin(), k1.end()]**, przy pomocy predykatu jednoargumentowego:

```
bool parzysta (int &n2)                    //predykat unarny
  { return ( ( n2%2) == 0) ; }
```

Algorytm sprawdza także, czy element jest liczbą parzystą czy nie. Gdy element jest liczbą parzystą zastępuje jego wartość argumentem algorytmu **replace_if()**, w naszym przypadku liczbą 999, w przeciwnym przypadku, (gdy liczba jest

nieparzysta) nie zmienia badanego elementu. Kolejna instrukcja:

```
copy (k1.begin(),k1.end(), ostream_iterator<int> (cout, "  "));
```

wywołuje algorytm **copy().** Jest to dość specyficzne wykorzystanie algorytmu **copy().**
Algorytm w pokazanej postaci kopiuje elementy kontenera **k1** do strumienia **cout** wykorzystując iterator strumieniowy wyjściowy. Użycie iteratora **ostream_iterator** wymaga dołączenia pliku **<iterator>.** Ta użyteczna postać algorytmu **copy(),** pozwala nam drukować elementy kontenera bez użycia pętli.
Kolejny algorytm **generate_n()**:

```
generate_n(k2.begin(),3, rand);
```

generuje zadaną ilość elementów, w naszym przypadku generuje trzy liczby losowe i umieszcza je w kontenerze **k2**. Generowane elementy umieszczane są kolejno od pozycji określonej pierwszym argumentem, w naszym przypadku jest to początek kontenera (**k2.begin()**).

5.4. Algorytmy usuwające

Algorytmy usuwające (ang. *removing algorithms*) stanowią podklasę algorytmów modyfikujących. Tego typu algorytmy albo usuwają elementy z kolekcji albo usuwają i jednocześnie kopiują. Cechą charakterystyczna algorytmów jest usuwanie elementów dla określonych wartości lub tych, które spełniają określone kryteria. Listaalgorytmów usuwających została przedstawiona w tabeli 5.3

Tabela 5.3 Algorytmy usuwające
(b – początek zakresu, b1-początek drugiego zakresu, e – koniec pierwszego zakresu, up – predykat unarny, bp – predykat binarny, v – wartość)

Algorytm	działanie
remove(b,e,v)	usuwa elementy z zakresu [b,e] o wartości **v**
remove_if(b,e,up)	usuwa elementy z zakresu [b,e] dla których predykat **up** ma wartość **true**
remove_copy(b,e,b1,v)	Kopiuje każdy element kolekcji z zakresu [b,e] różny od wartości **v** do zakresu od **b1**
remove_copy_if(b,e,b1,up)	Kopiuje każdy element kolekcji z zakresu [b,e] dla którego **up** ma wartość **false** do zakresu od **b1**
unique(b,e)	Usuwa wszystkie elementy z zakresu [b,e] gdy są równe poprzedniemu, ciąg

	musi być posortowany
unique(b,e,bp)	Usuwa wszystkie elementy z zakresu [b,e] gdy dla elementu **bp** ma wartość **true**
unique_copy(b,e,b1)	Kopiuje wszystkie elementy z zakresu [b,e] do pozycji określonej przez **b1** z wyjątkiem powtarzających się
unique_copy(b,e,b1,bp)	Kopiuje wszystkie elementy z zakresu [b,e] do pozycji określonej przez b1 , kryterium porównania jest predykat bp

Kolejny przykład omawia wykorzystanie algorytmu usuwającego **unique_copy().** Omówione będą różne aspekty wykorzystania takiego algorytmu, ponieważ wyniki jego działania mogą być nieoczekiwane.

Wydruk 5.9. Przykład użycia algorytmu **unique_copy()**

```
#include <iostream>
#include<conio.h>
#include <vector>
#include <algorithm>
#include <iterator>

bool warunek(int a,int b)
{ return (a+2 == b) || (a-2 == b);
}

using namespace std;

int main()
{
 int ar[11] = {2,8,8,8,15,17,18,20,18,2,4};
 vector<int> k1(ar,ar+11);
 ostream_iterator<int> output (cout, "  ");
 cout << "sekwencja wejsciowa: "<< endl;
 copy (k1.begin(),k1.end(),output);
 cout << "\nsekwencja wyjsciowa 1: "<< endl;
  unique_copy(k1.begin(),k1.end(),k1.begin());
 copy (k1.begin(),k1.end(),output);
 cout << "\nsekwencja wyjsciowa 2: "<< endl;
  unique_copy(k1.begin(),k1.end(),output);
 vector<int> w;
 cout << "\nsekwencja wyjsciowa 3: "<< endl;
  unique_copy(k1.begin(),k1.end(),back_inserter(w));
 copy(w.begin(),w.end(),output);
 cout << "\nsekwencja wyjsciowa 4: "<< endl;
```

```
 unique_copy(k1.begin(),k1.end(),output,warunek);
 getche();
 return 0;
}
```

Po uruchomieniu tego programu mamy następujący wynik:

```
sekwencja wejsciowa:
2  8  8   8   15  17  18  20  18  2  4
sekwencja wyjsciowa 1:
2  8  15  17  18  20  18  2   4   2  4
sekwencja wyjsciowa 2:
2  8  15  17  18  20  18  2   4   2  4
sekwencja wyjsciowa 3:
2  8  15  17  18  20  18  2   4   2  4
sekwencja wyjsciowa 4:
2  8  15  18  18  2   2
```

Algorytm **unique_copy()** jest kombinacją dwóch algorytmów – **copy()** i **unique(),** co oznacza, że kopiuje wszystkie elementy z podanego zakresu do zakresu docelowego i zgodnie z ustalonym wcześniej sposobem usuwa wybrane elementy. Pokazane w tabeli algorytmy usuwają elementy przez nadpisywanie, co oznacza, że nie zmieniają liczby elementów w zakresie kolekcji, na której operują, co może prowadzić do nieoczekiwanych wyników. W omawianym programie w instrukcjach:

```
 int ar[11] = {2,8,8,8,15,17,18,20,18,2,4};
vector<int> k1(ar,ar+11);
```

kontener **k1** wykorzystując zainicjalizowaną jedenastoelementową tablicę **ar** otrzymuje te elementy i wobec tego zawiera następujące wartości całkowite;

```
2,  8,  8,  8,  15,  17,  18,  20,  18,  2,  4
```

Do wydruku elementów wykorzystamy iterator wyjściowy i algorytm **copy():**

```
ostream_iterator<int> output (cout, "  ");
cout << "sekwencja wejsciowa: "<< endl;
copy (k1.begin(),k1.end(),output);
```

Do usunięcia powtarzających się sąsiednich elementów kontenera wykorzystamy algorytm **unique_copy(b,e,b1):**

```
unique_copy(k1.begin(),k1.end(),k1.begin());
```

w wyniku otrzymamy następujący wynik:

sekwencja wyjściowa: 2 8 15 17 18 20 18 2 4 2 4

porównując ten wynik z sekwencją wejściową

sekwencja wejściowa: 2, 8, 8, 8, 15, 17, 18, 20, 18, 2, 4

widzimy, że sąsiadujące elementy zostały usunięte (liczby 8). Łatwo zauważyć, że liczby o wartości 18 zostały w sekwencji – pamiętamy, że algorytm usuwa tylko powtarzające się liczby ***sąsiednie***.

Zauważamy także, że mimo usunięcia powtarzających się elementów, program wydrukował ponownie 11 elementów (dodatkowa sekwencja 2, 4 na końcu wydruku). Jest to nieoczekiwany wynik. Ten fakt można tłumaczyć, że dopisane na końcu elementy (2 i 4) logicznie nie należą do kolekcji. W jaki sposób możemy sobie poradzić z tym faktem wyjaśnimy w kolejnym przykładzie omawiając algorytm **remove().** W kolejnej instrukcji:

```
unique_copy(k1.begin(),k1.end(),output);
```

wysłano zmodyfikowaną sekwencję bezpośrednio do strumienia wyjściowego **output**, a nie jak poprzednio na początek kolekcji **k1**. Wynik działania tego wywołania algorytmu jest identyczny z poprzednim. W kolejnym fragmencie programu:

```
vector<int> w;
cout << "\nsekwencja wyjsciowa 3: "<< endl;
unique_copy(k1.begin(),k1.end(),back_inserter(w));
copy(w.begin(),w.end(),output);
```

tworzymy nowy kontener o nazwie **w**, algorytm **unique_copy()** usuwa elementy w kontenerze **k1** i zmodyfikowany zbiór przesyła do kontenera **w**. Jako argument algorytm wykorzystał tzw. wstawiacz końcowy (jest to iterator wstawiający) o nazwie **back_inserter(nazwa_kontenera)**, który wykorzystując metodę **push_back** wstawia elementy na końcu kontenera. Wynik działania tego algorytmu jest identyczny z poprzednim.

Na końcu programu wywołujemy algorytm **unique_copy()** z predykatem binarnym:

```
unique_copy(k1.begin(),k1.end(),output,warunek);
```

Algorytm operuje na elementach kontenera **k1** w zakresie **[k1.begin(), k1.end()],** trzeci argument określa pozycję, gdzie mają być umieszczane przetworzone elementy, w naszym przykładzie elementy kierowane są do strumienia wyjściowego. Czwarty argument (w tym przypadku o nazwie

warunek) jest predykatem dwuargumentowym:

```
bool warunek(int a,int b)
  { return (a+2 == b) || (a-2 == b);  }
```

Algorytm **unique_copy()** w zaprezentowanej postaci działa odmiennie niż to pokazano w poprzednich przykładach – nie usuwa sąsiadujących, powtarzających się elementów. Algorytm z predykatem wykonuje operacje na sąsiednich argumentach i sprawdza warunek. W naszym przypadku sprawdzamy, czy sąsiednie elementy różnią się o 2. Sekwencja wejściowa ma postać:

2 8 15 17 18 20 18 2 4 2 4

Po przetworzeniu mamy wynik:

2 8 15 18 18 2 2

Widzimy, że liczba 17 została usunięta (poprzednia wartość to 15, czyli usunięta liczba jest większą o 2 od poprzedniej, sąsiadującej), podobnie usunięto z kolekcji wartość 20 oraz 4.

W kolejnym przykładzie omówimy algorytm **remove()** oraz pokażemy, jak poradzić sobie można z nadpisywaniem niepotrzebnych elementów kolekcji. Algorytm **remove()** usuwa elementy z zakresu, nie zmienia ich ilości. Należy jednak mieć na uwadze, że algorytm zwraca nową pozycję końca kolekcji. Korzystając z tej informacji możemy zmniejszyć rozmiar kolekcji.

Wydruk 5.10. Przykład użycia algorytmu **remove()**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{int ar[11] = {2,8,8,8,15,17,18,20,18,2,4};
 vector<int> k1(ar,ar+11);
 ostream_iterator<int> output (cout, "  ");
 vector<int>::iterator e1;
 cout << "sekwencja wejsciowa: "<< endl;
 copy (k1.begin(),k1.end(),output);
 cout << "\nsekwencja wyjsciowa 1: "<< endl;
 remove(k1.begin(),k1.end(),8);
 copy (k1.begin(),k1.end(),output);
 e1 =  remove(k1.begin(),k1.end(),2);
 cout << "\nusunieto: " << distance(e1, k1.end())<< " elementy"<< endl;
 cout << "\sekwencja wyjsciowa 2: "<< endl;
```

```
 copy (k1.begin(),e1,output);
 cout << "\nsekwencja wyjsciowa 3: "<< endl;
 k1.erase(e1,k1.end());
 copy (k1.begin(),k1.end(),output);
  return 0;
}
```

Wyniki po uruchomieniu programu mają postać:

```
sekwencja wejściowa:
2  8   8   8  15   17  18  20  18  2  4
sekwencja wyjściowa 1:
2  15  17  18  20  18   2   4   18  2  4
usunieto:  3 elementy
sekwencja wyjściowa 2:
15  17  18  20  18   4   18   4
sekwencja wyjściowa 3:
15  17  18  20  18   4   18   4
```

Zastosowany algorytm **remove()** :

```
remove(k1.begin(),k1.end(),8);
```

przegląda elementy kontenera **k1** w zakresie **[k1.begin(), k1.end()]** i usuwa elementy, które mają zadaną wartość, w naszym przykładzie jest to 8. Jak widać w wyniku działania algorytmu usunięte zostały wszystkie wartości 8, ale na wydruku (sekwencja wyjściowa 1) liczba elementów się nie zmieniła, na końcu znalazły się powtórzenia elementów. W celu ograniczenia kolekcji, tak aby nie było nadpisywania, można skorzystać z rozwiązania zaproponowanego przez N. Josuttisa. Tworzymy iterator:

```
vector<int>::iterator e1;
```

i przypisujemy mu wartość:

```
e1 =  remove(k1.begin(),k1.end(),2);
```

Z poprzedniej kolekcji (sekwencja wyjściowa 1) algorytm **remove()** usuwa liczby o wartości 2 i zwraca pozycję w kolekcji po wykonaniu operacji. Jest to logiczny koniec zmodyfikowanej kolekcji. Tą wartość wykorzystamy w algorytmie **copy()** do wydrukowania poprawionej kolekcji:

```
copy (k1.begin(),e1,output);
```

w wyniku mamy spis elementów (bez dodatkowych elementów):

```
sekwencja wyjściowa 2:
15  17  18  20  18   4   18   4
```

W samym kontenerze **k1** ciągle są nadpisane wartości. Można na trwałe zmniejszyć nasz zbiór elementów.
Należy skorzystać z funkcji składowej kontenera **erase(),** która usuwa na trwałe wszystkie elementy kontenera zawarte w wyspecyfikowanym zakresie, pamiętamy, że **e1** określa koniec logiczny kolekcji :

```
k1.erase(e1,k1.end());
```

W wyniku mamy wydruk:

```
sekwencja wyjściowa 3:
15  17  18  20  18   4   18   4
```

Możemy także określić liczbę usuniętych elementów. W tym celu można wykorzystać funkcję pomocniczą iteratorów – **distance().** Funkcja **distance()** zwraca odległość pomiędzy dwoma iteratorami. Należy obliczyć różnicę pomiędzy logicznym i faktycznym końcem kolekcji:

```
distance(e1, k1.end())
```

Wynikiem działania instrukcji:

```
cout << "\nusunieto: " << distance(e1, k1.end())<< " elementy"<< endl;
```

jest wydruk:

```
usunieto:  3 elementy
```

5.5. Algorytmy mutujące

Algorytmy mutujące (ang. *mutating algorithms*) zamieniają kolejność elementów w kolekcji, nie zmieniają ich wartości. W tabeli 5.4 znajduje się zestawienie tych algorytmów.

Tabela 5.4 Algorytmy mutujące
(b – początek pierwszego zakresu, e – koniec pierwszego zakresu, b1 – początek drugiego zakresu, up – predykat unarny, bp – predykat binarny, op – funkcja)

Algorytm	działanie
next_permutation(b,e)	Zmienia kolejność elementów zgodnie z następną permutacją

next_permutation(b,e,bp)	Zmienia kolejność elementów zgodnie z następną permutacją, kolejność ustala predykat **bp**
prev_permutation(b,e)	Zmienia kolejność elementów zgodnie z poprzednią permutacją
prev_permutation(b,e,bp)	Zmienia kolejność elementów zgodnie z poprzednią permutacją, kolejność ustala **bp**
partition(b,e,up)	Przenosi na początek kolekcji elementy z zakresu [b,e] dla których **up** ma wartość **true**, względna kolejność może nie być zachowana
random_shufle(b,e)	Rozmieszcza losowo elementy w zakresie [b,e], zgodnie z rozkładem równomiernym
random_shufle(b,e, op)	Rozmieszcza losowo elementy w zakresie [b,e], zgodnie z rozkładem określonym przez funkcję **op**
reverse(b,e)	Odwraca kolejność elementów w zakresie [b,e]
reverse_copy(b,e,b1)	Odwraca kolejność elementów z zakresu [b,e] i kopiuje do zakresu docelowego od **b1**
rotate(b, b1,e)	Przesuwa cyklicznie w lewo elementy z zakresu [b,e]. Element z pozycji **b1** zostaje przesunięty na na pozycję **b**
rotate_copy(b,b1,e,op)	Kopiuje elementy z zakresu [b,e] cyklicznie tak jak **rotate(),** kolejność ustala **op**
stable_partition(b,e,up)	Przenosi na początek kolekcji elementy z zakresu [b,e] dla których **up** ma wartość **true**, zachowuje względną kolejność

Przykład wykorzystania algorytmu mutującego **reverse()** pokazany jest w kolejnym programie.

Wydruk 5.11. Przykład użycia algorytmu **reverse()**

```
#include <iostream>
#include<conio.h>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main()
{int i;
 vector<int> k1;
 ostream_iterator<int> output (cout, "  ");
 cout << "sekwencja wejsciowa: "<< endl;
 for (i=0;i<10;i++) k1.push_back(i);
 copy (k1.begin(),k1.end(),output);
 cout << "\nsekwencja wyjsciowa 1: "<< endl;
 reverse(k1.begin(),k1.end());
 copy (k1.begin(),k1.end(),output);
 cout << "\nsekwencja wyjsciowa 2: "<< endl;
 reverse(k1.begin()+5,k1.end());
 copy (k1.begin(),k1.end(),output);

  getche();
  return 0;
}
```

Wynikiem działania programu jest komunikat:

```
sekwencja wejsciowa:
0   1  2  3  4  5  6  7  8  9
sekwencja wyjsciowa 1:
9   8  7  6  5  4  3  2  1  0
sekwencja wyjsciowa 2:
9   8  7  6  5  0  1  2  3  4
```

W instrukcjach:

```
vector<int> k1;
for (i=0;i<10;i++) k1.push_back(i);
```

tworzymy kontener **k1** i przy pomocy pętli **for** umieszczamy w nim liczby całkowite z zakresu 0 - 9. Po wywołaniu algorytmu:

```
reverse(k1.begin(),k1.end());
```

kolejność elementów wektorze zostaje odwrócona. Odwracanie kolejności wykonane jest w zakresie **[k1.begin(), k1.end()],** odwrócona została kolejność wszystkich elementów. Możemy zmienić zakres odwracanych elementów:

```
reverse(k1.begin()+5,k1.end());
```

Określenie nowego zakresu odwracania **[k1.begin() + 5, k1.end()]** w algorytmie **reverse()** spowoduje pozostawicie pozostawienie pięciu pierwszych elementów wektora bez zmian, odwrócone zostaną pozostałe elementy poczynając od szóstego.
W kolejnym przykładzie ilustrujemy działanie algorytmu **rotate()** i **rotate_copy().**

Wydruk 5.12. Przykład użycia algorytmu **rotate()** i **rotate_copy()**

```
#include <iostream>
#include<conio.h>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{int i;
 vector<int> k1;
 ostream_iterator<int> output (cout, "  ");
 cout << "sekwencja wejsciowa: "<< endl;
 for (i=0;i<10;i++) k1.push_back(i);
 copy (k1.begin(),k1.end(),output);
 cout << "\nsekwencja wyjsciowa 1: "<< endl;
 rotate(k1.begin(),k1.begin() + 2,k1.end());
 copy (k1.begin(),k1.end(),output);
 cout << "\nsekwencja wyjsciowa 2: "<< endl;
 rotate_copy(k1.begin(),find(k1.begin(),k1.end(),0),k1.end(), output);
  getche();
  return 0;
}
```

Wyjście programu ma postać:

```
sekwencja wejsciowa:
0  1  2  3  4  5  6  7  8  9
sekwencja wyjsciowa 1:
2  3  4  5  6  7  8  9  0  1
sekwencja wyjsciowa 2:
0  1  2  3  4  5  6  7  8  9
```

Instrukcja:

```
rotate(k1.begin(),k1.begin() + 2,k1.end());
```

wywołuje algorytm **rotate().** Elementy z zakresu **[k1.begin(), k1.end()]** będą przesunięte cyklicznie, sekwencja będzie zaczynała się od trzeciego elementu, ponieważ drugi argument w **rotate()** ma postać:

```
k1.begin() + 2
```

Z kolei algorytm **rotate_copy():**

```
rotate_copy(k1.begin(),find(k1.begin(),k1.end(),0),k1.end(), output);
```

wykona szereg czynności. Najpierw znajdzie pozycję elementu o wartości 0, korzystając z algorytmu **find():**

```
find( k1.begin(), k1.end(), 0 )
```

Następnie wykona rotację elementów, na początku sekwencji umieszczona zostanie wartość 0, a cała sekwencja wysłana zostanie do strumienia wyjściowego, co spowoduje wydrukowanie wszystkich elementów.

5.6. Algorytmy sortujące

Algorytmy sortujące (ang. *sorting algorithms*) należą do kategorii algorytmów mutujących, ale ze względu na ich znaczenie klasyfikujemy te algorytmy oddzielnie. Algorytmy sortujące można podzielić na dwie grupy:

- Algorytmy sortujące ogólnego przeznaczenia
- Algorytmy sortujące stogowe (oparte o algorytm **heapsort**)

Zestaw algorytmów sortujących pokazany jest w tabeli 5.5 i 5.6

Tabela 5.5 Algorytmy sortujące ogólne
(b – początek pierwszego zakresu, e – koniec pierwszego zakresu,
b1 – początek drugiego zakresu, e1 – koniec pierwszego zakresu,
up – predykat unarny, bp – predykat binarny)

Algorytm	działanie
nth_element(b,n,e)	Znajduje w zakresie [b,e] element, który znajdowałby sie na n-tej pozycji, gdyby sekwencja była posortowana
nth_element(b,n,e,bp)	Znajduje w zakresie [b,e]

	element, który znajdowałby sie na n-tej pozycji, gdyby sekwencja była posortowana, **bp** ustala kolejność elementów
partial_sort(b,e1,e)	Częściowo sortuje elementy z zakresu [b,e]
partial_sort(b,e1,e, bp)	Częściowo sortuje elementy z zakresu [b,e], kryterium sortowania według **bp**
partial_sort_copy(b,e,b1,e1,bp)	Kopiuje posortowane elementy z zakresu [b,e] do zakresu [b1,e1] zgodnie z **op**
partition(b,e,up)	Przenosi na początek wszystkie elementy z zakresu [b,e], dla których **up** ma wartość **true**
sort(b,e)	Sortuje wszystkie elementy z zakresu [b,e]
sort(b,e,bp)	Sortuje wszystkie elementy z zakresu [b,e], kryterium sortowania zgodne z predykatem **up**
stable_partititon(b,e,up)	Przenosi na początek wszystkie elementy z zakresu [b,e], dla których **up** ma wartość **true**
stable_sort(b,e)	Sortuje wszystkie elementy z zakresu [b,e]
stable_sort(b,e,bp)	Sortuje wszystkie elementy z zakresu [b,e], kryterium sortowania zgodne z predykatem **up**

Tabela 5.6 Algorytmy sortujące stogowe
(b – początek pierwszego zakresu, e – koniec pierwszego zakresu,
bp – predykat binarny)

Algorytm	działanie
make_heap(b,e)	z elementów z zakresu [b,e] tworzy stóg
make_heap(b,e,bp)	z elementów z zakresu [b,e] tworzy stóg, kryterium sortowania zgodne z **bp**
pop_heap(b,e)	Przenosi najwyższy element stogu na ostatnią pozycję
pop_heap(b,e,bp)	Przenosi najwyższy element stogu na ostatnią pozycje, kryterium sortowania zgodne z **bp**

push_heap(b,e)	Dodaje do stogu wartość występującą na pozycji e-1
push_heap(b,e,bp)	Dodaje do stogu wartość występującą na pozycji e-1, kolejność elementów ustala bp
sort_heap(b,e)	Sortuje stóg
sort_heap(b,e, bp)	Sortuje stóg, ustala kolejność zgodnie z **bp**

Dla kontenera **list** algorytm **sort()** ma dwie wersje – algorytm niepobierający parametrów oraz wersję, która jako argument akceptuje funkcję z predykatem dwuargumentowy, w tym przypadku sortowanie odbywa się zgodnie z kryteriami ustalonymi w predykacie. W pokazanym programie sortowana będzie lista nazwisk słynnych kompozytorów muzyki poważnej.

Wydruk 5.13. Przykład użycia algorytmu **sort()**

```
#include <iostream>
#include <string>
#include <list>
#include <algorithm>
#include <conio>
using namespace std;
output (string& s) {cout << s << "   " ;}
bool lessLenght (string &s1, string &s2)
   {return s1.size() < s2.size() ;
   }
int main () {
 list<string> Kompozytor;
 Kompozytor.push_back("Bach");
 Kompozytor.push_back("Mozart");
 Kompozytor.push_back("Beethoven");
 Kompozytor.push_back("Vivaldi");
 Kompozytor.push_back("Wagner");

 cout << "\nnieposortpwana lista " << endl;
 for_each(Kompozytor.begin(), Kompozytor.end(), output );
 cout << "\nposortowana lista " << endl;
 Kompozytor.sort();
 for_each(Kompozytor.begin(), Kompozytor.end(), output);
 cout << "\nposortowana lista 2" << endl;
 Kompozytor.sort(lessLenght);
 for_each(Kompozytor.begin(), Kompozytor.end(), output);
 getche();
 return 0;
}
```

Wyjście programu ma następującą postać:

```
nieposortowana lista
Bach Mozart Beethoven Vivaldi Wagner
posortowana lista
Bach Beethoven Mozart Vivaldi Wagner
posortowana lista 2
Bach Mozart Wagner Vivaldi Bethoven
```

Na liście **Kompozytor** umieszczono pięć nazwisk słynnych kompozytorów. Wywołanie algorytmu sortującego ma prostą postać:

```
Kompozytor.sort();
```

Posortowane zostały wszystkie nazwiska umieszczone na liście w porządku leksykograficznym wzrastającym. Możemy posortować listę także ze względu na długość nazwisk kompozytorów. W tym celu należy wykorzystać algorytm **sort()** z predykatem dwuargumentowym **lessLenght():**

```
bool lessLenght (string &s1, string &s2)
  {return s1.size() < s2.size() ;}
```

W predykacie porównywane są długości łańcuchów, w tym celu wykorzystaliśmy funkcję **size().**

Sortowanie kolekcji można wykonać także częściowo, w tym celu korzystamy z algorytmu **partial_sort(),** pokazane jest to w kolejnym programie.

Wydruk 5.14. Przykład użycia algorytmu **partial_sort()**

```
#include <iostream>
#include <conio.h>
#include <vector>
#include <algorithm>
#include <iterator>       //ostream_iterator

using namespace std;
int main()
{ int i;
  vector <int> k1(10);
  cout<< "\n algorytm generate() :" << endl;
  generate(k1.begin(), k1.end(), rand);
  for (i=0; i<k1.size(); i++)
      cout << k1[i]<< ' ';
  cout << "\n sortowane 4 pierwsze elementy: "<< endl;
  partial_sort(k1.begin(), k1.begin()+3, k1.end());
```

```
    copy (k1.begin(),k1.end(), ostream_iterator<int> (cout, "  "));
    cout<< "\n sortowane wszystkie elementy malejaco :" << endl;
    partial_sort(k1.begin(), k1.end(), k1.end(), greater<int>());
    copy (k1.begin(),k1.end(), ostream_iterator<int> (cout, "  "));

    getche();
    return 0;
}
```

W wyniku działania programu otrzymamy następujący komunikat:

```
algorytm generate() :
130  10982  1090  11656  7117  17595  6415  22948  31126  9004
sortowane cztery pierwsze elementy:
130  1090  6415  11656  10982  17595  7117  22948  31126  9004
sortowane wszystkie elementy malejąco:
31126  22948  17595  11656  10982  9004  7117  6415  1090  130
```

Program generuje losowo liczby całkowite i umieszcza je w kontenerze **k1**:

```
generate(k1.begin(), k1.end(), rand);
```

W kolejnej instrukcji:

```
partial_sort(k1.begin(), k1.begin()+3, k1.end());
```

wywoływany jest algorytm **partial_sort().** Przy pomocy algorytmu **partial_sort()** sortujemy elementy, inaczej niż to wykonuje algorytm **sort().** Sortownie przebiega do momenty aż osiągniemy posortowane elementy do pozycji wskazywanej przez drugi argument. W naszym przypadku jest to

```
k1.begin()+3
```

to znaczy chcemy otrzymać tylko cztery pierwsze posortowane elementy kolekcji. Sortowanie algorytmem **partial_sort()** może przebiegać także zgodnie z ustaleniami predykatu dwuargumentowego, który przekazywany jest opcjonalnie, jako czwarty parametr:

```
partial_sort(k1.begin(), k1.end(), k1.end(), greater<int>());
```

Obiekt funkcyjny **greater<typ>** jest wykorzystywany, jako kryterium sortowania, spowoduje on, że porządek w kolekcji będzie rosnący. Domyślne kryterium sortowania jest określone przez obiekt funkcyjny **less<typ>,** dzięki któremu porządek sortowania jest rosnący.

Biblioteka STL w zbiorze algorytmów posiada algorytmy służący do realizacji sortowania stosu. Sortowanie stosu (ang. *heapsort*) polega na utworzenia specjalnego drzewa binarnego. W takim drzewie (zwanym stosem) największy element znajduje się zawsze na wierzchołku stosu. W określonych przypadkach sortowanie stosowe jest bardzo wydajne.

Poniższy przykład jest ilustracją wykorzystania algorytmu **sort_heap()** do sortowania stogowego.

Wydruk 5.15. Przykład użycia algorytmu **make_heap()** i **sort_heap()**

```
#include <iostream>
#include <conio.h>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{ int i;
  ostream_iterator<int> output (cout, "  ");
  int ar[10] = {3, 99, 100, 77, 25, 13, 44, 66, 11, 27};
  vector <int> k1(ar, ar+10);
  cout << "\n 1. kolekcja nieposortowana : "<< endl;
  copy (k1.begin(),k1.end(),output);
  cout << "\n 2. kolekcja nieposortowana, make_heap: "<< endl;
  make_heap(k1.begin(),k1.end());
  copy (k1.begin(),k1.end(),output);
  cout << "\n 3. kolekcja posortowana, sort_heap: "<< endl;
  sort_heap(k1.begin(),k1.end());
  copy (k1.begin(),k1.end(),output);
  cout << "\n 4. nowy element (33), push_back: "<< endl;
  k1.push_back(33);
  copy (k1.begin(),k1.end(),output);
  getche();
  return 0;
}
```

Po uruchomieniu programu mamy następujący wynik:

```
kolekcja nieposortowana :
3  99  100  77  25  13  44  66  11  27
kolekcja nieposortowana, make_heap:
100  99  44  77  27  13  3  66  11  25
kolekcja posortowana, sort_heap:
3  11  13  25  27  44  66  77  99  100
nowy element (33), push_back:
```

3 11 13 25 27 44 66 77 99 100 33

Oczywiście, aby można było stosować sortowanie stogowe, musimy mieć utworzony stos. W tym celu wykorzystywany jest algorytm **make_heap():**

make_heap(k1.begin(),k1.end());

dzięki któremu pobrana sekwencja wartości z wektora **k1** zostanie wykorzystana do utworzenia stosu.
Ponieważ argumenty tego algorytmu muszą być iteratorami o dostępie bezpośrednim, sortowania stogowe w zasadzie można wykonać tylko dla obiektów klasy **vector** i **deque**. Sortowanie stogowe realizuje instrukcja:

sort_heap(k1.begin(),k1.end());

Należy pamiętać, że po wywołaniu algorytmu **sort_heap(),** posortowany zakres nie jest już stosem. W zestawie algorytmów sortowania stogowego mamy dodatkowe algorytmy pozwalające na usuwanie elementów ze stogu (**pop_heap()**) i dodawanie nowych elementów do stogu (**push_heap()**). Działanie tego typu operacji przedstawione jest w kolejnym programie.

Wydruk 5.16. Algorytmy **make_heap(), pop_heap(), sort_heap()**

```
#include <iostream>
#include <conio.h>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{ ostream_iterator<int> output (cout, "  ");
  int ar[10] = {3, 99, 100, 77, 25, 13, 44, 66, 11, 27};
  vector <int> k1(ar, ar+10);
  cout << "\n 1. kolekcja nieposortowana : "<< endl;
  copy (k1.begin(),k1.end(),output);
   make_heap(k1.begin(),k1.end());
  cout << "\n 2. kolekcja posortowana : "<< endl;
  sort_heap(k1.begin(),k1.end());
  copy (k1.begin(),k1.end(),output);
  cout << "\n 3. usuwanie ze stosu : "<< endl;
  make_heap(k1.begin(),k1.end());
  pop_heap(k1.begin(),k1.end());
  k1.pop_back();
  copy (k1.begin(),k1.end(),output);
  cout << "\n 4. nowe sortowanie stosu : "<< endl;
  sort_heap(k1.begin(),k1.end());
  copy (k1.begin(),k1.end(),output);
```

```
  getche();
  return 0;
}
```

W wyniku działania programu otrzymujemy następujący komunikat:

```
kolekcja nieposortowana :
3  99  100  77  25  13  44  66  11  27
kolekcja posortowana:
3  11  13  25  27  44  66  77  99  100
usuwanie ze stosu :
 77  66  25  27  44  13  3  11
Nowe sortowanie stosu :
3  11  13  25  27  44  66  77  99
```

Podobnie, jak w poprzednim programie, korzystając z tablicy **ar** tworzymy wektor **k1**:

```
int ar[10] = {3, 99, 100, 77, 25, 13, 44, 66, 11, 27};
vector <int> k1(ar, ar+10);
```

Wykorzystując **make_heap()** oraz **sort_heap()** tworzymy stos i sortujemy nasze elementy. Ponieważ po zadziałaniu algorytmu **sort_heap()** kolekcja nie jest stosem, ponownie tworzymy stos korzystając z algorytmu **make_heap(),** możemy, więc wykonać operacje usuwania elementu ze stosu:

```
make_heap(k1.begin(),k1.end());
pop_heap(k1.begin(),k1.end());
k1.pop_back();
```

Funkcja **pop_back()** usuwa ostatni element wektora. Po usunięciu elementu spokojnie można wywołać algorytm **sort_heap(),** co spowoduje posortowanie stogowe kolekcji zmniejszonej o usunięty element.

5.7. Algorytmy przeznaczone dla zakresów posortowanych

W bibliotece STL znajdują się algorytmy wykonujące operacje na posortowanych zakresach (ang. *sorted range algorithms*). Tabela 5.7 przedstawia zestaw algorytmów przeznaczonych do operacji na posortowanych kolekcjach. Omawiane algorytmy mają dość skomplikowane definicje i zasady korzystania. W każdym przypadku ich użycia, należy zaznajomić się z ich opisem technicznym.

Tabela 5.7 Algorytmy operujące na kolekcjach posortowanych
(b – początek pierwszego zakresu, b1 – początek drugiego zakresu, b2 – początek trzeciego zakresu, e – koniec pierwszego zakresu, e1 – koniec drugiego zakresu, bp – predykat binarny, v - wartość)

Algorytm	działanie
binary_search(b,e,v)	Sprawdza, czy w zakresie [b,e] jest element v
binary_search(b,e,v,bp)	Sprawdza, czy w zakresie [b,e] jest element **v**, **bp** jest kryterium sortowania
equal_range(b,e,v)	Dla elementu **v** szuka miejsca wstawienia
equal_range(b,e,v,bp)	Dla elementu **v** szuka miejsca wstawienia, **bp** jest kryterium wyszukiwania
includes(b,e,b1,e1)	Sprawdza czy posortowany zakres [b,e] zawiera wszystkie elementy z zakresu [b1,e1]
includes(b,e,b1,e1,bp)	Sprawdza czy posortowany zakres [b,e] zawiera wszystkie elementy z zakresu [b1,e1], **bp** jest kryterium sortowania
inplace_merge(b,e,e1)	Scala elementy z zakresu [b,e] oraz [e,e1]
inplace_merge(b,e,e1,bp)	Scala elementy z zakresu [b,e] oraz [e,e1]
lower_bound(b,e,v)	Zwraca pozycję pierwszego elementu o wartości **v**
lower_bound(b,e,v,bp)	Zwraca pozycję pierwszego elementu o wartości **v**, **bp** jest kryterium sortowania
merge(b,e,b1,e,b2)	Scala wszystkie elementy z zakresu [b,e] i [b1,e1]
merge(b,e,b1,e,b2,bp)	Scala wszystkie elementy z zakresu [b,e] i [b1,e1], bp jest kryterium sortowania
set_difference(b,e,b1,e1,b2)	Scala elementy z zakresów [b,e] i [b1,e1], tak, że zbiór wynikowy zawiera elementy, które nie występują w drugim zbiorze.
set_difference(b,e,b1,e1,b2,bp)	Scala elementy z zakresów [b,e] i

	[b1,e1], tak, że zbiór wynikowy zawiera elementy, które nie występują w drugim zbiorze, **bp** - predykat
set_intersection(b,e,b1,e1,b2)	Scala dwa zbiory tak, że zbiór wynikowy zawiera elementy, które występują w obu zbiorach
set_intersection(b,e,b1,e1,b2, bp)	Scala dwa zbiory tak, że zbiór wynikowy zawiera elementy, które występują w obu zbiorach, bp – kryterium sortowania
set_symetric_difference(b,e,b1, e1, b2)	Scala dwa zbiory tak, że zbiór wynikowy zawiera elementy, które występują albo w pierwszym, albo w drugim, ale nie jednocześnie
set_symetric_difference(b,e,b1, e1, b2, bp)	Scala dwa zbiory tak, że zbiór wynikowy zawiera elementy, które występują albo w pierwszym, albo w drugim, ale nie jednocześnie, **bp** – kryterium sortowania
set_union(b,e,b1,e1,b2)	Scala dwa zbiory tak, że zbiór wynikowy zawiera elementy, które występują albo w pierwszym, albo w drugim, albo w obydwu
set_union(b,e,b1,e1,b2,bp)	Scala dwa zbiory tak, że zbiór wynikowy zawiera elementy, które występują albo w pierwszym, albo w drugim, albo w obydwu, **bp** jest kryterium sortowania
upper_bound(b,e,v)	Zwraca pozycję ostatniego elementu o wartości **v**
upper_bound(b,e,v,bp)	Zwraca pozycję ostatniego elementu o wartości **v**, **bp** jest kryterium sortowania.

Zilustrujemy prostym przykładem zastosowania wybranych algorytmów scalających. W programie utworzone zostaną dwa zbiory liczb całkowitych, a następnie wykonane zostaną podstawowe operacje na zbiorach:

- unia (ang. *union*)
- różnica (ang. *difference*)
- różnica symetryczna (ang. *symmetric difference*)
- przecięcie (ang. *intersection)*

Wydruk 5.17. Algorytmy scalające – **set_intersection, set_union, set_difference, set_symmetric_difference,**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{ ostream_iterator<int> output (cout, "  ");
  int aa[5] = {4,5,6,7,8};
  int ab[10] = {1,2,3,4,5,6,7,8,9,10};
  vector <int> ka(aa, aa+5);
  vector <int> kb(ab, ab+10);
  cout << "\nset ka: ";
  copy (ka.begin(),ka.end(),output);
  cout << "\nset kb: ";
  copy (kb.begin(),kb.end(),output);
  cout<<"\ndifference" << endl;
  set_difference(kb.begin(),kb.end(),ka.begin(),ka.end(),output);
  cout<<"\nintersection" << endl;
  set_intersection(kb.begin(),kb.end(),ka.begin(),ka.end(),output);
  cout<<"\nsymmetric_difference" << endl;
  set_symmetric_difference(kb.begin(),kb.end(),ka.begin(),ka.end(),output);
  cout<<"\nunion" << endl;
  set_union(ka.begin(),ka.end(),kb.begin(),kb.end(),output);
  //inne kodowanie :
  int a_set[15];
  int *p = set_union(aa,aa+5,ab,ab+10,a_set);
  cout<<"\nunion" << endl;
  copy(a_set, p, output);
  return 0;
}
```

Po uruchomieniu programu otrzymujemy następujący wynik:

```
set ka: 4  5  6  7  8
set kb: 1  2  3  4  5  6  7  8  9  10
difference
1  2  3  9  10
intersection
4  5  6  7  8
symmetric_difference
1  2  3  9  10
union
1  2  3  4  5  6  7  8  9  10
```

```
union
1 2 3 4 5 6 7 8 9 10
```

W programie tworzymy dwie tablice, a na ich podstawie dwa kontenery typu wektor:

```
int aa[5] = {4,5,6,7,8};
int ab[10] = {1,2,3,4,5,6,7,8,9,10};
vector <int> ka(aa, aa+5);
vector <int> kb(ab, ab+10);
```

Należy pamiętać, że algorytmy STL mogą działać na zwykłych tablicach, ponieważ wskaźnik do tablicy jest iteratorem o bezpośrednim dostępie. W instrukcji:

```
set_difference(kb.begin(),kb.end(),ka.begin(),ka.end(),output);
```

wykorzystany jest algorytm **set_difference()** do wyznaczenia elementów ze zbioru **kb**, które nie są zawarte w zbiorze **ka**. Zbiory muszą być uporządkowane rosnąco. Wybrane elementy zostają skopiowane do strumienia wyjściowego o nazwie **output** i wydrukowane. Algorytm zwraca iterator wyjściowy.
W instrukcji:

```
set_intersection(kb.begin(),kb.end(),ka.begin(),ka.end(),output);
```

algorytm **set_intersection** wyznacza, które elementy ze zbioru **kb** są także w zbiorze **ka**. Wspólne elementy są kopiowane do strumienia wyjściowego.
W instrukcji:

```
set_symmetric_difference(kb.begin(),kb.end(),ka.begin(),ka.end(),output);
```

algorytm **set_symmetric_difference()** określa elementy ze zbioru **kb**, które nie są w zbiorze **ka** i elementy zbioru **ka,** które nie są w zbiorze **kb**. Elementy, które są różne zostaną umieszczone w strumieniu wyjściowym **output.**
W instrukcji:

```
set_union(ka.begin(),ka.end(),kb.begin(),kb.end(),output);
```

algorytm **set_union()** tworzy zbiór z elementów, które znajdują się w pierwszym i drugim zbiorze, a następnie wysyła je do strumienia wyjściowego.
We fragmencie programu:

```
//inne kodowanie :
 int a_set[15];
 int *p = set_union(aa,aa+5,ab,ab+10,a_set);
```

```
cout<<"\nunion" << endl;
copy(a_set, p, output);
```

mamy przykład ponownego działania algorytmu **set_union(),** ale inaczej zakodowanego. Utworzona najpierw zostaje tablica **a_set[].** Algorytm **set_union** zwraca iterator wyjściowy umieszczony bezpośrednio po ostatniej wartości kopiowanej do tworzonego zestawu, na który wskazuje piąty argument (tutaj tablica **a_set**). Do wydruku danych wykorzystujemy algorytm **copy().** Pamiętamy, że definicja algorytmu **copy()** jest następująca:

```
OutputIterator
copy ( InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg )
```

Widzimy też wyraźnie, że algorytmy STL mogą operować także na klasycznych strukturach, jaką jest tablica.

5.8. Algorytmy numeryczne

Algorytmy numeryczne (ang. *numeric algorithms*) wykonują pewne operacje algebraiczne. W tabeli 5.8 pokazane są algorytmy numeryczne STL.

Tabela 5.8 Algorytmy numeryczne
(b – początek pierwszego zakresu, b1 – początek drugiego zakresu, e – koniec pierwszego zakresu, bp – predykat binarny, bp1 – predykat binarny, v - wartość)

Algorytm	działanie
adjacent_difference(b,e,b1)	Oblicza różnicę pomiędzy elementem i jego poprzednikiem
adjacent_difference(b,e,b1,bp)	Wywołuje operację **bp** dla elementu i jego poprzednika
accumulate(b,e,v)	Zwraca sumę wartości wszystkich elementów kolekcji i **v**
accumulate(b,e,v,bp)	Wykonuje operację **bp** na wszystkich elementach kolekcji oraz **v** i zwraca wynik
inner_product(b,e,b1,v)	Oblicza iloczyn skalarny dwóch zbiorów
inner_product(b,e,b1,v,bp,bp1)	Wykonuje operacje podobne do iloczynu skalarnego, operator + zastąpiony jest przez **bp** i **bp1**,
partial_sum(b,e,b1)	Oblicza sumy częściowe
partial_sum(b,e,b1,bp)	Wykonuje operacje podobne do obliczania sumy częściowej, zamiast operatora + stosuje **bp**

Algorytm **adjacent_difference()** przegląda wszystkie elementy kolekcji, operacja odejmowania wykonywana jest od drugiego elementu, pierwszy element jest przepisywany bez zmiany. Algorytm oblicza różnicę pomiędzy bieżącym elementem i poprzednim. Dwa pierwsze argumenty określają zakres wykonywania operacji, trzeci jest iteratorem wyjściowym, który określa miejsce przechowywania wyniku.
Jeżeli kolekcja wejściowa ma postać:

a1, a2 , a3 , a4 ,….

to kolekcja przekształcona algorytmem **adjacent_difference()** ma postać:

a1 , a2-a1 , a3-a2 , a4-a3…….

Gdy zastosujemy wersję algorytmu z dwuargumentowym predykatem, to przekształcona kolekcja będzie miała postać:

a1, a2 bp a1 , a3 bp a2 , a4 bp a3, ……..

Algorytm **accumulate()** oblicza sumę wszystkich wartości z zakresu **[b,e]** oraz **v** i zwraca tę sumę.
Jeżeli kolekcja wejściowa ma postać:

a1, a2 , a3 , a4 ,….

to algorytm **accumulate()** oblicza następującą sumę:

v + a1+ a2 + a3 + a4 +…….

Gdy zastosujemy czteroargumentową wersję algorytmu **accumulate(),** to algorytm zwróci następujący wynik:

v bp a1 bp a2 bp a3 bp a4 bp …….

Na przykład, gdy wykorzystamy obiekt funkcyjny **multiplies<typ>** to obliczony będzie iloczyn:

v*a1*a2*a3*a4*……

Algorytm **inner_product()** w najprostszej postaci wykonuje operacje obliczania iloczyny skalarnego.

Jeżeli mamy dwa zbiory:

a1, a2,a3

oraz

b1, b2,b3

i wywołamy algorytm **inner_product()** z czwartym argumentem **v=0,** to otrzymamy wynik:

0+ (a1+b1)+ (a2+b2) + (a3+b3)

Możemy także wywołać algorytm **inner_product()** z trzecim argumentem, którym może być iterator odwrotny (**rbegin()**) to w sytuacji, gdy np. **v=100** otrzymamy wynik:

100 + (a1+a3) + (a2+a2) + (a3+a1)

Algorytm **partial_sum()** wylicza sumy częściowe dla podanego ciągu. Jeżeli zbiór wejściowy ma postać:

a1, a2 , a3 , a4 ,….

to zostanie obliczony następujący ciąg:

a1, a1+ a2, a1+a2+a3, a1+a2+a3+a4, …….

Gdy zastosujemy wersję algorytmu z czwartym argumentem (dwuargumentową funkcją bp) to wygenerowany będzie następujący ciąg:

a1, a1 bp a2, a1 bp a2 bp a3, ……..

Poniższy program ilustruje wykorzystanie algorytmu numerycznego **inner_product().**

Wydruk 5.18. Algorytmy numeryczne, **inner_product()**

```
#include <iostream>
#include <conio.h>
#include <vector>
#include<numeric>
#include <algorithm>
#include <iterator>

using namespace std;
int main()
{ ostream_iterator<int> output (cout, "  ");
```

```
int a1[5] = {1,2,3,4,5 };
int a2[5] = {6,7,8,9,10};

vector <int> k1(a1, a1+5);
vector <int> k2(a2, a2+5);
vector <int> k3(a2,a2+5);
cout << "\n     zbior k1  : ";
copy (k1.begin(),k1.end(),output);
cout << "\n     zbior k2  : ";
copy (k2.begin(),k2.end(),output);
cout <<"\n  iloczyn skalarny k1.k2 : ";
cout << inner_product(k1.begin(),k1.end(), k2.begin(),0)<<endl;
cout <<"\n zanegowany k2 : ";
transform(k3.begin(),k3.end(),k3.begin(),negate<int>());
copy (k3.begin(),k3.end(),output);
cout <<"\n iloczyn skalarny k1.-k2 : ";
cout << inner_product(k1.begin(),k1.end(), k3.begin(),0)<<endl;
cout <<"\n odwrotny iloczyn skalarny k1.k2 : ";
cout << inner_product(k1.begin(),k1.end(), k2.rbegin(),0)<<endl;

getche();
return 0;
}
```

W wyniku działania program otrzymujemy następujący komunikat:

```
     zbior k1  :  1  2  3  4  5
     zbior k2  :  6  7  8  9  10
 iloczyn skalarny k1.k2 : 130
 zanegowany k2 : -6  -7  -8  -9  -10
iloczyn skalarny k1.-k2 : -130
odwrocony iloczyn skalarny k1.k2 : 110
```

Dla dwóch zbiorów:

```
k1  :  1  2  3  4  5
k2  :  6  7  8  9  10
```

program korzystając z algorytmu **inner_product():**

```
cout << inner_product(k1.begin(),k1.end(), k2.begin(),0)<<endl;
```

liczy iloczyn skalarny następująco:

k1.k2 = 0 + (1*6) + (2*7) + (3*8) + (4*9) + (5*10)= 6+14+24+36+50 = 130

Możemy odwrócić porządek w drugim zbiorze, aby otrzymać inną postać iloczynu skalarnego (korzystamy z funkcji **rbegin()**) :

```
cout << inner_product(k1.begin(),k1.end(), k2.rbegin(),0)<<endl;
```

W tym przypadku iloczyn skalarny liczony jest następująco:

k1.k2 = 0 + (1*10) + (2*9) + (3*8) + (4*7) + (5*6)= 10+18+24+28+30 = 110

W instrukcji :

```
transform(k3.begin(),k3.end(),k3.begin(),negate<int>());
```

wykorzystaliśmy algorytm modyfikujący **transform()** z funktorem **negate<typ>.** Wywołanie tego algorytmu tworzy zbiór z zanegowanymi wszystkimi elementami. Na marginesie zauważmy, że algorytm **transform()** może mieć wiele zastosowań. Na przykład dla obliczenia pierwiastka kwadratowego z każdego elementu kolekcji można by użyć funktora **sqrt<typ>:**

```
transform(k3.begin(),k3.end(),k3.begin(),sqrt<int>());
```

Pamiętajmy, że zawsze możemy tworzyć także własne funktory.

ROZDZIAŁ 6

OBIEKTY FUNKCYJNE

6.1. Wstęp..... 166
6.2. Opis obiektów funkcyjnych 173
6.3. Zastosowania obiektów funkcyjnych unarnych 178
6.4. Zastosowania obiektów funkcyjnych binarnych 181

6.1. Wstęp

Obiekty funkcyjne (zwane także funktorami) są obiektami klasy implementującej operator(). Obiekty funkcyjne (ang. *function objects*) są bardzo przydatne w programach wykorzystujących metody kontenerów oraz wspomagają działania algorytmów STL. Na poziomie pojęciowym, obiekty funkcje są obiektami działającymi jak funkcje. Choć funkcje i wskaźniki także mogą być klasyfikowane, jako obiekty funkcyjne, to obiekty klasy implementującej operator() są bardziej elastyczne i znacznie szybciej działające. Wiele funktorów, takich jak na przykład **less** lub **minus** dostarcza biblioteka STL. Użytkownik może także implementować własne funktory.

Mamy dwa typy obiektów funkcyjnych: jednoargumentowe (unarne) i dwuargumentowe (binarne). Zawsze należy sprawdzić, jakiego typu funktor jest wymagany przez konkretny algorytm lub metodę. Te dwa typy są umieszczone w STL i formalnie mają postać (A.Stepanov):

```
template <class Arg, class Result>
struct unary_function  {
        typedef Arg  argument_type;
                typedef Result  result_type;
};

template <class Arg1, calss Arg2, class Result>
struct binary_function  {
        typedef Arg1  first_argument_type;
                typedef Arg2  second_argument_type;
                typedef Result  result_type;
};
```

Biblioteka STL dostarcza następujące obiekty funkcyjne do wykonywania operacji arytmetycznych:

```
template <class T>
struct plus : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const {return x + y; }

template <class T>
struct minus : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const {return x - y; }

template <class T>
struct times : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const {return x * y; }
```

```
template <class T>
struct divide : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const {return x / y; }

template <class T>
struct modulus : binary_function<T, T, T> {
        T operator() (const T& x, const T& y) const {return x % y; }

template <class T>
struct negate : unary_function<T, T> {
        T operator() (const T& x) const {return -x ; }
```

Biblioteka STL dostarcza następujące obiekty funkcyjne do wykonywania operacji porównywania:

```
template <class T>
struct equal_to : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x == y; }

template <class T>
struct not_equal_to : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x != y; }

template <class T>
struct greater : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x > y; }

template <class T>
struct less : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x < y; }

template <class T>
struct greater_equel : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x >= y; }

template <class T>
struct less_equal : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x <= y; }
```

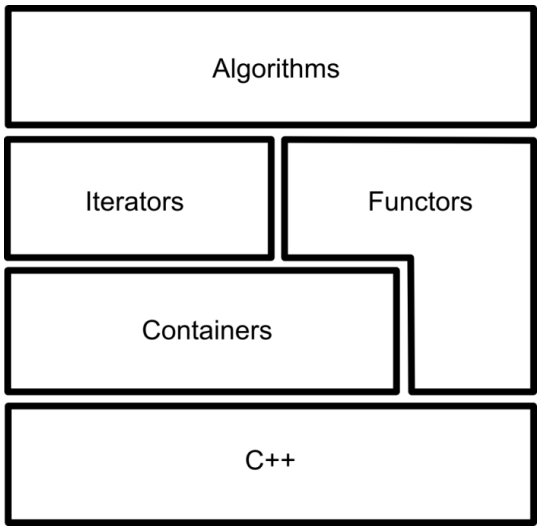
Biblioteka STL dostarcza następujące obiekty funkcyjne do wykonywania operacji logicznych:

```
template <class T>
struct logical_and : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x && y; }

template <class T>
struct logical_or : binary_function<T, T, bool> {
        bool operator() (const T& x, const T& y) const {return x || y; }

template <class T>
struct logical_not : unary_function<T,  bool> {
        bool operator() (const T& x) const {return !x ; }
```

Można powiedzieć, że zasadnicze elementy STL to : obiekty funkcyjne, kontenery, iteratory i algorytmy.



Rys. 6.1 Warstwowa struktura STL

Wymienione elementy współdziałają, tworząc wydajne programy w języku C++. Kontenery przechowują dane. Iteratory zapewniają dostęp do elementów przechowywanych w kontenerach, algorytmy umożliwiają typowe operacje programistyczne na elementach kontenerów. Obiekty funkcyjne wspomagają algorytmy, dzięki którym możemy używać uogólnionych algorytmów w sposób bardzo wydajny.

Należy pamiętać, że algorytmy mogą korzystać także ze zwykłych funkcji, definiowanych przez programistę.

Na przykład można zdefiniować prostą funkcję do wypisywania elementów i wykorzystać ją, jako argument w algorytmie. Taką konstrukcję ilustruje kolejny przykład.

Wydruk 6.1. Przykład użycia funkcji, jako argumentu algorytmu STL

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <conio.h>
using namespace std;

void drukuj(int x)
{ cout << x << ' '; }

int main()
{
 int a[4] = {1,22,333,4444 };
 vector<int> v(a,a+4);
 for_each(v.begin(), v.end(), drukuj);
 getche();
 return 0;
}
```

Wyjście uruchomionego programu ma postać:

1 22 333 4444

W programie została zdefiniowana funkcja **drukuj().** Algorytm **for_each()** wykonuje operacje na każdym elemencie kolekcji. Trzecim argumentem algorytmu **for_each()** jest funkcja **f**, co wynika wprost z jego z definicji:

```
template <class InputIterator, class Function>
void for_each(InputIterator first, InputIterator last, Function f);
```

Wynikiem działania algorytmu **for_each()** z funkcją zewnętrzną jest wydrukowanie wszystkich wartości kolekcji.

W programowaniu generycznym bardzo użyteczne są funkcje zwane predykatami (ang. *predicates*). Tym terminem określamy funkcje ogólne zwracające wartości boolowskie. W STL występują predykaty jednoargumentowe (unarne) oraz dwuargumentowe (binarne).

Predykat jednoargumentowy określa czy dana cecha jest prawdziwa czy fałszywa. Przykładem predykatu jednoargumentowego może być funkcja sprawdzająca, czy liczba jest pierwsza.

Wydruk 6.2. Predykat unarny, jako argument algorytmu STL

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <conio.h>
using namespace std;

bool pierwsza(int x)
 { if ( (x == 0) || (x ==1) )
      return false;
   int k;
   for (k = x/2; x%k != 0; --k)
     ;
      return k==1;
 }

int main()
{ vector<int> v;
 vector <int> :: iterator p;
 for( int i = 4; i <13; ++i)     v.push_back(i);
 for( p = v.begin(); p!= v.end(); ++p)
     cout << *p << ' ';
 p = find_if(v.begin(), v.end(), pierwsza);
 if (p != v.end())
    cout <<"\npierwsza znaleziona liczba pierwsza to : "<< *p <<endl;
 else
    cout << " nie znaleziono liczby pierwszej" << endl;
 getche();
 return 0;
}
```

W wyniku działania programu mamy komunikat:

```
4 5 6 7 8 9 10 11 12
pierwsza znaleziona liczba pierwsza to : 5
```

W programie tworzony jest kontener wektor, iterator, a przy pomocy pętli **for** umieszczane są w nim elementy:

```
vector<int> v;
vector <int> :: iterator p;
for( int i = 4; i <13; ++i)
      v.push_back(i);
```

Predykat unarny **pierwsza(int x)** sprawdza czy liczba jest pierwsza.

W instrukcji:

p = find_if(v.begin(), v.end(), pierwsza);

algorytm **find_if()** znajduje w sekwencji wartość, tą wartość, która spełnia wyspecyfikowane warunki dostarczone przez predykat. Formalnie szablon **find_if()** ma postać:

```
#include <algorithm>
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

i zwraca pierwszy iterator **p** w zakresie [**first,last**] gdy spełniony jest warunek:

pred(*i) == true.

Wykorzystanie predykatu dwuargumentowego prezentujemy w kolejnym programie. Elementy dwóch kontenerów są mnożone kolejno przez siebie, a iloczyny są sprawdzane, aby ustalić czy są parzyste, czy nie. Wyniki są umieszczane w trzecim kontenerze.

Wydruk 6.3. Użycie predykatu binarnego jako argumentu algorytmu STL

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <conio.h>
using namespace std;

bool parzysta(int x, int y)
 { if ((x*y)%2 == 0)
        return true;
   else
        return false;
  }

int main()
{ vector<int> v1,v2,vw;
  vector <int> :: iterator p;
  for( int i = 0; i < 5; ++i)
     { v1.push_back(i);
       v2.push_back(i+2);
```

```
    }
  for( p = v1.begin(); p!= v1.end(); ++p)
     cout << *p << ' ';
  cout << endl;
  for( p = v2.begin(); p!= v2.end(); ++p)
     cout << *p << ' ';
  vw.resize(5);
  transform ( v1.begin(), v1.end(),
              v2.begin(),
              vw.begin(), parzysta);
  cout <<"\nwynik:"<<endl;
  for( p = vw.begin(); p!= vw.end(); ++p)
     cout << *p << ' ';

  getche();
  return 0;
}
```

Wynik działania programu ma postać:

```
0 1 2 3 4
2 3 4 5 6
wynik:
1 0 1 0 1
```

Na początku deklarujemy trzy kontenery:

```
vector<int> v1,v2,vw;
```

dwa pierwsze przechowywać będą elementy, w trzecim wektorze vw umieszczamy wynik badania parzystości. W instrukcji:

```
vw.resize(5);
```

powiększamy wektor przechowujący wyniki tak, aby mógł przechować 5 elementów. Główne zadanie wykonuje algorytm **transform().** Algorytm wykonuje operacje zgodnie z predykatem, w naszym przypadku jest to wersja dla predykatu dwuargumentowego, pobiera elementy z zakresu [**b1,e1**] oraz [**b2,e2**] i wysyła wynik do miejsca przeznaczenia. W naszym przypadku elementy wektorów **v1** i **v2** na tych samych pozycjach są mnożone, predykat dwuargumentowy **parzysta()** sprawdza czy iloczyn jest parzysty i wysyła wynik (**true** albo **false**) do kontenera **vw.**

6.2. Opis obiektów funkcyjnych

W języku C++ mamy możliwość zmiany działania operatorów – operacje tego typu nazywamy przeciążeniem operatora. Istnieje możliwość przeciążenia operatora dzięki wywołaniu funkcji operatorowej - **operator ()**.
Po prawidłowym zdefiniowaniu takiego operatora w klasie, można używać obiektów klasy, jakby były wskaźnikami na funkcję.

Wydruk 6.4. Obiekt funkcyjny, klasyczne C++, jeden operator()

```
#include<iostream>
#include<conio.h>
using namespace std;

class funktor
{ public:
  int operator() (int x)
     { return(x/10) ; }
};

int main()
{  int x = 100 ;
   funktor operacja;
   cout << "wynik dzialania obiektu funkcyjnego: " << operacja(x) << endl;
  getche();
  return 0;
}
```

Po uruchomieniu programu mamy komunikat::

wynik dzialania obiektu funkcyjnego: 10

W klasie o nazwie **funktor** zdefiniowaliśmy przeciążony operator wywołania funkcji:

```
class funktor
{ public:
        int operator() (int x)
                { return(x/10) ; }
};
```

W programie tworzymy obiekt **operacja**, który następnie jest wywoływany i działa jak metoda klasy:

```
funktor operacja;
```

cout << "wynik dzialania obiektu funkcyjnego: " << operacja(x) << endl;

Zaletą stosowania obiektów funkcyjnych jest fakt, że można je przekazywać, jako funkcje zwrotne dla procedur oczekujących wskaźnik na funkcję. Należy także pamiętać, że w danej klasie możemy umieścić kilka przeciążonych operatorów wywołania funkcji. Ilustruje to zagadnienie kolejny przykład.

Wydruk 6.5. obiekt funkcyjny, klasyczne C++, podwójny operator()

```
#include<iostream>
#include<conio.h>
using namespace std;

class funktor
{ public:
  int operator() (int x)
        { return(x/10) ; }
  int operator() (int x,int y)
        { return(x+y)  ; }
};

int main()
{  int x = 100, y = 900 ;
   funktor operacja;
   cout << "wynik dzialania obiektu funkcyjnego: " << operacja(x) << endl;
    cout <<"wynik dzialania obiektu funkcyjnego: " << operacja(x,y) << endl;

  getche();
  return 0;
}
```

W wyniku działania programu otrzymujemy następujący komunikat:

```
wynik dzialania obiektu funkcyjnego:  10
wynik dzialania obiektu funkcyjnego: 1000
```

W klasie **funktor** zdefiniowaliśmy dwa operatory:

```
int operator() (int x)                  // 1
  { return(x/10) ; }
 int operator() (int x,int y)           // 2
  { return(x+y)  ; }
```

Pierwszy operator przyjmuje jeden argument, drugi operator przyjmuje dwa

argumenty. Widać wyraźnie, że obiekty funkcyjne są bardzo uniwersalne i elastyczne.

Ale dopiero zalety stosowania obiektów funkcyjnych można docenić, gdy zastosujemy je w obsłudze algorytmów STL. Jeżeli w danej klasie przeciążymy operator wywołania funkcji, to obiekty takiej klasy mogą być używane zamiast wskaźników funkcji. W STL jest kilkanaście predefiniowanych klas obiektów funkcyjnych (funktorów).
W szczególności mamy pięć szablonów klas funktorów dla dwuargumentowych operacji arytmetycznych:

- plus
- minus
- multiples
- divides
- modulus

W kolejnym przykładzie pokażemy proste wykorzystanie szablonu klasy **multiples**. Należy zauważyć, że pokazany przykład pełni rolę wybitnie dydaktyczną, gdyż lepszym rozwiązaniem jest wykorzystanie metody operator*, trzeba tylko upewnić się, czy dana metoda jest implementowana w danym kontenerze. Formalnie stosowanie funktorów wymaga dołączenia pliku <functional>, ale wiele nowych kompilatorów, włącza ten plik w tle.

Wydruk 6.6. STL - funktor **multiplies**

```
#include<iostream>
#include<conio.h>
#include<functional>
using namespace std;

int main()
{
 multiplies<int> iloczyn;
 cout << "obiekt funkcyjny STL, wynik: " <<  iloczyn(5, 6) ;

 getche();
 return 0;
}
```

Wynik działania programu jest następujący:

```
obiekt funkcyjny STL, wynik: 30
```

Bardzo często zachodzi potrzeba tworzenia własnych obiektów funkcyjnych. W tym celu korzystamy z ogólnych zasad definiowania metod klasy.

Wydruk 6.7. STL – własny obiekt funkcyjny

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>
#include<conio.h>
using namespace std;
void drukuj(const char *msg, vector<int> vect);

class op : unary_function<int, int>
{ public:
   result_type operator() (argument_type i)
    { return(result_type) i/10  ;
    }
 };
int main()
{ vector<int> v(10);
  for(unsigned i=0; i < v.size(); i++)
    v[i] = rand() % 100;
  drukuj("sekwencja in:\n", v);
  cout << endl;
  transform (v.begin(), v.end(),
              v.begin(),
              op());
  drukuj("sekwencja out:\n", v);
  sort(v.begin(), v.end(), greater<int>());
  drukuj("sekwencja sortowana out:\n", v);
  getche();
  return 0;
}

void drukuj(const char *msg, vector<int> vect)
 { cout << msg;
   for(unsigned i=0; i < vect.size(); ++i)
     cout << vect[i] << " ";
   cout << "\n";
 }
```

W wyniku działania programu mamy następujący komunikat:

```
sekwencja in:
30  82  90  56  17  95  15  48  26  4
sekwencja out:
3  8  9  5  1  9  1  4  2  0
sekwencja sortowana out:
9  9  8  5  4  3  2  1  1  0
```

Wielu autorów podręczników programowania (np. H.Schildt) zaleca w trakcie tworzenia własnych obiektów funkcyjnych wykorzystanie jednej z dwóch klas bazowych zdefiniowanych w STL:

- struct unary_function
- struct binary_function

Opracowany w STL szablon klasy zawiera ustalone nazwy typów ogólnych wykorzystywanych przez funktory.
W naszym programie, korzystając z szablonu **unary_function** tworzymy własny funktor o nazwie **op**, który pobiera argument i dzieli go przez 10:

```
class op : unary_function<int, int>
{ public:
  result_type operator() (argument_type i)
   { return(result_type) i/10  ;
   }
};
```

Przy pomocy pętli **for** oraz generatora liczb pseudolosowych **rand()** umieszczamy 10 elementów w wektorze **v**:

```
for(unsigned i=0; i < v.size(); i++)
      v[i] = rand() % 100;
```

Algorytm **transform()**:

```
transform (v.begin(), v.end(),  v.begin(), op() ) ;
```

przegląda kontener **v** i przy pomocy obiektu funkcyjnego **op()** wykonuje odpowiednie operacje na elementach sekwencji. Kolejny algorytm:

```
sort(v.begin(), v.end(), greater<int>());
```

wykorzystując predefiniowany binarny obiekt funkcyjny **greater<int>** wymusza sortowanie w porządku malejącym.

6.3. Zastosowania obiektów funkcyjnych unarnych

Funktory zdecydowanie upraszczają pisanie programów. Jak pamiętamy mamy trzy możliwości:

- skorzystać z jednego z predefiniowanych funktorów STL
- napisać własny funktor, wykorzystujący klasę bazową zdefiniowaną w STL
- napisać całkowicie własny funktor

W kolejnym przykładzie demonstrujemy sposób tworzenia własnego funktora jednoargumentowego, wykorzystanego przez algorytm **transform()** umieszczonego w funkcji użytkowej. Zadaniem programu jest zamiana dużych znaków łańcucha (o ile są) na znaki pisane małą literą.

Wydruk 6.8. STL – własny unarny obiekt funkcyjny, funkcja ogólna

```
#include <iostream>
#include <algorithm>
#include <conio.h>
using namespace std;

class to_lower
{public:
 char operator() (char c) const
   { return isupper (c)? tolower(c) : c;
   }
};

string zmiana (const string &str)
{ string st;
  transform ( str.begin(), str.end(),
              back_inserter(st),
                   to_lower());
  return st;
}

int main()
{string s("PROGRAM");
 cout << s << endl;
 s = zmiana (s);
 cout << s << endl;
  getche();
  return 0;
}
```

W wyniku działania programu mamy na ekranie komunikat:

```
PROGRAM
program
```

W klasie **to_lower**:

```
class to_lower
{ public:
    char operator() (char c) const
     { return isupper (c)? tolower(c) : c;
     }
};
```

zdefiniowana jest funkcja operatorowa **operator().** W tej klasie realizowane jest zadanie zamiany dużej litery na małą.

Funkcja **zamiana()**:

```
string zmiana (const string &str)
{ string st;
  transform ( str.begin(), str.end(),
              back_inserter(st),
                to_lower());
  return st;
}
```

wykorzystuje algorytm **transform()** do zamiany łańcucha, wykorzystuje funkcję **to_lower(),** która została przekształcona w obiekt funkcyjny.

W kolejnym programie pokazujemy zastosowanie obiektu funkcyjnego do obsługi algorytmu **for_each().** Bardzo często się zdarza, że wykonujemy operacje na elementach jakiegoś kontenera, a następnie chcemy mieć wydrukowane wyniki. W pokazanym programie tworzymy dwa wektory, pierwszy z nich inicjalizowany jest wartościami typy **int**, drugi z nich przechowuje łańcuchy. Dzięki obiektowi funkcyjnemu, mamy zapewniony wydruk wartości, niezależnie od typu danych. Pokazany przykład dobitnie pokazuje, że obiekty funkcyjne są eleganckie i w zastosowaniach bardzo wydajne.

Wydruk 6.9. STL – własny unarny obiekt funkcyjny, wydruk danych

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<conio.h>
using namespace std;
template<class T>
struct drukuj
{ drukuj (ostream &out): os(out), licz(0) {}
  void operator() (T x) { os << x << " "; ++licz; }
  ostream &os;
  int licz;
};
int main()
{ vector<int> v(10);
  for(unsigned i=0; i < v.size(); i++)
    v[i] = rand() % 100;
  drukuj<int> a1 = for_each (v.begin(), v.end(), drukuj<int>(cout));
  cout << endl << a1.licz << " elementow drukowanych" << endl;
  vector <string> vs;
    vs.push_back("Jacek");
    vs.push_back("Helena");
    vs.push_back("Ewa");
    vs.push_back("Edmund");
    vs.push_back("Hipolit");
  drukuj<string> a2 = for_each(vs.begin(),vs.end(),drukuj<string>(cout));
  cout << endl << a2.licz << " elementow drukowanych" <<endl;
  getche();
  return 0;
     }
```

Wynik działania programu ma postać:

```
30  82  90  56  17  95  15  48  26  4
10 elementow drukowanych
Jacek  Helena  Ewa  Edmund Hipolit
5 elementow drukowanych
```

W instrukcjach:

```
template<class T>
struct drukuj
{ drukuj (ostream &out): os(out), licz(0) {}
  void operator() (T x) { os << x << " "; ++licz; }
  ostream &os;
```

```
    int licz;
  };
```

definiujemy klasę **drukuj**, w której jest definiowana przeciążona funkcja **operator()** z jednym argumentem. Klasa **drukuj** jest używana do definiowania obiektów funkcji, dla których przeciążone funkcje **operator()** wykonują takie same zadania. Jest to zlecenie drukowania danych.
W instrukcjach:

```
drukuj<int> a1 = for_each (v.begin(), v.end(), drukuj<int>(cout));
drukuj<string> a2 = for_each(vs.begin(),vs.end(),drukuj<string>(cout));
```

tworzone są obiekty funkcyjne **a1** i **a2**, a algorytm **for_each()** wykonuje takie operacje na każdym elemencie kolekcji, jakie zleca trzeci argument, w naszym przypadku jest to zlecenie drukowania wartości każdego elementu.

6.4. Zastosowania obiektów funkcyjnych binarnych

Sortowanie łańcuchów jest popularną usługą w wielu aplikacjach. Często podczas wprowadzania danych, następują pomyłki, na przykład jakieś nazwisko pisane jest małą literą. W języku C++ małe litery są innymi znakami niż duże, wobec czego sortowanie nazwisk pisanych małymi i dużymi literami może prowadzić do nieprawidłowego wyniku. W kolejnym programie zademonstrujemy wykorzystanie predykatu dwuargumentowego, dzięki któremu sortowanie nazwisk, bez względu na wielkość znaków przebiegnie prawidłowo.

Wydruk 6.10. STL – własny binarny predykat, sortowanie nazwisk

```
#include<iostream>
#include<set>
#include<algorithm>
#include<string>
#include<iterator>
#include<conio.h>

using namespace std;

class sort_s
{
 public:
  bool operator() (string& s1, string& s2)
   {string s1_m, s2_m;
    s1_m.resize(s1.size());
    transform (s1.begin(), s1.end(),s1_m.begin(), tolower);
    s2_m.resize(s2.size());
    transform (s2.begin(), s2.end(),s2_m.begin(), tolower);
```

```
    return (s1_m < s2_m);
   }
};

int main()
{ typedef set<string, sort_s> NZ;
  ostream_iterator<string> output(cout, "  ");
  NZ nazwiska;
  nazwiska.insert("kowalski");
  nazwiska.insert("Kwiatek");
  nazwiska.insert("bielska");
  nazwiska.insert("Palitor");
  nazwiska.insert("Anielski");
  copy(nazwiska.begin(),nazwiska.end(),output);
  getche();
  return 0;
}
```

Po uruchomieniu programu otrzymujemy następujący wynik:

Anielska bielska kowalski Kwiatek Palitor

W programie umieszczamy klasę **sort_s**. Predykat binarny implementowany w funkcji operatorowej pobiera dwa łańcuchy: **s1** i **s2**. Przy pomocy algorytmu **transform()** zamienia znaki wejściowe na małe znaki. Algorytm **transform()** (w przykładzie korzystamy z jego czteroargumentowej postaci), wymaga unarnej funkcji. W naszym przykładzie korzystamy ze znanej funkcji **tolower(c),** która zwraca małą literę odpowiadającą znakowi **c**, w przypadku, gdy taka litera istnieje.
W instrukcjach:

```
s1_m.resize(s1.size());
transform (s1.begin(), s1.end(),s1_m.begin(), tolower);
```

metoda **resize()** zapewnia odpowiednią ilość pamięci na przechowywanie napisu. Formalnie **resize(num)** zmienia liczbę elementów na **num**, (jeśli wzrasta wartość funkcji **size()**). W naszym przykładzie algorytm **transform()** przekształca łańcuch **s1** (a także **s2**) na tekst pisany małymi literami.
Dopiero w poleceniu:

```
return (s1_m < s2_m);
```

dzięki przeciążonemu operatorowi "<" porównujemy dwa łańcuchy.
Listę nazwisk przechowujemy w kontenerze skojarzeniowym **<set>.** Ten kontener doskonale się nadaje do szybkiego wyszukiwania przechowywanych w nim kluczy. Kontener **set** przechowuje unikalne klucze. Implementacja **set**

podobna jest do struktury drzewa binarnego. Elementy wstawiane do obiektów **set** są sortowane w kolejności ich wstawiania, co znacznie może przyśpieszyć operacje wyszukiwania. Domyślnie sortowanie przebiega przy pomocy unarnego predykatu:

std:: less<T>

Jeżeli chcemy zmienić sposób sortowania, musimy dostarczyć inny predefiniowany predykat lub utworzyć własny i ustanowić obiekt **set** wraz z tym nowym predykatem sortowania:

set<Object_Type, OptionalBinaryPredicate>

W naszym przykładzie korzystamy z własnego predykatu **sort_s**:

typedef set<string, sort_s> NZ;

Dzięki naszemu predykatowi, sortowanie nazwisk odbywa się poprawnie bez względu na wielkość liter. Przy pomocy metody **insert():**

nazwiska.insert("kowalski");

wstawiamy do kontenera nazwiska.

Przy pomocy zdefiniowanego iteratora wyjściowego (należy włączyć plik **<iterator>**) oraz algorytmu **copy()** wyświetlamy posortowaną listę nazwisk.

```
ostream_iterator<string> output(cout, "  ");
                    copy(nazwiska.begin(),nazwiska.end(),output);
```

Prawdziwa zaleta pokazanej techniki obsługi nazwisk zapisanych w kontenerze **<set>** ujawnia się, gdy chcemy wyszukać interesujące nas nazwisko. Kontener **<set>** używany jest do szybkiego zapisywania i odszukiwania elementów. Stosowany do tych celów ma zdecydowaną przewagę w porównaniu do innych kontenerów.

W kolejnym przykładzie prezentujemy algorytm numeryczny **accumulate(),** który wykonuje operacje sumowania. Rozszerzona czteroargumentowa wersja tego algorytmu realizuje sumowanie zgodne z operacjami zdefiniowanymi przez czwarty argument, którym jest obiekt funkcyjny.

Wydruk 6.11. STL – własna binarna funkcja, algorytm **accumulate()**

```
#include<iostream>
#include<vector>
#include<numeric>
#include<iterator>
using namespace std;
template <class T>
```

```
class suma_v : public binary_function <T, T, T>
{ public:
   T operator() ( T suma,  T val)
    { return suma + val*val;
    }
 };

template <class T>
class suma_vv : public binary_function <T, T, T>
{ public:
   T operator() ( T suma,  T val)
    { return suma + val*val*val;
    }
 };
int main()
{ int wynik = 0;
  vector<int> v;
  for(int i = 0; i < 6; ++i)
    v.push_back(i) ;
  ostream_iterator<int> output(cout, "  ");
  copy(v.begin(),v.end(),output);
  wynik = accumulate(v.begin(), v.end(),0);
  cout << "\nwynik, suma pojedyncza= " << wynik;
  wynik = accumulate(v.begin(), v.end(),0, suma_v< int >());
  cout << "\nwynik, suma kwadratow = " << wynik;
  wynik = accumulate(v.begin(), v.end(),0, suma_vv < int >());
  cout << "\nwynik, suma szescianow= " << wynik;
  return 0;
}
```

Program generuje następujący komunikat:

```
0   1   2   3   4   5
wynik, suma pojedyncza= 15
wynik, suma kwadratow =  55
wynik, suma szescianow= 225
```

W programie mamy dwa wzorce klasy, w których zdefiniowano przeciążony **operator().** Na tej podstawie zastosowany algorytm numeryczny **accumulate()** wykonuje zadaną operację korzystając z obiektu funkcyjnego.
W instrukcjach:

```
template <class T>
class suma_vv : public binary_function <T, T, T>
{ public:
```

```
  T operator() ( T suma,  T val)
    { return suma + val*val*val;
    }
};
```

zdefiniowana jest klasa dziedzicząca z klasy **binary_function().** Ta klasa bazowa, znajdująca się w pliku **<functional>** definiuje przeciążony **operator ().**

Obiekt funkcyjny tworzony na podstawie klasy **suma_vv** (a także klasy **suma_v**) będzie wymagał dwóch argumentów.
W instrukcji:

```
wynik = accumulate(v.begin(), v.end(),0);
```

wywoływany jest algorytm **accumulate()** z trzema argumentami. Dwa pierwsze określają zakres przeglądania elementów kontenera, trzeci przechowuje wartość zwracaną, czyli sumę elementów. W tej wersji, algorytm nie potrzebuje obiektu funkcyjnego. Jeżeli chcemy wykonać inny rodzaj sumowania, tak jak w naszym przykładzie, gdzie chcemy sumować kwadraty i sześciany elementów, możemy zadanie to przekazać do obiektu funkcyjnego i wywołać czteroargumentową wersję algorytmu **accumulate():**

```
wynik = accumulate(v.begin(), v.end(),0, suma_v< int >());
```

Ta postać wywołania może być zastąpiona bardziej czytelną konstrukcją:

```
suma_vv<int> obiekt_vv;              //definiuje obiekt klasy suma_vv
wynik = accumulate(v.begin(), v.end(),0, obiekt_vv);
```

ROZDZIAŁ 7

ŁAŃCUCHY I KLASA STRING

7.1. Wstęp..... 188
7.2. Łańcuchy znakowe w stylu języka C 188
7.3. Łańcuchy w stylu języka C++..... 192
7.4. Operacje i metody klasy string..... 197
7.5. Kontenery i klasa string..... 203

7.1. Wstęp

Język C nie posiada wbudowanego odrębnego typu do obsługi napisów (łańcuchów znakowych), takiego, jakim mógłby być na przykład typ **string.** Podobnie język C++ także nie posiada wbudowanego typu **string**. Język C i C++ posługują się typem znakowym **char**, a napis (łańcuch) to grupa znaków. W języku C++ napis jest tablicą znaków. Tablica znaków zakończona jest znacznikiem końca łańcucha (znak zerowy, ang. *null characters*, w kodzie ASCII jest to znak '\0'). Obsługa napisu odbywa wykorzystując wskaźnik do pierwszego znaku. W C++ napis jest stałym wskaźnikiem. Oznacza to, że napisy obsługiwane są tak, jak zwykle tablice. Główna trudność w wydajnym posługiwaniu się łańcuchami znakowymi polega według K. Schildta na fakcie, że na łańcuchach zakończonych znakiem **null** nie można przeprowadzić żadnych operacji posługując się standardowymi operatorami C i C++. W czasach tworzenia języka C, przetwarzanie tekstów nie było ważne, ale w miarę rozwoju technik komputerowych, obsługa napisów stała się pierwszoplanowym zagadnieniem. Dostrzegli to twórcy języka C++ i zaproponowali całkowicie nowy sposób obsługi napisów – opracowano standardową klasę **string**, dzięki której programista ***praktycznie*** otrzymał do dyspozycji typ **string**. Ponieważ, jak zwykle obowiązuje zasada kompatybilności, w języku C++ mamy dwie metody obsługi napisów:

- napisy w stylu języka C
- napisy w stylu języka C++

Posługiwanie się napisami w stylu języka C wymaga włączenia pliku nagłówkowego <string.h>, posługiwanie się napisami w stylu języka C++ wymaga pliku <string>. Możliwe jest umieszczenie w programie napisów i ich obsługa zarówno w stylu C i stylu C++, ale nie jest to polecane.

7.2. Łańcuchy znakowe w stylu języka C

W programach literały napisowe (ang. *string literals*) lub stałe napisowe (ang. *string constans*) są specjalnie oznaczane, tak jak w tym przykładzie:

```
"program"
```

Jest to ciąg znaków zapisanych w podwójnych apostrofach. Deklarowanie napisu w języku C/C++ może mieć jedną z następujących postaci:

```
char n1[ 8 ] = "program" ;
char n2[   ] = "program" ;
char n3[ ] = {'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
char * n4   = "program";
```

W języku C++ napis może być wprowadzany ze strumienia wejściowego (na przykład z klawiatury) i przypisany tablicy przy pomocy **cin**, a gdy należy wprowadzić do tablicy tekst, w języku C++ możemy posłużyć się funkcją **cin.getline().** Pokazany program odczytuje wprowadzone z klawiatury słowo i liczy ilość liter (określa długość słowa). Jest to typowe użycie **cin** do obsługi napisu.

Wydruk 7.1. Przykład użycia **cin** do odczytywania napisu

```
#include<iostream>
#include <conio.h>
using namespace std;

int main()
{
  int k= 0;
 char s1[80];
 cout << "wpisz slowo: ";
 cin>> s1;
 while(s1[k++])
  ;
 cout << "dlugosc slowa = " << k-1<< endl;

 getche();
 return 0;
}
```

Wynik działania programu ma postać:

wpisz slowo: program
długość slowa = 7

Metoda określania długości słowa jest prosta – wiemy, że napis zakończony jest znakiem **null**. W pętli **while** przeglądane są znaki, do momentu, gdy nie natrafimy na wartość zero (jest to wartość **null**). W tym momencie pętla przerwie działanie a my otrzymujemy długość słowa.

Sprawy się komplikują, gdy chcemy wprowadzić więcej niż jedno słowo. Jeżeli zechcemy wprowadzić więcej niż jedno słowo, nie osiągniemy tego z obiektem **cin**. Obiekt **cin** traktuje białe spacje, jako separator. Gdy zidentyfikuje białą spację, zinterpretuje to, jako koniec wprowadzania napisu i doda na jego końcu znak **null**. Rozwiązaniem jest wykorzystanie metody **get(),** co jest pokazane w kolejnym programie.

Wydruk 7.2. Przykład użycia **cin.get()** do odczytywania napisu

```
#include<iostream>
#include <stdio.h>
#include <conio.h>
#include <string.h>

using namespace std;
int main()
{ char s1[80];
  char s2[80];
  cout << "wpisz powitanie: ";
  cin.get(s1,79);
  cout << "wprowadzono: "<< s1<<endl;
  cin.ignore(255,'\n');
  cout << "wpisz tekst: ";
  cin.get(s2,79);
  cout << "wprowadzono: "<< s2<<endl;

  getche();
  return 0;
}
```

Wynik działania programu jest następujący:

```
wpisz powitanie: to jest
wprowadzono: to jest
wpisz tekst: nasze programowanie
wprowadzono: nasze programowanie
```

Formalnie prototyp funkcji **get()** ma postać:

```
get( pCharArray,  StreamSize, TermChar);
```

Parametr **pCharArray** jest wskaźnikiem do tablicy znaków, **StreamSize** jest maksymalną ilością znaków do odczytania plus jeden, parametr **TermChar** jest znakiem kończącym. Należy zwrócić uwagę na instrukcję:

```
cin.ignore(255,'\n');
```

Gdyby nie było instrukcji **ignore(),** to sesja miałaby postać:

```
Wpisz powitanie: to jest
Wprowadzono: to jest
Wpisz tekst: wprowadzono:
```

co wskazuje na błędną obsługę napisów. Problem jest z funkcją **get().** Ta funkcja odczytuje znak końca linii, ale go nie odrzuca, znak pozostaje w buforze. Aby zaradzić naszemu problemowi korzystamy z funkcji **ignore()**. Ta funkcja przyjmuje dwa parametry: pierwszy określa maksymalną ilość znaków do pominięcia, drugi określa znak końcowy. W naszym zapisie funkcja **ignore()** odrzuci do 255 znaków oraz znajduje znak końca łańcucha **\0** i też go odrzuca.

Tego typu niedogodności w obsłudze napisów w stylu języka C mamy wiele. Kolejny przykład ilustruje problemy związane ze stosowaniem funkcji operujących na napisach.

Wydruk 7.3. Przykład użycia **cin.getlin(), strcpy(), strcat()**

```
#include<iostream>
#include <stdio.h>
#include <conio.h>
#include <string.h>

using namespace std;

int main()
{
 char text[200];
 char s1[80];
 char s2[80];
 cout << "wpisz tekst: ";
 cin.getline(s1,79);
 cout << "wpisz tekst: ";
 cin.getline(s2,79) ;

 strcpy(text,"\n ");
 strcat(text,s1);
 strcat(text,s2);
 cout << text << endl;

 getche();
 return 0;
}
```

Wynikiem działania programu jest komunikat:

```
wpisz tekst: to jest
wpisz tekst: programowanie

to jest programowanie
```

Jeżeli w programie nie będzie instrukcji:

```
strcpy(text,"\n ");
```

to otrzymamy wynik:

```
wpisz tekst: to jest
wpisz tekst: programowanie
debugHoo@to jest programowanie
```

Ewidentnie, ponownie wystąpił problem z obsługą napisów. W naszym programie wprowadzamy napisy z klawiatury (tablice **s1** i **s2**) a następnie przy pomocy funkcji **strcat()** łączymy te dwa napisy. Pierwsza wywoływana w tym celu jest funkcja **strcpy(),** która kopiuje stałą tekstową na początek zmiennej **tekst**. Gdybyśmy, jako pierwszą wywołali funkcje **strcat(),** wystąpiłby problem (zależny od typu kompilatora). Umieszony w tej zmiennej znak przejścia do nowej linii **\n** rozwiązuje problem.

Takich problemów, jaki tu zasygnalizowaliśmy jest znacznie więcej. Opracowanie nowej koncepcji obsługi napisów w języku C++, jakim było wprowadzenie klasy **string**, zdecydowanie uprościło programowanie z udziałem łańcuchów znakowych.

7.3. Łańcuchy w stylu języka C++

Twórcy języka C++ zdając sobie sprawę z niedoskonałości obsługi tablic znakowych w języku C, zaproponowali nową koncepcję przetwarzania łańcuchów – stworzona została nowa klasa, klasa o nazwie **string.** Pamiętamy, że klasa jest typem zdefiniowanym w programie lub bibliotece, co oznacza, że praktycznie dysponujemy nowym typem, typem **string**. Oczywiście w programie możemy wprowadzać i wykonywać żądane operacje na łańcuchach w stylu C a także na łańcuchach w stylu C++. W celu programowania z użyciem łańcuchów w stylu C++ należy włączyć plik nagłówkowy <string>. Plik nagłówkowy o nazwie <string.h> umożliwia przetwarzanie łańcuchów „starego typu", to znaczy łańcuchów w stylu C.
Zmienne typu **string** można deklarować w zwykły sposób:

```
string a,b;
```

Jest to deklaracje dwóch zmiennych typu **string**. Gdy już zostały utworzone zmienne typu **string** należy je zainicjalizować. Można to zrobić na kilka sposobów:

```
string a,b;
a = ”to jest”;
b = ”programowanie;
```

Można połączyć deklarację z inicjalizacją:

string a = ”to jest”

Ponieważ mamy do czynienia z klasą, obiekt **string** można inicjalizować konstruktorem:

string a (”to jest”);

Klasa **string** ma wiele metod i przeciążonych operatorów, co pozwala na stosunkowo łatwe manipulowanie napisami w stylu języka C++. Podstawowe operacje wykonywane na napisach ilustruje kolejny program.

Wydruk 7.4. Łańcuchy w stylu języka C++

```
#include<iostream>
#include <conio.h>
#include <string>

using namespace std;

int main()
{ string s1 ( " - to jest ");
  string s2,s3,s4;
  s2 = "programowanie!";
  cout <<"s1 = " << s1 << endl;
  cout <<"s2 = " << s2 << endl;
  cout << "konkatenacja: " ;
  s3 = s1+s2;
  cout << s3<<endl;
  cout << " napisz nazwisko : " ;
  getline(cin, s4);
  cout << "Panie " << s4 +s3 << endl;
  string s5 = "lis";
  cout << "    s5 : " << s5;
  s5[1] = 'o';
  cout << "\nnowy s5 : ";
  cout << s5;
  string s6 = " - to jest proste ";
  s1.assign(s6);
  cout << "\nmetoda assign(): " << s1+s2;

  getche();
  return 0;
}
```

W wyniku działania programu mamy następujący komunikat:

```
s1 =  - to jest
s2 = programowanie!
konkatenacja:  to jest programowanie!
 napisz nazwisko : Kowalski
Panie Kowalski – to jest programowanie!
s5 : lis
 nowy s5 : los
metoda assign(): - to jest proste programowanie!
```

Gdy mamy zainicjalizowane łańcuchy **s1** i **s2** możemy je połączyć przy pomocy przeciążonego operatora ” +” :

```
s3 = s1+s2;
```

Funkcja **getline()** umożliwia odczytanie z wejścia całej linii tekstu i umieszczeniu go w zmiennej napisowej:

```
cout << " napisz nazwisko : " ;
getline(cin, s4);
```

Do elementów łańcucha możemy odwołać się także przy pomocy operatora indeksu:

```
 string s5 = "lis";
cout << "     s5 : " << s5;
s5[1] = 'o';
cout << "\nnowy s5 : ";
cout << s5;
```

W napisie “**lis**” została zmieniona druga litera po odwołaniu się do tego elementu przez indeks i po podstawieniu nowej wartości.
W instrukcjach:

```
string s6 = " - to jest proste ";
s1.assign(s6);
cout << "\nmetoda assign(): " << s1+s2;
```

utworzony został nowy łańcuch **s6** i wywołana została funkcja składowa klasy **string assign().** Ta funkcja służy do zamiany jednego łańcucha na inny. W naszym przypadku łańcuch **s1** zostaje zamieniony na łańcuch **s6**, w efekcie napis ”**- to jest**” zostaje zamieniony na ” **– to jest proste**” i ta nowa postać jest wykorzystana do połączenia z napisem **s2**.

Klasa **string** zawiera dość dużą ilość metod pozwalających, albo na dokonywanie ciekawych operacji na napisach, albo na otrzymywaniu istotnych informacji o łańcuchu. W kolejnym programie wykorzystamy kilka takich metod.

Wydruk 7.5. Łańcuchy w stylu języka C++, metody klasy **string**

```
#include<iostream>
#include <conio.h>
#include <string>
using namespace std;

void info(string &);

int main()
{
 string s1 ( " Matejko ");
 string s2 = " wielkim malarzem jest!";
 string s3 = "Jan";
 string s4,s5;
 cout << "konkatenacja: " ;
 s4 = s3+s1+s2;
 cout << s4<<endl;
 for (int i= 0; i<s3.size();i++) cout << s3[i]<< endl;
 cout << "nazwisko ma " << s1.size() << " liter" << endl;
 cout << "podaj imie Matejki: " ;
 getline(cin, s5);
 if (s3 == s5)
    cout << "poprawne imie" << endl;
 else
    cout<< "nieporawne imie" << endl;
 cout << s1.erase(4,2)<< endl;;
 cout << s1.replace(0,1,"0 ");
 info(s4);

 getche();
 return 0;
}

void info ( string &s)
{cout << " \nnapis: " <<s << endl;
 cout << " pojemnosc: "<<s.capacity()<< endl;
 cout << "   rozmiar: "<<s.size()<< endl;
 cout << "   dlugosc: "<<s.length()<< endl;
 cout << "     pusty: "<<(s.empty() ? "tak" : "nie");
}
```

Po uruchomieniu pokazanego programu otrzymujemy następujący komunikat:

```
konkatenacja: Jan Matejko wielkim malarzem jest!
J
a
n
nazwisko ma 9 liter
podaj imie Matejki: Ewa
niepoprawne imie
     Matko
  o Matko
napis : Jan Matejko wielkim malarzem jest!
         pojemność: 35
            rozmiar: 35
            długość: 35
               pusty: nie
```

Po inicjalizacji łańcuchów **s1,s2** i **s3** mamy możliwość testowania metod klasy **string**. Ponieważ mamy do czynienia z klasą, to wywołania funkcji mają postać **obiekt.funkcja**. Tego typy wywołanie jest użyte w instrukcji:

```
for (int i= 0; i<s3.size();i++) cout << s3[i]<< endl;
cout << "nazwisko ma " << s1.size() << " liter" << endl;
```

Funkcja **size()** wywołana na rzecz obiektu **s3** zwraca rzeczywistą liczbę znaków w łańcuchu znakowym (identycznie działa inna metoda klasy **string** – **lenght()**). W pokazanych instrukcjach, korzystając z operatora indeksu w pętli **for** realizowane jest kolejne wyświetlanie znaków tworzących łańcuch **s3** oraz wyświetlana jest ilości znaków. Zwracamy uwagę, że otrzymaliśmy liczbę znaków równą 9, spowodowane jest to faktem, że w badanym łańcuchu mamy także dwie spacje.
W kolejnym fragmencie programu:

```
cout << "podaj imie Matejki: " ;
getline(cin, s5);
if (s3 == s5)
        cout << "poprawne imie" << endl;
else
        cout<< "nieporawne imie" << endl;
```

należy podać imię malarza (obsługę łańcucha kontroluje funkcja **getline(cin,s5),** a wyrażenie **s3==s5** sprawdza **s3** i **s5** pod kątem równości. Mamy tu do czynienia z przeciążonym operatorem równości, wszystkie przeciążone funkcje operatorów zwracają wartości logiczne **bool**.

Wśród dostępnych metod mamy takie, które pozwalają na manipulowanie znakami w łańcuchu. W instrukcjach:

```
cout << s1.erase(4,2)<< endl;;
cout << s1.replace(0,1,"o ");
```

wykorzystano dwie funkcje: **erase()** oraz **replace()** do wykonania zmian w łańcuchu. Funkcja **erase()** usuwa znaki, a funkcja **replace()** zastępuje znaki. W naszym łańcuchu **s1**, który zawiera napis ” **Matejko** ” funkcja, od pozycji czwartej usuwa dwa znaki i otrzymujemy nowy napis ” **Matko** ”, który jest nadpisywany w zmiennej **s1**. Z kolei zastosowanie funkcji **replace()** do nowego napisu **s1** powoduje, że na samym początku napisu spacja zostaje zastąpiona znakiem ‘**o**’, w wyniku, czego mamy nowy napis ” **o Matko** ”. Na marginesie jest to bardzo często spotykane westchnienie pracowitych studentów na sprawdzianach z podstaw programowania w języku C++.

Występująca w programie funkcja **info()** służy do dostarczenia informacji o rozmiarze, pojemności, długości i innych cechach charakterystycznych łańcucha:

```
void info ( string &s)
{ cout << " \nnapis: " <<s << endl;
  cout << " pojemnosc: "<<s.capacity()<< endl;
  cout << "   rozmiar: "<<s.size()<< endl;
  cout << "   dlugosc: "<<s.length()<< endl;
  cout << "     pusty: "<<(s.empty() ? "tak" : "nie");
}
```

Metoda **capacity()** zwraca liczbę znaków, która może być umieszczona w łańcuchu, bez ponownego przydziału pamięci. Wiemy, że nie musimy z góry określać ilości pamięci potrzebnej do przechowywania łańcucha, system przydziela odpowiednią ilość pamięci w miarę potrzeb. Metoda **empty()** zwraca wartość określającą, czy dany łańcuch jest pusty.

7.4. Operacje i metody klasy string

Klasa **string** jest bardzo rozbudowana, jej opis techniczny jest dość skomplikowany. Na przykład, w systemie pomocy kompilatora Builder C++ wersji szóstej, firmy Borland, opis techniczny zajmuje około 30 stron. W naszym skrypcie zasygnalizujemy najważniejsze (wybrane dość arbitralnie) główne operacje i przydatne metody. Czytelnik chcący zapoznać się z kompletnym opisem klasy **string** powinien przestudiować podręcznik N. Josuttisa „C++. Biblioteka standardowa. Podręcznik programisty”.

Obsługa łańcuchów w stylu C++ wymaga włączenia pliku nagłówkowego <string>. Plik nagłówkowy <string> zawiera definicje klasy wzorca dla wszystkich typów łańcuchowych, ta klasa nosi nazwę **basic_string<>.**

W STL mamy dwie wyspecjalizowane wersje klasy **basic_string<>:**

- klasa string do obsługi typów char
- klasa wstring do obsługi typu wchar_t (poszerzony zestaw znaków)

Klasa **string** ma wiele konstruktorów i jeden destruktor. Lista konstruktorów przedstawiona jest w tabeli 7.1

Tabela 7.1. Konstruktory klasy **string**

konstruktor	działanie
string s	Tworzy pusty łańcuch znakowy
string s(s1)	Łańcuch **s** jest kopią łańcucha **s1**
string s(s1,n)	Łańcuch **s** zawiera znaki **s1** począwszy od indeksu **n**
string s(s1,n,d)	Łańcuch **s** zawiera **d** znaków z łańcucha **s1**, umieszczonych od pozycji **n**
string s(cs1)	Tworzy łańcuch w stylu języka C
string s(c1,d)	Tworzy łańcuch **s**, który zawiera **d** znaków tablicy znakowej **c1**
string s(b,e)	Tworzy łańcuch znakowy z zakresu [b,e]

Obsługując łańcuchy do dyspozycji mamy następujące iteratory klasy **string**:

- begin()
- begin() const
- end()
- end() const
- rbegin()
- rbegin() const
- rend()
- rend() const

Zestaw metod klasy **string** przedstawia tabela 7.2

Tabela 7.2. Metody klasy **string**

metoda	działanie
append()	6 wersji, dołącza do istniejącego łańcucha znaki
assign()	6 wersji, zastępuje łańcuch nowymi wartościami
at()	2 wersje, umożliwia dostęp do znaku
capacity()	Podaje ilość znaków, które mogą być umieszczone w łańcuchu bez ponownego przydziału pamięci
compare()	6 wersji, porównuje dwa łańcuchy
copy()	Tworzy łańcuch znakowy
data()	Zwraca wskaźnik do pierwszego elementu tablicy utworzonej z wartości łańcucha
empty()	Informuje, czy łańcuch jest pusty
erase()	3 wersje, usuwa znaki w łańcuchu
find()	3 wersje, wyszukuje wystąpienia znaku w łańcuchu
find_first_not_of()	4 wersje, poszukuje pierwszego znaku niebędącego w łańcuchu
find_first_of()	4 wersje, poszukuje pierwszego znaku będącego w łańcuchu
find_last_not_of()	4 wersje, poszukuje ostatniego znaku niebędącego w łańcuchu
insert()	5 wersji, wstawia dodatkowy element do łańcucha
length()	Zwraca liczbę elementów w łańcuchu
max_size()	Zwraca maksymalny rozmiar łańcucha
rfind()	3 wersje, szuka ostatniej sekwencji znaków w łańcuchu
replace()	5 wersji, wybrane znaki sekwencji zastępuje innymi znakami

reserve()	Rezerwuje pamięć na określoną ilość elementów
resize()	Zmienia rozmiar łańcucha
size()	Zwraca liczbę elementów w łańcuchu
substr()	Zwraca ustalony fragment łańcucha
swap()	Wymienia elementy pomiędzy dwoma łańcuchami

Większość pokazanych metod może przyjmować różne kombinacje argumentów, ich działanie należy sprawdzać w opisie technicznym biblioteki. W kolejnym programie pokazujemy kilka wariantów działania metody **find().**

Wydruk 7.6. wyszukiwanie znaków w łańcuchu

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;

int main()
{ size_t n1;
 string s1 = "quicksort is usually implemented recursively" ;
 cout << " 'usually' na pozycji "<< s1.find("usually") << endl;
 cout << "k jest na pozycji "<< s1.find('k') << endl;
 cout << "i jest na pozycji "<< s1.find('i', 20) << endl;
   const char znak1 = 's';
   cout << "wszystkie pozycje znaku "<< znak1 << endl;
   n1=s1.find(znak1,0);
   while(n1 != string :: npos)
      {cout << znak1 << " na pozycji " << n1 << endl;
       n1 = n1+1;
       n1 = s1.find(znak1, n1);
      }
 const char znak2 = 'b';
 n1 = s1.find(znak2,0);
 if (n1 != string:: npos)
   cout << "jest znak na pozycji " << n1 << endl;
 else
   cout << " nie ma znaku "<< znak2;

 getche();
 return 0;
}
```

Po uruchomieniu program mamy następujący wynik:

```
  ‘usually’ na pozycji 13
k jest na pozycji 4
i  jest na pozycji 21
wszystkie pozycje znaku s
s na pozycji 5
s na pozycji 11
s na pozycji 14
s na pozycji 38
  nie ma znaku b
```

Należy pamiętać, że pierwszy znak w łańcuchu ma pozycję 0.
W instrukcjach:

```
string s1 = "quicksort is usually implemented recursively" ;
cout << " 'usually' na pozycji "<< s1.find("usually") << endl;
cout << "k jest na pozycji "<< s1.find('k') << endl;
cout << "i jest na pozycji "<< s1.find('i', 20) << endl;
```

mamy zdefiniowany łańcuch **s1** oraz pokazane przypadki użycia (różne wersje) metody **find().** Warianty działania metody **find()** opisano w tabeli 7.3

Tabela 7.3. Wersje metody **find()** klasy **string**

metoda	działanie
s1.find("usually")	Szuka pierwszego wystąpienia łańcucha ”**usually**” i zwraca pozycję jego początku w łańcuchu **s1**
s1.find('k')	Szuka pierwszego wystąpienia znaku ‘**k**’ i zwraca jego pozycję w łańcuchu **s1**
s1.find('i', 20)	Szuka pierwszego wystąpienia znaku ‘**i**’ i zwraca jego pozycję w łańcuchu **s1**, przeszukiwanie zaczyna od pozycji 20

W instrukcjach:

```
const char znak1 = 's';
cout << "wszystkie pozycje znaku "<< znak1 << endl;
n1=s1.find(znak1,0);
while(n1 != string :: npos)
    {cout << znak1 << " na pozycji " << n1 << endl;
     n1 = n1+1;
     n1 = s1.find(znak1, n1);
    }
```

realizowane jest wyszukiwanie wszystkich pozycji znaku „**s**” w badanym łańcuchu **s1**. W klasie **string** znajduje się definicja typu **string::size_type**. Jest to typ całkowitoliczbowy bez znaku, określający rozmiar wartości oraz indeksów. Ten typ jest silnie zalecany w obsłudze łańcuchów. Typ **size_t** jest równoważny typowi **size_type** (dla łańcuchów).
Występująca na początku programu deklaracja

```
size_t  n1;
```

deklaruje zmienną **n1**, jako typ **size_t**, w zmiennej **n1** przechowywane będą pozycje znaków w łańcuchu. W pętli **while** występuje wartość **string::npos**. W przypadku, gdy poszukiwanie określonego znaku w łańcuchu kończy się niepowodzeniem (nie ma szukanego znaku) metody **string** zwracają wartość **npos**. Należy zachować ostrożność, ponieważ, projektanci biblioteki nadali **npos** wartość -1. Z drugiej strony wartość **size_t** musi być typu całkowitoliczbowego bez znaku i **npos** musi być konwertowana. Dlatego zaleca się, aby zawsze stosować **string::size_t** a nie na przykład **int** lub **unsigned int**, gdy operujemy pozycjami znaków w łańcuchach.

Zwrócenie wartości **npos** sygnalizuje, że nie ma szukanego znaku w łańcuchu, obsługa tego zadania realizowana jest kolejnym fragmencie programu:

```
const char znak2 = 'b';
n1 = s1.find(znak2,0);
if (n1 != string:: npos)
  cout << "jest znak na pozycji " << n1 << endl;
else
  cout << " nie ma znaku "<< znak2;
```

Podczas przetwarzania łańcuchów możemy szeroko wykorzystywać przeciążone operatory klasy **string**. Lista takich operatorów pokazana jest w tabeli 7.4.

Tabela 7.4. Przeciążone operatory klasy **string**

operator	działanie
<, >, <=, >=, ==, !=	Porównywanie łańcuchów
=	Przypisanie
+, +=	Łączenie łańcuchów (konkatenacja)
[ ]	Umożliwia dostęp do znaku (indeksacja)
<<	Wyjście na strumień klasy **ostream**
>>	Wejście ze strumienia klasy **istream**

Operacje wejścia dla łańcuchów można także realizować przy pomocy

specjalnej funkcji zdefiniowanej w przestrzeni nazw **std**. Jest to funkcja **std::getline().** Służy ona do odczytywania danych wiersz po wierszu i umieszczania ich w łańcuchu. Mamy dwie wersje funkcji **getline():**

```
istream& getline(steram & we, string& s)
istream& getline(steram & we, string& s, char zn)
```

Przykłady użycia tych wersji są następujące:

```
string s1,s2;
getline(cin, s1);
getline(cin, s2, ‘ : ’)
```

Druga wersja (z trzema argumentami) funkcji **getline()** wczytuje do łańcucha **s2** znaki ze strumienia wejściowego **cin**, aż do napotkania znaku separatora (u nas jest to dwukropek), osiągnięcia maksymalnego rozmiaru łańcucha lub napotkania znacznika końca pliku. Znak separatora nie jest zachowany. Pierwsza, dwuargumentowa wersja **getline(),** odczyta wejście do końca pliku. Znak nowego wiersza pełni rolę domyślnego separatora.

Dostęp do dowolnego elementu łańcuch można zrealizować na dwa sposoby – przy pomocy przeciążonego operatora **[ ]** lub funkcji składowej **at().** Należy pamiętać, że operator **[ ]** nie sprawdza, czy indeks jest poprawny (tak samo jak w przypadku zwykłych tablic). Zaleca się używania funkcji **at ().** Jeżeli wystąpi przypadek użycia niewłaściwego indeksu, funkcja **at()** generuje wyjątek **out_of_range**.

7.5. Kontenery i klasa string

Zmienna **string** jest sekwencyjnym kontenerem znaków. Formalnie w języku C++ **string** jest definiowany jako konkretyzacja szablonu klasy **basic_string** dla typu **char**. Zmienna **string** najbardziej przypomina kontener **vector**. Obsługując łańcuchy możemy korzystać z metod **begin()** i **end(),** które zwracają iteratory do zmiennej typu string. Możemy korzystać także z metod takich jak **insert(), erase(), size(), empty()** i wiele innych. Oczywiście **string** różni się od kontenera **vector**, nie posiada wszystkich metod tego kontenera. Należy pamiętać, że w literaturze można też spotkać wiele stwierdzeń nieprecyzyjnych, w jednej ze znanych monografii napisano, że string nie posiada insertera **push_back().** Przykład użycia obiektu **string** w charakterze kontenera pokazuje kolejny program, pokazano także użycie metody **push_back().**

Wydruk 7.7. obiekt **string**, jako kontener STL

```
#include <iostream>
#include <string>
#include <conio.h>
```

```
using namespace std;
int main()
{ string :: const_iterator p;
  string s1;
  s1.insert(s1.end(), 'C');
  s1.insert(s1.end(), '+');
  s1.insert(s1.end(), '+');
  s1.insert(s1.end(), ' ');
  s1.push_back('!');
  for (p = s1.begin(); p != s1.end(); ++p)
    cout << *p ;
  getche();
  return 0;
}
```

Wynikiem uruchomienia programu jest komunikat:

C++ !

Instrukcja

```
s1.insert(s1.end(), 'C');
```

wykorzystuje metodę **insert()** do wstawienia znaku '**C**' do łańcucha **s1**. Podobnie działają kolejne instrukcje. Metoda **insert()** korzysta z iteratora **end().** Pokazana jest także metoda **push_back()** do umieszczenia znaku '**!**' na końcu łańcucha **s1**. Instrukcja pętli **for**:

```
for (p = s1.begin(); p != s1.end(); ++p)
    cout << *p ;
```

powoduje wyprowadzenie na ekran monitora naszego napisu. W pętli **for** wykorzystano definicję iteratora:

```
string :: const_iterator p;
```

Iterator stały umożliwia odczytywanie znaków z łańcucha, ale nie pozwala na ich modyfikacje. W pętli inicjowana jest zmienna **p** przez użycie metody **begin(),** która zwraca **const_iterator** do pierwszego elementu łańcucha **s1**. Pętla działa tak długo, aż nie będzie osiągnięty koniec łańcucha, wykorzystano porównanie **p** do wyniku **s1.end().**
Iterator **p** służy do wykonania przejścia po wszystkich elementach łańcucha **s1.** Wartość każdego elementu łańcucha jest wyświetlana przez dereferencję iteratora, podobnie jak to się dzieje dla wskaźnika.

Wektory są bardzo wygodnym kontenerem. Mogą przechowywać różnego typu elementy, w tym także łańcuchy. Poniższy przykład pokazuje wykorzystanie kontenera **vector** do obsługi łańcuchów.

Wydruk 7.8. obiekt **string** i kontener **vector**

```
#include <iostream>
#include <vector>
#include <string>
#include <iterator>   //ostream_iterator
#include <conio.h>
using namespace std;

int main()
{ostream_iterator<string> output (cout, " ");
 vector<string> :: iterator p ;
  vector<string> s1;
  s1.push_back("To");
  s1.push_back(" jest");
  s1.push_back(" programowanie");
  for (p = s1.begin(); p != s1.end(); ++p)
    cout << *p ;
    cout << endl;
  s1.insert(find(s1.begin(),s1.end()," jest"), "takze");
  copy (s1.begin(),s1.end(),output);

 getche();
  return 0;
}
```

Po uruchomieniu programu otrzymujemy następujący komunikat:

```
To jest programowanie
To takze jest programowanie
```

W deklaracjach:

```
vector<string> :: iterator p ;
 vector<string> s1;
```

tworzymy iterator oraz pusty wektor do przechowywania łańcuchów **s1**.

Korzystając z metody **push_back()** dołączamy elementy do sekwencji łańcuchów. W pętli **for**:

```
for (p = s1.begin(); p != s1.end(); ++p)
    cout << *p ;
```

korzystając z dereferencji iteratora wyświetlamy kolejne łańcuchy. W kolejnym fragmencie programu modyfikujemy sekwencję naszych łańcuchów oraz zlecamy ich wyświetlenie:

```
s1.insert(find(s1.begin(),s1.end()," jest"), "takze");
copy (s1.begin(),s1.end(),output);
```

Najpierw przy pomocy metody **find()** w wektorze **s1** szukamy napisu "**jest**". Funkcja **find()** zwraca pozycję pierwszego elementu o wartości przekazanej w argumencie (u nas jest to napis "**jest**"). Po ustaleniu pozycji, metoda **insert()** przed znalezionym elementem "**jest**" wstawia element, którego wartość jest przekazywana przez drugi jej argument, w naszym przypadku jest to napis "**takze**". Metoda **copy()** korzystając z iteratora wyjściowego **ostream_iterator<string>** powoduje wyświetlenie wszystkich elementów nowej sekwencji wektora **s1**.

Kontener **map** pozwala na elegancką obsługę par danych, z których jeden element może być łańcuchem. Kolejny przykład wykorzystuje mapę, jako tablicę asocjacyjną. Tworzymy pary – nazwa pojazdu i jego średnia prędkość. W takiej parze kluczem jest nazwa pojazdu a wartością jest szybkość. W programie tworzymy kolekcję pojazdów i ich szybkości podanych w kilometrach na godzinę, a następnie podajemy szybkość w milach na godzinę.

Wydruk 7.9. obiekt **string** i kontener **map**

```
#include <iostream>
#include <map>
#include <string>
#include <iterator>   //ostream_iterator
#include <conio.h>
using namespace std;

int main()
{ typedef map<string, float> sfmap;
 sfmap :: iterator p;
 sfmap pojazd;
 pojazd["rower"]   = 50.0;
 pojazd["auto"]    = 120.0;
 pojazd["pociag"]  = 160.0;
 pojazd["samolot"] = 950.0;
 cout <<"szybkosc w km/godz : " << endl;
 for (p = pojazd.begin(); p != pojazd.end(); ++p)
   cout << p-> first << "\t" << p-> second << endl ;
 for (p = pojazd.begin(); p != pojazd.end(); ++p)
    p-> second /= 1.609; ;
 cout <<"\nszybkosc w milach/godz : " << endl;
```

```
 for (p = pojazd.begin(); p != pojazd.end(); ++p)
   cout << p-> first << "\t" << p-> second << endl ;

 getche();
 return 0;
}
```

Wynikiem wykonania programu jest komunikat:

```
szybkosc w km/godz :
auto                120
pociag      160
rower               50
samolot     950

szybkosc w milach/godz :
auto                74.5805
pociag              99.4406
rower               31.0752
samolot     590.429
```

W programie utworzono mapę o nazwie **pojazd**, klucze są typu **string**, wartości są typu **float,** utworzono także iterator **p**:

```
typedef map<string, float> sfmap;
sfmap :: iterator p;
sfmap pojazd;
```

Przy pomocy instrukcji:

```
pojazd ["rower"]   = 50.0;
```

wstawiamy elementy do kontenera, a następnie przy pomocy pętli **for** wyświetlamy zawartość kontenera **pojazd.** Przypominamy, że dostęp do elementów mapy odbywa się za pośrednictwem iteratorów.
Jeżeli zdefiniujemy iterator:

```
map<string, float> :: iterator p;
```

to dzięki iteratorowi możemy przeglądać elementy mapy.
Wyrażenie

```
p -> first
```

zwraca klucz bieżącego elementu.

Z kolei wyrażenie:

p-> second

zwraca jego wartość.
Jeżeli chcemy modyfikować wartość to możemy skorzystać na przykład z prostego podstawienia:

p -> second = 1313.13;

Wypisywanie wszystkich elementów mapy realizuje się przy pomocy instrukcji:

```
for (p = pojazd.begin(); p != pojazd.end(); ++p)
     cout << p-> first << "\t" << p-> second << endl ;
```

W kolejnej pętli realizowane jest zamiana wartości, szybkość w km/godz. wyliczana jest w jednostkach mila/godz.:

```
for (p = pojazd.begin(); p != pojazd.end(); ++p)
     p-> second /= 1.609; ;
```

Pamiętamy, że 1 mila to 1.609 km.
Kontener wektor umożliwia przechowywanie obiektów klasy, zdefiniowanej przez użytkownika (wydruk 7.10). Jest to bardzo silne narzędzie. Tego typu konstrukcję zademonstrujemy na przykładzie tworzenia bazy danych o pracownikach.

Wydruk 7.10. obiekt **string**, kontener **vector**, klasa użytkownika

```
#include <iostream>
#include <vector>
#include <string>
#include <conio.h>
using namespace std;
class osoba
{ public:
   string imie;
   string nazwisko;
   string pesel;
   osoba( string iimie, string nnazwisko, string ppesel );
};
int main()
{  int n;
  string xi,xn,xp;
   vector < osoba > pracownik;
   cout << "ilu pracownikow wprowadzic: " ;
   cin >> n;
```

```
  for (int i = 0; i<n; ++i)
    { cout << "\n Imie      : ";
        cin >> xi;
      cout << "Nazwisko    : ";
        cin >> xn;
      cout << "pesel       : ";
        cin >> xp;
      pracownik.push_back( osoba( xi,xn,xp) );
     }
  cout << "\nLista osob:\n";
  for( int i = 0; i < pracownik.size(); i++ )
     { cout << endl;
       cout << "Imie     : " << pracownik[i].imie << endl;
       cout << "Nazwisko  : " << pracownik[i].nazwisko << endl;
       cout << "pesel    : " << pracownik[i].pesel << endl;
     }
  getche();
  return 0;
}

osoba::osoba( string iimie, string nnazwisko, string ppesel )
        : imie( iimie ), nazwisko( nnazwisko ), pesel( ppesel )
  {
  }
```

Wynikiem działania program może być wydruk:

```
ilu pracownikow wprowadzic: 2
 Imie                  : Lola
Nazwisko               : Kwiatek
Pesel                  : 1111

Imie                   : Buba
Nazwisko               : Kowalska
Pesel                  : 2222

Lista osob:
 Imie                  : Lola
Nazwisko               : Kwiatek
Pesel                  : 1111

 Imie                  : Buba
Nazwisko               : Kowalska
Pesel                  : 2222
```

Z pokazanego przykładu możemy wnioskować, że wykorzystanie kontenerów i odpowiednich metod może znacznie uprościć pisanie praktycznych aplikacji.

BIBLIOGRAFIA

1. B. Stroustrup, Programowanie, Teoria i praktyka z wykorzystaniem C++, Helon, Gliwice, 2010
2. A.Stepanow, M. Lee, The Standard Template Library, Silicon Graphics Inc, CA, USA, 1995
3. J.Weidl, The Standard Template Library Tutorial, Technical University Vienna, 1996
4. N. M. Josuttis, C++, Biblioteka standardowa, podręcznik programisty,
5. Helion, Gliwice, 2001
6. H. Schildt, C++, Programowanie, Wydawnictwo RM, Warszawa, 2002
7. B. Eckel, c. Allison, Thinking in C++, tom 2, Helion, Gliwice, 2004
8. A.Koenig, B. Moo, C++, potęga języka, Helion, 2004
9. S.Prata, Szkoła programowania,. Język C++, Helion, Gliwice, 2005
10. A.Koenig, B. Moo, Język C++, Koncepcje i techniki programowania, Helion, 2005
11. N.Solter, S.Kleper, C++, zaawansowane programowanie, Helion, Gliwice, 2006