Лекція 13

Контейнери

У цій лекції...

Контейнерами називаються об'єкти, що містять інші об'єкти. Контейнери управляють розміщенням об'єктів в оперативній пам'яті, використовуючи конструктори, видаляють об'єкти з пам'яті за допомогою деструкторів, а також додають і видаляють об'єкти за допомогою операцій вставки й видалення.

У стандартній бібліотеці описано 10 шаблонних класів контейнерів.

Назва	Заголовок	Опис
`vector`	`<vector>`	Вектор
`list`	`<list>`	Двозв'язний список
`deque`	`<deque>`	Двостороння черга
`set`	`<set>`	Множина
`multiset`	`<set>`	Мультимножина
`map`	`<map>`	Асоціативний масив, що містить унікальні ключі й значення
`multimap`	`<multimap>`	Асоціативний масив, що допускає дублювання ключів і значень
`stack`	`<stack>`	Стек
`queue`	`<queue>`	Черга
`priority_queue`	`<queue>`	Черга із пріоритетом
`bitset`	`<bitset>`	Бітовий набір

13.1. Загальна структура контейнерів

Кожен контейнер повинен задовольняти наступним вимогам.

1. Контейнер може містити об'єкти будь-якого типу, крім посилань.
2. Конструктор копіювання об'єкта, що втримується в контейнері, повинен виконувати операцію вставки.
3. Виділення й звільнення пам'яті, зайнятої об'єктом, що зберігається в контейнері, повинне здійснюватися автоматично.
4. Контейнер повинен допускати автоматичну зміну розмірів.
5. Деструктор контейнера повинен викликати деструктори об'єктів, що зберігаються в ньому. Якщо конструктор містить вказівники, звільнення пам'яті здійснюється окремою операцією.

Контейнери в бібліотеці STL можна розділити на дві категорії — *послідовності* й *асоціативні контейнери*.

Для стандартизації імен типів елементів, що зберігаються в контейнері, у контейнерних класах утримується оператор `typedef`, що уніфікує імена, як показано в табл. 13.1. Операції, загальні для всіх контейнерів, перераховані в табл. 13.2.

Таблиця 13.1. Імена типів, загальні для всіх контейнерів

Ім'я	Опис
`size_type`	Цілочисельний тип
`reference`	Посилання на елемент
`const_reference`	Константне посилання на елемент
`difference_type`	Цілочисельний тип відстані між двома ітераторами
`iterator`	Ітератор
`const_iterator`	Константний ітератор
`reverse_iterator`	Зворотний ітератор
`const_reverse_iterator`	Константний зворотний ітератор
`value_type`	Тип значення, що зберігається в контейнері
`allocator_type`	Тип розподільника
`key_type`	Тип ключа
`key_compare`	Тип функції, що порівнює два ключі
`value_compare`	Тип функції, що порівнює два значення

Таблиця 13.1. Операції, загальні для всіх контейнерів

Операції над ітераторами	
`iterator begin()` `const iterator begin()`	Указує на перший елемент
`iterator end()` `const iterator end()`	Указує на позамежний елемент
`reverse_iterator rbegin()` `const iterator rbegin()`	Указує на перший елемент зворотної послідовності
`reverse_iterator rend()` `const iterator rend()`	Указує на позамежний елемент зворотної послідовності
Конструктори	
`explicit Container()`	Конструктор без параметрів. Створює порожній об'єкт, що має тип `Container`
`Container(const Container &)`	Конструктор копіювання
`~Container()`	Деструктор
`Container(iterator first, iterator last)`	Створює об'єкт класу `Container`, копіюючи в нього діапазон елементів з іншого контейнера, обмежений ітераторами `first` й `last`
Допоміжні функції	
`size()`	Повертає кількість елементів у контейнері
`empty()`	Перевіряє, чи порожній контейнер
`max_size()`	Визначає максимально можливий розмір об'єкта класу `Container`
`swap(Container&, Container&)`	Міняє місцями два контейнери
Операції порівняння	
`==, !=, <, <=, >, >=`	Два контейнери вважаються рівними, якщо вони містять однакову кількість елементів одного типу, розташованих в однаковому порядку
Присвоювання	
`operator=(const Container&)`	Копіювання існуючого контейнера

13.2. Послідовності

Послідовність — це різновид контейнерів, що містять скінчену множину однотипних об'єктів, упорядкованих у лінійному порядку. У бібліотеці STL передбачено три види послідовних контейнерів: вектор (клас `vector`), список (клас `list`) і двостороння черга (клас `deque`). Крім того, вони є базовими класами для створення більше складних контейнерів — стека (клас `stack`), черги (клас `queue`) і черги із пріоритетом (клас `priority_queue`). Для кожної з послідовностей передбачений свій набір операцій, але існують операції, загальні для всіх послідовностей (табл. 13.2).

Таблиця 13.2. Операції, загальні для всіх послідовностей

Конструктор	
`explicit Sequence(size_type n, const T& v = T())`	Копіює n елементів контейнера v. Якщо клас T не має конструктора без параметрів,

	використається явний виклик конструктора
Присвоювання	
`assign(first, last)`	Копіює діапазон, заданий ітераторами вводу `first` й `second`
`assign(size_type n, const T& v = T()`	Присвоює `n` елементів контейнера `v`
Доступ до елементів контейнера	
`reference front()` `const reference front()`	Повертає перший елемент. Тип `reference` залежить від типу контейнера (як правило, `T&`)
`reference back()` `const reference back()`	Повертає останній елемент
Вставка й видалення	
`iterator insert(iterator p, T t)`	Вставляє копію елемента `t` перед елементом, на який посилається ітератор `p`. Повертає ітератор, що вказує на вставлену копію
`void insert(iterator p, T t)`	Вставляє `n` копій елемента `t` перед елементом, на який посилається ітератор `p`
`void insert(iterator p, InputIterator i, InputIterator j)`	Вставляє копії елементів з діапазону `[i,j)` перед елементом, на який посилається ітератор `p`
`iterator erase(iterator p)`	Видаляє елемент, на який посилається ітератор `p`, повертає ітератор, установлений на наступний елемент. Якщо цього елемента ні, повертається значення `end()`
`iterator erase(iterator i, iterator j)`	Видаляє діапазон `[i,j)`, повертає ітератор, установлений на наступний елемент. Якщо цього елемента ні, повертається значення `end()`
`clear()`	Видаляє всі елементи

13.2.1.1 Вектор

Клас `vector` описує вид послідовності, що підтримує довільний доступ до своїх елементів. Він допускає автоматичне збільшення розміру (але не зменшення!), а також забезпечує операції вставки й видалення елементів, а також переміщення односпрямованих ітераторів по елементах масиву.

13.2.1.1 Конструктори й деструктор

У класі `vector` визначені наступні конструктори й деструктор.

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

Створює порожній вектор.

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

Створює вектор, у якому `n` елементов мають значення `value`.

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

Створює вектор, елементи якого втримуються в заданому діапазоні. Якщо ітератори `first` й `last` є односпрямованими, двунаправленными або ітераторами довільного доступу, для створення об'єкта класу `vector` конструктор робить `N = first-last` викликів конструктора копіювання класу `T`, не перерозподіляючи пам'ять. Якщо ітератори `first` й `last` є ітераторами уведення, відстань між ітераторами обчислити неможливо, тому виконується перерозподіл пам'яті й кількість викликів конструктора копіювання збільшується до `2N`.

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

Створює вектор, ініціалізуючи його елементами параметра `x`.

- `~vector();`

Знищує вектор.

13.2.1.2 Операції копіювання й присвоювання

У класі `vector` визначені наступні операції копіювання й присвоювання.

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

Присвоює викликаючому вектору елементи параметра `x`.

- ```
  template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
  ```

Присвоює вектору елементи з діапазону, обмеженого ітераторами `first` й `last`.

- `void assign(size_type n, const T& u);`

Присвоює вектору `n` елементів, що мають значення `u`.

13.2.1.3 Зміна й визначення розміру об'єкта

Маніпуляції з розміром об'єкта класу `vector` визначаються наступними функціями-членами.

- `size_type size() const;`

Визначає кількість елементів, що зберігаються у векторі.

- `size_type max_size() const;`

Визначає максимальна кількість елементів, що зберігаються у векторі.

- `void resize(size_type sz, T c = T());`

Змінює розмір вектора. Новий розмір задається параметром `sz`. Якщо вектор збільшується в розмірі, нові елементи мають значення `c`.

- `size_type capacity() const;`

Повертає розмір пам'яті, виділеної для вектора.

- `bool empty() const;`

Якщо вектор порожній, повертає значення `true`, якщо немає — значення `false`.

- `void reserve(syze_type n);`

Виділяє пам'ять для вектора. Якщо поточний розмір вектора менше величини, що повертає функцією `reserve()`, відбувається додаткове виділення пам'яті. Це перерозподіл пам'яті робить недійсними всі посилання, вказівники й ітератори, що посилаються на елементи вектора.

13.2.1.4 Операції доступу

Доступ до елементів об'єкта класу `vector` забезпечується наступними функціями-членами.

- `reference operator[](size_type n);`
- `const_reference operator[](size_type n) const;`

Оператор індексованого доступу до елемента вектора.

- `const_reference at(syze_type n) const;`
- `reference at(size_type n);`

Повертає посилання на *n-й* елемент вектора.

- `reference front();`
- `const_reference front() const;`

Повертає посилання, установлену на перший елемент вектора.

- `reference back();`
- `const_reference back() const;`

Повертає посилання, установлену на останній елемент вектора.

13.2.1.5 Операції вставки

Вставка елементів в об'єкт класу `vector` здійснюється наступними функціями-членами.

- `iterator insert(iterator position, const T& x = T());`

Вставляє новий елемент, що має значення `x`, перед елементом, на який посилається ітератор `position`.

- `void insert(iterator position, size_type n, const T& x);`
  ```

Вставляє n елементів, що мають значення x, перед елементом, на який посилається ітератор position.

- `template <class InputIterator>`
  `void insert(iterator position, InputIterator first, InputIterator last);`

Вставляє елементи з діапазону, обмеженого ітераторами first й last, перед елементом, на який установлений ітератор position.

- `void push_back(const T& x);`

Додає в кінець вектора елемент, що має значення x.

Оскільки елементи вектора індексовані, вставка приводить до їхніх перестановок. Поки розмір зарезервованої пам'яті більше розміру вектора, операція вставки елемента не приводить до перерозподілу пам'яті. Якщо після вставки розмір вектора перевершує максимально припустимий, відбувається перерозподіл. Складність вставки єдиного елемента у вектор прямо пропорційна відстані точки вставки від кінця вектора. Складність вставки діапазону елементів прямо пропорційна сумі кількості елементів і відстані до кінця вектора. Таким чином, вставка всього діапазону набагато ефективніше, ніж почергова вставка окремих елементів цього діапазону. Вставка всього діапазону допускається лише для односпрямованих і двунаправленных ітераторов, а також для ітераторов довільного доступу. ітератори уведення дозволяють лише заелементну вставку.

13.2.1.6 Операції видалення

Видалення елементів з об'єкта класу vector здійснюється наступними функціями-членами.

- `void pop_back();`

Видаляє останній елемент вектора.

- `void erase(iterator position);`

Видаляє елемент із зазначеної позиції.

- `void erase(iterator first, iterator last);`

Видаляє елементи із зазначеного діапазону.

- `void swap(vector<T, Allocator>&);`

Міняє місцями елементи викликаючого вектора й аргументу.

- `void clear();`

Видаляє всі елементи з вектора.

Операція видалення робить недійсними всі ітератори й посилання на елементи, розташовані після крапки стирання. Деструктор класу T викликається для кожного з елементів контейнера, що видаляють, а оператор присвоювання, певний у класі T, - для кожного з елементів, розташованих після крапки стирання.

13.2.7. Операції порівняння

Порівняння елементів об'єкта класу vector здійснюється наступними функціями-членами.

- ```
  template <class T, class Allocator>
    bool operator==(const vector<T, Allocator>& x,
                    const vector<T, Allocator>& y);
  ```

Оператор перевірки рівності двох векторів. Повертає значення true, якщо об'єкти x й y складаються з однакових елементів.

- ```
  template <class T, class Allocator>
     bool operator<(const vector<T, Allocator>& x,
                    const vector<T, Allocator>& y);
  ```

Оператор порівняння двох векторів. Повертає значення true, якщо об'єкт x менше об'єкта y.

- ```
  template <class T, class Allocator>
    bool operator!=(const vector<T, Allocator>& x,
                    const vector<T, Allocator>& y);
  ```

Оператор перевірки нерівності двох векторів. Повертає значення true, якщо об'єкти x й y складаються з неоднакових елементів.

- ```
  template <class T, class Allocator>
    bool operator>(const vector<T, Allocator>& x,
                   const vector<T, Allocator>& y);
  ```

Оператор порівняння двох векторів. Повертає значення true, якщо об'єкт x більше об'єкта y.

- ```
  template <class T, class Allocator>
    bool operator>=(const vector<T, Allocator>& x,
                    const vector<T, Allocator>& y);
  ```

Оператор порівняння двох векторів. Повертає значення true, якщо об'єкт x більше або дорівнює об'єкту y.
  ```

- `template <class T, class Allocator>`
  `bool operator<=(const vector<T, Allocator>& x,`
  `const vector<T, Allocator>& y);`

Оператор порівняння двох векторів. Повертає значення `true`, якщо об'єкт `x` менше або дорівнює об'єкту `y`.

- `template <class T, class Allocator>`
  `void swap(vector<T,Allocator>& x, vector<T, Allocator>& y);`

Спеціальний алгоритм перестановки, результатом якого є об'єкт `x.swap(y);`.

Приклад вектора

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

using namespace std ;

void print(vector<int> &array);

int main()
{
   const int SIZE = 10;
   vector<int> array;    // Порожній вектор

   // Заповнюємо вектор за допомогою функції-члена push_back()

   for (int i = 0; i < SIZE; i++)  array.push_back(i);

   // Виводимо вектор на екран
   printf("Вихідний вектор\n");
   print(array);

   // Видаляємо 5-й елемент вектора

   array.erase(array.begin() + 4);
      printf("Після видалення числа 4\n");
   print(array);

   // Видаляємо діапазон [6,9)
   array.erase(array.begin()+5, array.begin()+8);
   printf("Після видалення чисел 6, 7 й 8\n");
   print(array);

   // Виводимо на печатку зарезервований розмір вектора
   int arrayMaxSize = array.max_size();
   printf("Максимальний розмір вектора = %d\n", arrayMaxSize);

   // Виводимо на печатку розмір вектора
   int arraySize = array.size();
   printf("Розмір вектора = %d\n", arraySize);

   // Виводимо на печатку ємність вектора
   int arrayCapacity = array.size();
   printf("Ємність вектора = %d\n", arrayCapacity);

   // Вставляємо в діапазон [1,5) чотири п'ятірки й виводимо вектор на печатку
   array.insert(array.begin()+1,4,5);
   printf("Після вставки чотирьох п'ятірок у діапазон [1,5) \n");
   print(array);

   // Видаляємо останній елемент
   printf("Після видалення останнього елемента\n");
   array.pop_back();
   print(array);

   // Повідомляємо новий вектор. Копіюємо в нього вміст вектора array
```

```
    vector<int> newArray(array.begin(),array.end());
    printf("Вектор newArray\n");
    print(newArray);

    // Заповнюємо вектор newArray сімками
    newArray.insert(newArray.begin(),newArray.size(),7);
    printf("Після вставки сімок\n");
    print(newArray);

    // Дописуємо у вектор newArray уміст вектора newArray
    newArray.assign(array.begin(),array.end());
    printf("Після присвоєння вектора array\n");
    print(newArray);

    // Стираємо вектор newArray
    newArray.clear();
    printf("Спроба вивести порожній вектор\n");
    print(newArray);

 // Порівняння векторів
    array.clear();
    for (i = 0; i < SIZE; i++)
    {
        array.push_back(i);
        newArray.push_back(i+1);
    }
    printf("Вектор array:    ");
    print(array);
    printf("Вектор newArray: ");
    print(newArray);
    if(array == newArray) printf("array == newArray\n");
    if(array >  newArray) printf("array >  newArray\n");
    if(array >= newArray) printf("array >= newArray\n");
    if(array <  newArray) printf("array <  newArray\n");
    if(array <= newArray) printf("array <= newArray\n");

    return 0;
 }

 void print(vector<int> &array)
 {
    if (array.empty())
    {
        printf("%s \n","Вектор порожній");
        return;
    }


    vector<int>::iterator iter;

    for (iter = array.begin(); iter != array.end(); iter++)
         printf("%d ", *iter);
         printf("\n") ;
}
```

Результат

```
Вихідний вектор
0 1 2 3 4 5 6 7 8 9
Після видалення числа 4
0 1 2 3 5 6 7 8 9
Після видалення чисел 6, 7 й 8
0 1 2 3 5 9
Максимальний розмір = 1073741823
```

```
Розмір вектора = 6
Ємність вектора = 6
Після вставки чотирьох п'ятірок у діапазон [1,5)
0 5 5 5 5 1 2 3 5 9
Після видалення останнього елемента
0 5 5 5 5 1 2 3 5
Вектор newArray
0 5 5 5 5 1 2 3 5
Після вставки сімок
7 7 7 7 7 7 7 7 0 5 5 5 5 1 2 3 5
Після присвоєння вектора array
0 5 5 5 5 1 2 3 5
Спроба вивести порожній вектор
Вектор порожній
Вектор array:    0 1 2 3 4 5 6 7 8 9
Вектор newArray: 1 2 3 4 5 6 7 8 9 10
array <  newArray
array <= newArray
```

Для більше ефективного розподілу пам'яті в стандартній бібліотеці визначений шаблонний клас `vector<bool>`, що описує компактний вектор. У ньому визначений клас `reference`, що описує поводження посилань на окремий біт в об'єкті класу `vector<bool>`. Інші шаблонові функції являють собою уточнення функцій-членів класу `vector<T,Allocator>`, коли клас `T` є типом `bool`.

13.2.2 Список

Клас `list` визначає різновид послідовності, що підтримує двунаправленные ітератори. У цьому класі передбачені операції вставки й стирання в довільному місці послідовності з автоматичним управлінням пам'яті. На відміну від інших різновидів послідовності, клас `list` допускає лише послідовний доступ до елементів списку, використовуючи двунаправленный ітератор, що посилається на об'єкти класу `T`.

13.2.2.1 Конструктори й деструктор

Клас `list` мають наступні конструктори й деструктор.

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

Створює порожній список.

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

Створює список, у якому `n` елементов мають значення `value`.

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

Створює список, елементи якого втримуються в заданому діапазоні.

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

Створює список, ініціалізуючи його елементами аргументу `x`.

- `~list();`

Знищує список.

13.2.2.2. Операції копіювання й присвоювання

Копіювання й присвоювання об'єктів класу `list` здійснюється наступними функціями-членами.

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

Присвоює викликаючому об'єкту елементи параметра `x`.

- ```
  template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
  ```

Присвоює списку елементи з діапазону, обмеженого ітераторами `first` й `last`.

- `void assign(size_type n, const T& t);`

Присвоює списку `n` елементів, що мають значення `u`.
  ```

13.2.2.3. Зміна й визначення розміру об'єкта

Маніпуляції з розміром об'єктів класу `list` забезпечуються наступними функціями-членами.

- `bool empty() const;`

Якщо список порожній, повертає значення `true`, якщо немає — значення `false`.

- `size_type size() const;`

Повертає кількість елементів у списку.

- `size_type max_size() const;`

Визначає максимальна кількість елементів, що зберігаються в списку.

- `void resize(size_type sz, T c = T());`

Змінює розмір списку. Новий розмір задається параметром `sz`. Якщо список збільшується в розмірі, нові елементи мають значення `c`.

13.2.2.4. Операції доступу

Доступ до елементів об'єкта класу `list` здійснюється наступними функціями-членами.

- `reference front();`
- `const_reference front() const;`

Повертає посилання, установлену на перший елемент списку.

- `reference back();`
- `const_reference back() const;`

Повертає посилання, установлену на останній елемент списку.

13.2.2.5. Операції вставки

Вставку елементів об'єкта класу `list` виконують наступні функції-члени.

- `void push_front(const T& x);`

Додає в початок списку елемент, що має значення `x`.

- `void push_back(const T& x);`

Додає в кінець списку елемент, що має значення `x`.

- `void pop_front();`

Видаляє елемент із початку списку.

- `void pop_back();`

Видаляє елемент із кінця списку.

- `iterator insert(iterator position, const T& x);`

Вставляє новий елемент, що має значення `x`, перед елементом, на який посилається ітератор `position`.

- `void insert(iterator position, size_type n, const T& x);`

Вставляє `n` елементів, що мають значення `x`, перед елементом, на який посилається ітератор `position`.

- ```
  template <class InputIterator>
    void insert(iterator position, InputIterator first, InputIterator last);
  ```

Вставляє елементи з діапазону, обмеженого ітераторами `first` й `last`, перед елементом, на який установлений ітератор `position`.

Операції вставки елементів у список не впливають на коректність ітераторов і посилань. Вставка одного елемента в список забирає постійний час. Конструктор копіювання класу `T` при вставці елемента викликається тільки один раз. Складність вставки декількох елементів прямо пропорційна кількості елементів, що вставляють, а число викликів конструктора `копіювання` `T` дорівнює числу вставлених елементів.

13.2.2.6. Операції видалення

Видалення елементів з об'єкта класу `list` здійснюється наступними функціями-членами.

- `void erase(iterator position);`

Видаляє елемент із зазначеної позиції.

- `void erase(iterator first, iterator last);`

Видаляє елементи із зазначеного діапазону.
  ```

- `void swap(list<T,Allocator>&);`

Міняє місцями елементи викликаючого списку й аргументу.

- `void clear();`

Видаляє всі елементи зі списку.

Зазначені операції анулюють ітератори й посилання, що вказують на вилучені елементи. Видалення одного елемента не вимагає переміщення інших елементів і забирає постійний час. Деструктор класу `T` при видаленні елемента викликається тільки один раз. Видалення діапазону елементів списку прямо пропорційно розміру діапазону, а число викликів деструктора класу `T` дорівнює розміру діапазону.

13.2.2.7. Спеціальні операції над списком

Для об'єкта класу `list` передбачено кілька спеціальних операцій, які виконуються наступними функціями-членами.

- `void splice(iterator position, list<T, Allocator>& x)`

Вставляє вміст списку *x* перед позицією, заданої ітератором `position`, причому список *x* у результаті спустошується. Якщо ця операція застосовується до викликаючого списку (`&x == this`), результат не визначений.

- `void splice(iterator position, list<T, Allocator>& x, iterator i)`

Вставляє елемент, на який посилається ітератор `i`, зі списку `x` перед позицією, заданої параметром `position`, і видаляє елемент зі списку `x`.

- ```
  void splice(iterator position, list<T, Allocator>& x,
              iterator first, iterator last)
  ```

Вставляє елементи з діапазону `[first, last)` перед елементом, заданим аргументом `position`, і видаляє цей діапазон з `x`. Якщо ітератор `position` перебуває в діапазоні `[first, last)`, результат операції не визначений.

- `void remove(const T& value);`
- `template <class Predicate> void remove_if(Predicate pred);`

Ці функції видаляють зі списку елементи, для яких виконуються умови `*i == value` й `Pred(*i) == true`, де *i* — ітератор контейнера, а `Pred` — предикат. Якщо ці умови порушуються, генерується виняткова ситуація. Порядок проходження інших елементів списку не змінюється. Предикат `pred` виконується `size` раз.

- `void unique();`
- `template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);`

Видаляють із кожної групи однакових елементів списку всі елементи, крім першого. Таким чином, зі списку виключаються всі дублікати. Відповідний бінарний предикат застосовується `size()-1` раз.

- `void merge(list<T, Allocator>& x);`
- `template <class Compare> void merge(list<T,Allocator>& x, Compare comp);`

Поєднують список, заданий аргументом, із зухвалим списком (передбачається, що обидва списки впорядковані). Якщо в об'єднаному списку існують однакові елементи, спочатку треба елемент із викликаючого списку. Функції `merge()` виконують не більше `size()+x.size()-1` порівнянь.

- `void reverse();`

Переставляє елементи списку у зворотному порядку. Складність цієї ітерації прямо пропорційна довжині списку.

- `void sort();`
- `template <class Compare> void sort(Compare comp);`

13.2.2.8. Операції порівняння

Упорядковують список, відповідно до відношення порівняння, заданому функцією `operator<()` або функтором. Складність операції приблизно дорівнює *Nlog*, де *N* дорівнює `size()`.

- ```
  template <class T, class Allocator>
    bool operator==(const list<T, Allocator>& x,
                    const list<T, Allocator>& y);
  ```

Оператор перевірки рівності двох списків. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з однакових елементів.

- ```
  template <class T, class Allocator>
     bool operator<(const list<T, Allocator>& x,
                    const list<T, Allocator>& y);
  ```
  ```

Оператор порівняння двох списків. Повертає значення `true`, якщо об'єкт `x` менше об'єкта `y`.

- ```
  template <class T, class Allocator>
    bool operator!=(const list<T, Allocator>& x,
                    const list<T, Allocator>& y);
  ```

Оператор перевірки нерівності двох списків. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з неоднакових елементів.

- ```
  template <class T, class Allocator>
    bool operator>(const list<T, Allocator>& x,
                   const list<T, Allocator>& y);
  ```

Оператор порівняння двох списків. Повертає значення `true`, якщо об'єкт `x` більше об'єкта `y`.

- ```
  template <class T, class Allocator>
    bool operator>=(const list<T, Allocator>& x,
                    const list<T, Allocator>& y);
  ```

Оператор порівняння двох списків. Повертає значення `true`, якщо об'єкт `x` більше або дорівнює об'єкту `y`.

- ```
  template <class T, class Allocator>
    bool operator<=(const list<T, Allocator>& x,
                    const list<T, Allocator>& y);
  ```

Оператор порівняння двох списків. Повертає значення `true`, якщо об'єкт `x` менше або дорівнює об'єкту `y`.

- ```
  template <class T, class Allocator>
     void swap(vector<T,Allocator>& x, vector<T, Allocator>& y);
  ```

Спеціальний алгоритм, результатом якого є `x.swap(y);`.

Приклад списку

```
#include <iostream>
#include <algorithm>
#include <list>

using namespace std ;

void print(list<int>&List);

int main()
{
   const int SIZE = 10;
   int i;

   // Порожній список
   list<int>List;

   list<int>::iterator where;

   // Заповнюємо список за допомогою функції-члена push_back()
   for (i = 0; i < SIZE; i++)  List.push_back(i);

   // Заповнюємо список за допомогою функції-члена push_front()
   for (i = 0; i < SIZE; i++)  List.push_front(100-i);

   // Виводимо список на екран
   printf("Вихідний список\n");
   print(List);

   // Знаходимо й видаляємо зі списку число 10

   where = find(List.begin(),List.end(),9);
   if( where != List.end()) List.erase(where);
     else printf("Такого числа в списку ні");
   printf("Після видалення числа 9\n");
   print(List);

   // Видаляємо число з початку списку
   if(!List.empty())List.pop_front();
   printf("Після видалення числа з початку списку\n");
```
```

```
print(List);

// Видаляємо число з кінця списку
if(!List.empty())List.pop_back();
printf("Після видалення числа з кінця списку\n");
print(List);

// Видаляємо всі числа, не рівні 5
List.remove_if(bind2nd(not_equal_to<int>(), 5));
printf("Після видалення всіх чисел, не рівних 5\n");
print(List);

// Виводимо на печатку зарезервований розмір списку
int ListMaxSize = List.max_size();
printf("Максимальний розмір списку = %d\n", ListMaxSize);

// Виводимо на печатку розмір списку
int ListSize = List.size();
printf("Розмір списку = %d\n", ListSize);

// Виводимо на печатку ємність списку
int ListCapacity = List.size();
printf("Ємність списку = %d\n", ListCapacity);

// Вставляємо в початок списку 5 десяток
List.insert(List.begin(),5, 10);
printf("Після вставки п'яти десяток у початок списку\n");
print(List);

// Видаляємо останній елемент
printf("Після видалення останнього елемента\n");
List.pop_back();
print(List);

// Повідомляємо новий список. Копіюємо в нього вміст списку List
list<int>newList(List.begin(),List.end());
printf("Список newList\n");
print(newList);

// Заповнюємо список newList сімками
newList.insert(newList.begin(),newList.size(),7);
printf("Після заповнення списку newList сімками\n");
print(newList);

// Дописуємо в список newList уміст списку List
newList.assign(List.begin(),List.end());
printf("Після присвоєння списку List\n");
print(newList);

// Стираємо список newList
newList.clear();
printf("Спроба вивести порожній список\n");
print(newList);

// Видалення дублікатів
for (i = 0; i < SIZE; i++)
{
   if( i < 5) newList.push_back(20); else newList.push_back(30);
}
printf("Список з дублікатами\n");
print(newList);
newList.unique();
printf("Список без дублікатів\n");
```

```
    print(newList);

    // Злиття списків
    newList.merge(List);
    printf("Після злиття списків\n");
    print(newList);

    // Упорядкування списку
    newList.sort();
    printf("Після сортування\n");
    print(newList);

    // Функція splice()
    List.clear();
    newList.clear();
    for (i = 0; i < SIZE; i++)
    {
        List.push_back(i);
        newList.push_back(i+3);
    }
    print(List);
    print(newList);
    newList.splice(newList.begin(),List);
    printf("Після застосування функції splice\n");
    printf("newList = ");
    print(newList);
    printf("List = ");
    print(List);

    // Порівняння списків

    if(List == newList) printf("List == newList\n");
    if(List >  newList) printf("List >  newList\n");
    if(List >= newList) printf("List >= newList\n");
    if(List <  newList) printf("List <  newList\n");
    if(List <= newList) printf("List <= newList\n");

    return 0;
}

void print(list<int>&List)
{
    if (List.empty())
    {
        printf("%s \n","Список порожній");
        return;
    }

    list<int>::iterator iter;

    for (iter = List.begin(); iter != List.end(); iter++)
         printf("%d ", *iter);
         printf("\n") ;
}
```

Результат

```
Вихідний список
91 92 93 94 95 96 97 98 99 100 0 1 2 3 4 5 6 7 8 9
Після видалення числа 9
91 92 93 94 95 96 97 98 99 100 0 1 2 3 4 5 6 7 8
Після видалення числа з початку списку
92 93 94 95 96 97 98 99 100 0 1 2 3 4 5 6 7 8
Після видалення числа з кінця списку
```

```
92 93 94 95 96 97 98 99 100 0 1 2 3 4 5 6 7
Після видалення всіх чисел, не рівний 5
Максимальний розмір списку = 1073741823
Розмір списку = 1
Ємність списку = 1
Після вставки п'яти десяток у початок списку
10 10 10 10 10 5
Після видалення останнього елемента
10 10 10 10 10
Список newList
10 10 10 10 10
Після вставки сімок  у список newList
7 7 7 7 7 10 10 10 10 10
Після присвоєння списку List
10 10 10 10 10
Спроба вивести порожній список
Список порожній
Список з дублікатами
20 20 20 20 20 30 30 30 30 30
Список без дублікатів
20 30
Після злиття списків
10 10 10 10 10 20 30
Після сортування
10 10 10 10 10 20 30
0 1 2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10 11 12
Після застосування функції splice
newList = 0 1 2 3 4 5 6 7 8 9 3 4 5 6 7 8 9 10 11 12
List = Список порожній
List <  newList
List <= newList
```

13.2.3. Двостороння черга

Двостороння черга (дек) — це різновид послідовності, що підтримує ітератори довільного доступу. Вона забезпечує виконання операцій вставки й видалення елементів як на початку й кінці, так й у середині контейнера. Керування пам'яттю здійснюється автоматично. З одного боку, по своїх функціональних можливостях двостороння черга є комбінацією вектора й списку. З іншого боку, дек можна вважати стеком, у якого дозволені вставки в кінець і середину. Правда ці операції малоефективні.

13.2.3.1. Конструктори й деструктор

У класі `deque` визначені наступні конструктори й деструктор.

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

Створює порожній груд.

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

Створює дек, що містить `n` елементів, що мають значення `value`.

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

Створює дек, ініціалізуючи його елементами з діапазону, обмеженого ітераторами `first` й `last`.

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

Створює дек, ініціалізуючи його елементами об'єкта `x`.

- `~deque();`

Знищує сміття.

13.2.3.2 Операції копіювання й присвоювання

Копіювання й присвоювання об'єктів класу `deque` виконуються наступними функціями-членами.

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

Присвоює викликаючому об'єкту елементи параметра `x`.

- ```
  template <class InputIterator>
     void assign(InputIterator first, InputIterator last);
  ```

Присвоює деку елементи з діапазону, обмеженого ітераторами `first` й `last`.

- `void assign(size_type n, const T& t);`

Присвоює деку `n` елементів, що мають значення `u`.

13.2.3.3 Зміна й визначення розміру об'єкта

Маніпуляції з розміром об'єктів класу `deque` виконуються наступними функціями-членами.

- `size_type size() const;`

Повертає кількість елементів у деку.

- `size_type max_size() const;`

Повертає максимальна кількість елементів у деку.

- `void resize(size_type sz, T c = T());`

Змінює розмір дека. Новий розмір задається параметром `sz`. Якщо дек збільшується в розмірі, нові елементи мають значення `c`.

- `bool empty() const;`

Якщо дек порожній, повертає значення `true`, якщо немає — значення `false`.

13.2.3.4 Операції доступу

Доступ до елементів об'єктів класу `deque` забезпечується наступними функціями-членами.

- `reference operator[](size_type n);`
- `const_reference operator[](size_type n) const;`

Оператор доступу до елемента дека.

- `reference at(size_type n);`
- `const_reference at(size_type n) const;`

Повертає посилання на *n-й* елемент дека.

- `reference front();`
- `const_reference front() const;`

Повертає посилання на перший елемент дека.

- `reference back();`
- `const_reference back() const;`

Повертає посилання на останній елемент дека.

13.2.3.5. Операції вставки

Вставка елементів в об'єкти класу `deque` виконується наступними функціями-членами.

- `void push_front(const T& x);`

Додає елемент, що має значення `x`, у початок дека.

- `void push_back(const T& x);`

Додає елемент, що має значення `x`, у кінець дека.

- `iterator insert(iterator position, const T& x = T());`

Вставляє новий елемент, що має значення `x`, перед елементом, на який посилається ітератор `position`.

- `void insert(iterator position, size_type n, const T& x);`

Вставляє `n` елементів, що мають значення `x`, перед елементом, на який посилається ітератор `position`.

```
template <class InputIterator>
   void insert(iterator position, InputIterator first, InputIterator last);
```

Вставляє елементи з діапазону, обмеженого ітераторами `first` й `last`, перед елементом, на який установлений ітератор `position`.

Оскільки елементи дека індексовані, це приводить до їхніх перестановок. Поки розмір зарезервованої пам'яті більше розміру двосторонньої черги, операція вставки елемента не приводить до перерозподілу пам'яті. Якщо після вставки розмір дека перевершує максимально припустимий, відбувається перерозподіл. Складність вставки єдиного елемента у вектор прямо пропорційна відстані крапки вставки від кінця вектора. Складність
```

вставки діапазону елементів прямо пропорційна сумі кількості елементів і відстані до кінця вектора. Таким чином, вставка всього діапазону набагато ефективніше, ніж почергова вставка окремих елементів цього діапазону. Вставка всього діапазону допускається лише для односпрямованих і двунаправленных ітераторов, а також для ітераторов довільного доступу. ітератори уведення дозволяють лише заелементну вставку.
Операція вставки в середину двосторонньої черги робить недійсними всі ітератори й посилання на елементи дека. Крім того, виклик функцій-членів `insert()` і `push()` з обох кінців двосторонньої черги робить недійсними всі ітератори, установлені на дек, але не впливає на коректність посилань, установлених на окремі елементи двосторонньої черги. У найгіршому разі вставка окремого елемента в дек прямо пропорційна мінімуму двох відстаней: від місця вставки до початку й від місця вставки до кінця двосторонньої черги. Вставка одного елемента в початок або кінець дека виконується за постійний час і зводиться до одного виклику конструктора копіювання класу `T`. Таким чином, двостороння черга функціонує оптимально, якщо операції вставки й видалення виробляються лише на початку й кінці дека.

13.2.3.6. Операції видалення

Видалення елементів з об'єктів класу `deque` виконується наступними функціями-членами.

- `void pop_front();`

Видаляє перший елемент вектора.

- `void pop_back();`

Видаляє останній елемент вектора.

- `void erase(iterator position);`

Видаляє елемент із зазначеної позиції.

- `void erase(iterator first, iterator last);`

Видаляє елементи із зазначеного діапазону.

- `void swap(deque<T, Allocator>&);`

Міняє місцями елементи викликаючого дека й аргументу.

- `void clear();`

Видаляє всі елементи з дека.

13.2.3.7. Операції порівняння

Для порівняння елементів об'єктів класу `deque` призначені наступні функції-члени.

- ```
  template <class T, class Allocator>
    bool operator==(const deque<T, Allocator>& x,
                    const deque<T, Allocator>& y);
  ```

Оператор перевірки рівності двох деков. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з однакових елементів.

- ```
  template <class T, class Allocator>
    bool operator<(const deque<T, Allocator>& x,
                   const deque<T, Allocator>& y);
  ```

Оператор порівняння двох деков. Повертає значення `true`, якщо об'єкт `x` менше об'єкта `y`.

- ```
  template <class T, class Allocator>
    bool operator!=(const deque<T, Allocator>& x,
                    const deque<T, Allocator>& y);
  ```

Оператор перевірки нерівності двох деков. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з неоднакових елементів.

- ```
  template <class T, class Allocator>
    bool operator>(const deque<T, Allocator>& x,
                  const deque<T, Allocator>& y);
  ```

Оператор порівняння двох деков. Повертає значення `true`, якщо об'єкт `x` більше об'єкта `y`.

- ```
  template <class T, class Allocator>
    bool operator>=(const deque<T, Allocator>& x,
                    const deque<T, Allocator>& y);
  ```

Оператор порівняння двох деков. Повертає значення `true`, якщо об'єкт `x` більше об'єкта `y` або дорівнює йому.

- ```
  template <class T, class Allocator>
    bool operator<=(const deque<T, Allocator>& x,
                    const deque<T, Allocator>& y);
  ```

Оператор порівняння двох деков. Повертає значення `true`, якщо об'єкт `x` менше об'єкта `y` або дорівнює йому

- `template <class T, class Allocator>`
  `void swap(deque<T,Allocator>& x, deque<T, Allocator>& y);`

Спеціальний алгоритм перестановки, результатом якого є об'єкт `x.swap(y);`.

Приклад двосторонньої черги

```
#include <iostream>
#include <algorithm>
#include <deque>

using namespace std ;

void print(deque<int> &Deque);

int main()
{
   const int SIZE = 5;
   int i;

   // Порожній дек
   deque<int>Deque;

   deque<int>::iterator where;

   // Заповнюємо дек за допомогою функції-члена push_back()
   for (i = 0; i < SIZE; i++)  Deque.push_back(i);

   // Заповнюємо дек за допомогою функції-члена push_front()
   for (i = 0; i < SIZE; i++)  Deque.push_front(100-i);

   // Виводимо дек на екран
   printf("Вихідний дек\n");
   print(Deque);

  // Присвоюємо нові значення за допомогою функції-члена operator[]()
   for (i = 0; i < SIZE; i++)  Deque[i]=200-i;
   printf("Після присвоювання\n");
   print(Deque);

   // Знаходимо і видаляємо з дека число 198

   where = find(Deque.begin(),Deque.end(),198);
   if( where != Deque.end()) Deque.erase(where);
     else printf("Такого числа в списку ні");
      printf("Після видалення числа 198\n");
   print(Deque);

   // Видаляємо число з початку дека
   if(!Deque.empty())Deque.pop_front();
   printf("Після видалення числа з початку дека\n");
   print(Deque);

   // Видаляємо число з кінця дека
   if(!Deque.empty())Deque.pop_back();
   printf("Після видалення числа з кінця дека\n");
   print(Deque);

   // Визначаємо початок дека
   printf("Голова черги = %d\n",Deque.front());

   // Визначаємо кінець дека
   printf("Кінець черги = %d\n",Deque.back());

   // Визначаємо другий елемент дека
```

```
    printf("Число в другій позиції = %d\n",Deque.at(2));
    print(Deque);

    // Виводимо на печатку зарезервований розмір дека
    int DequeMaxSize = Deque.max_size();
    printf("Максимальний розмір дека = %d\n", DequeMaxSize);

    // Виводимо на печатку розмір дека
    int DequeSize = Deque.size();
    printf("Розмір дека = %d\n", DequeSize);

    // Виводимо на печатку ємність дека
    int DequeCapacity = Deque.size();
    printf("Ємність дека = %d\n", DequeCapacity);

    // Вставляємо в початок дека п'ять десяток
    Deque.insert(Deque.begin(),5, 10);
    printf("Після вставки п'яти десяток у початок дека\n");
    print(Deque);

    // Повідомляємо новий груд. Копіюємо в нього вміст дека Deque
    deque<int>newDeque(Deque.begin(),Deque.end());
    printf("Список newDeque\n");
    print(newDeque);

    // Заповнюємо дек newDeque сімками
    newDeque.insert(newDeque.begin(),newDeque.size(),7);
    printf("Після вставки сімок у дек newDeque\n");
    print(newDeque);

    // Дописуємо в дек newDeque уміст дека Deque
    newDeque.assign(Deque.begin(),Deque.end());
    printf("Після присвоєння дека Deque\n");
    print(newDeque);

    // Стираємо дек newDeque
    newDeque.clear();
    printf("Спроба вивести порожній дек\n");
    print(newDeque);

    // Порівняння деков

    if(Deque == newDeque) printf("Deque == newDeque\n");
    if(Deque >  newDeque) printf("Deque >  newDeque\n");
    if(Deque >= newDeque) printf("Deque >= newDeque\n");
    if(Deque <  newDeque) printf("Deque <  newDeque\n");
    if(Deque <= newDeque) printf("Deque <= newDeque\n");

    return 0;
}

 void print(deque<int> &Deque)
 {
    if (Deque.empty())
    {
        printf("%s \n","Дек порожній");
        return;
    }

    deque<int>::iterator iter;

    for (iter = Deque.begin(); iter != Deque.end(); iter++)
         printf("%d ", *iter);
```

```
        printf("\n") ;
}
```

Результат

```
Вихідний дек
96 97 98 99 100 0 1 2 3 4
Після присвоювання
200 199 198 197 196 0 1 2 3 4
Після видалення числа 198
200 199 197 196 0 1 2 3 4
Після видалення числа з початку дека
199 197 196 0 1 2 3 4
Після видалення числа з кінця дека
199 197 196 0 1 2 3
Голова черги = 199
Кінець черги  = 3
Число в другій позиції = 196
199 197 196 0 1 2 3
Максимальний розмір дека = 1073741823
Розмір дека  = 7
Ємність дека = 7
Після вставки п'яти десяток у початок дека
10 10 10 10 10 199 197 196 0 1 2 3
Список newDeque
10 10 10 10 10 199 197 196 0 1 2 3
Після вставки сімок у дек newDeque
7 7 7 7 7 7 7 7 7 7 7 10 10 10 10 10 199 197 196 0 1 2 3
Після присвоєння дека Deque
10 10 10 10 10 199 197 196 0 1 2 3
Спроба вивести порожній дек
Деків порожній
Deque >  newDeque
Deque >= newDeque
```

13.3 Адаптери послідовностей

Шаблонові класи `vector`, `list` й `deque` описують всі основні операції над послідовностями. Хоча вони містять досить велику кількість однакових операцій, по своїй структурі й принципам функціонування жоден із цих класів не можна замінити іншим. Завдяки тому, що в класах `vector`, `list` й `deque` реалізовані базові операції, їх можна використати як основу для створення більше складних структур — стеков і черг. Отже, для правильного визначення об'єктів адаптерів послідовностей у програму необхідно включати два заголовки: один повинен містити визначення шаблонового класу адаптера, а іншої — визначення базисного класу.

13.3.1. Стек

По визначенню стек являє собою структуру даних, організовану за принципом LIFO (“last-n, last-out” — “останнім увійшов — першим вийшов”). Для елементів стека повинні бути визначені відносини “менше” й “дорівнює”.

За замовчуванням стек будується на основі дека. Однак його можна створити, використовуючи вектор або список. Для цього при визначенні об'єкта як шаблонний параметр варто вказати відповідний клас — `vector` або `list`.

```
stack<int, vector<int> > obj1;
stack<int, list<int> >   obj2;
```

Якщо стек порожній, функція-член `empty()` повертає значення `true`. Функція-член `size()` повертає довжину стека, тобто кількість елементів, що втримуються в ньому. Функція-член `top()` повертає посилання на вершину стека, тобто на елемент, вставлений останнім. Функція-член `push()` заштовхує в стек новий елемент, а функція `pop()` виштовхує зі стека елемент, розташований на вершині. Зверніть увагу на те, що всі ці операції реалізуються за допомогою виклику відповідного функції-члена класу, заданого в якості другого шаблонового параметра.

Пример стека

```
#include <iostream>
#include <algorithm>
```

```
#include <stack>

using namespace std ;

void print(stack<int> &Stack);

int main()
{
   const int SIZE = 5;
   int i;

   // Порожній стек

   stack<int>Stack, newStack;

   // Заповнюємо стек за допомогою функції-члена push()
   for (i = 0; i < SIZE; i++)
   {
       Stack.push(i);
       newStack.push(i+1);
   }

   // Виводимо на печатку розмір стека
   int StackSize = Stack.size();
   printf("2 Розмір першого стека =  %d\n", StackSize);

   StackSize = newStack.size();
   printf("2 Розмір другого стека =  %d\n", StackSize);

   // Порівняння стеков

   if(Stack == newStack) printf("Stack == newStack\n");
   if(Stack >  newStack) printf("Stack >  newStack\n");
   if(Stack >= newStack) printf("Stack >= newStack\n");
   if(Stack <  newStack) printf("Stack <  newStack\n");
   if(Stack <= newStack) printf("Stack <= newStack\n");

   // Виводимо стек на екран
   printf("Перший стек\n");
   print(Stack);

   printf("Другий стек\n");
   print(newStack);

   return 0;
}

void print(stack<int> &Stack)
{
   if (Stack.empty())
   {
       printf("%s \n","Стек порожній");
       return;
   }


   while(!Stack.empty())
   {
        printf("%d ", Stack.top());
        Stack.pop();
   }
        printf("\n") ;
}
```

Результат

```
Розмір першого стека =  5
Розмір другого стека =  5
Stack <  newStack
Stack <= newStack
Перший стек
4 3 2 1 0
Другий стек
5 4 3 2 1
```

Примітка: оскільки доступ до елементів стека можлива лише на його вершині, вивести на печатку всі елементи можна, лише по черзі виштовхнувши їх зі стека.

13.3.2. Черга

За допомогою контейнера `deque` можна створити *черга* — структуру даних, організовану за принципом FIFO (“first in, first out” — “першим увійшов, першим вийшов”).

Як основу для створення черги може застосовуватися не тільки шаблонний клас `deque`, але й будь-яка інша послідовність, для якої передбачені функції `front()`, `back()`, `push_back()` і `pop_front()`. Зокрема, клас `vector` для цієї мети не підходить.

Як бачимо, у класі `queue` немає операції видалення останнього елемента — відповідно до принципу FIFO, операція `pop` видаляє тільки перший елемент, викликаючи з базового контейнера функцію `pop_front()`. Крім того, по тій же причині операція `push` вставляє новий об'єкт тільки в кінець черги, викликаючи з базового контейнера функцію `push_back()`.

Приклад черги

```
#include <iostream>
#include <algorithm>
#include <queue>

using namespace std ;

void print(queue<int> &Queue);

int main()
{
   const int SIZE = 5;
   int i;

   // Порожня черга

   queue<int>Queue, newQueue;

   // Заповнюємо стек за допомогою функції-члена push()
   for (i = 0; i < SIZE; i++)
   {
       Queue.push(i);
       newQueue.push(i+1);
   }

   // Виводимо на печатку розмір стека
   int QueueSize = Queue.size();
   printf("2 Розмір першої черги =  %d\n", QueueSize);

   QueueSize = newQueue.size();
   printf("2 Розмір другої черги =  %d\n", QueueSize);

   // Порівняння черг

   if(Queue == newQueue) printf("Queue == newQueue\n");
   if(Queue >  newQueue) printf("Queue >  newQueue\n");
   if(Queue >= newQueue) printf("Queue >= newQueue\n");
```

```
    if(Queue <  newQueue) printf("Queue <  newQueue\n");
    if(Queue <= newQueue) printf("Queue <= newQueue\n");

    // Виводимо черга на екран
    printf("Перша черга\n");
    print(Queue);

    printf("Друга черга\n");
    print(newQueue);

    return 0;
 }

 void print(queue<int> &Queue)
 {
    if (Queue.empty())
    {
        printf("%s \n","Черга порожня");
        return;
    }

    while(!Queue.empty())
    {
         printf("%d ", Queue.front());
         Queue.pop();
    }
         printf("\n") ;
 }
```

Результат

```
Розмір першої черги =  5
Розмір другої черги =  5
Queue <  newQueue
Queue <= newQueue
Перша черга
0 1 2 3 4
Друга черга
1 2 3 4 5
```

13.3.3 Черга із пріоритетом

Як відомо, у житті ідеальних черг не буває — завжди найдеться спосіб порушити черговість. У науці це явище називається *пріоритетом*, а відповідна структура даних — *чергою із пріоритетами*. Незалежно від того, у якому порядку елементи містилися в чергу, виключення з її відбувається відповідно до заданих пріоритетів. Для того щоб установити порядок виключення елементів, їхні пріоритети необхідно порівнювати за допомогою функції порівняння. Як і звичайну чергу, об'єкт класу `priority_queue` можна створювати на основі дека або вектора.

Очередь із пріоритетом реалізується на основі купи, тобто особливим образом організованого дерева. Вона підтримує ітератори довільного доступу, а також операції `front`, `push` й `pop`.

13.3.3.1 Конструктори

У класі `priority_queue` передбачені наступні конструктори.

- `explicit priority_queue(const Compare& x = Compare(),`
  `const Container& y = Container());`

Инициализирует об'єкт `comp` об'єктом `x`, а об'єкт `c` — об'єктом `y`. Після цього конструктор викликає функцію `make_heap(c.begin(), c.end(), comp)`, що створює купу.

- `template <class InputIterator>`
  `priority_queue(InputIterator first, InputIterator last,`
  `const Compare& x = Compare(),`
  `const Container& y = Container());`

Инициализирует об'єкт `comp` об'єктом `x`, а об'єкт `c` — об'єктом `y`. Після цього конструктор викликає функцію `c.insert(c.end(), first, last)`, ініціалізуючи контейнер, а потім викликає `make_heap(c.begin(), c.end(), comp)`, створюючи купу.

13.3.3.2 Функції-члени

Функціональні можливості класу `priority_queue` забезпечуються наступними функціями-членами.

- `bool empty() const;`

Якщо черга порожня, повертає значення `true`, якщо немає — значення `false`.

- `size_type size() const;`

Повертає кількість елементів, що зберігаються в черзі із пріоритетом.

- `const value_type& top() const;`

Повертає посилання на елемент, що має найбільший пріоритет.

- `void push(const value_type& x);`

Додає в чергу із пріоритетом елемент, що має значення `x`.

- `void pop();`

Видаляє елемент із черги із пріоритетом.

Приклад черги із пріоритетом

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>

using namespace std ;
typedef priority_queue<int, vector<int>, less<int> > PriorityQueue ;
void print(priority_queue<int, vector<int>, less<int> > &priorQueue);

int main()
{
   const int SIZE = 5;
   int i,k;


   // Порожня черга

   PriorityQueue priorQueue, newPriorQueue;

   // Заповнюємо черги за допомогою функції-члена push()

   printf("Порядок заповнення\n");
   for (i = 0; i < SIZE; i++)
   {
       k = rand();
       printf("%d ", k);
       priorQueue.push(k);
   }
   printf("\n");
   for (i = 0; i < SIZE; i++)
   {
       k = rand();
       printf("%d ", k);
       newPriorQueue.push(k);
   }
   printf("\n");

   // Виводимо на печатку розмір черг
   int QueueSize = priorQueue.size();
   printf("2 Розмір першої черги =  %d\n", QueueSize);

   QueueSize = newPriorQueue.size();
```

```
    printf("2 Розмір другої черги =  %d\n", QueueSize);

    // Виводимо черга на екран
    printf("Перша черга\n");
    print(priorQueue);

    printf("Друга черга\n");
    print(newPriorQueue);

    return 0;
}

void print(PriorityQueue &priorQueue)
{
    if (priorQueue.empty())
    {
        printf("%s \n","Черга порожня");
        return;
    }

    while(!priorQueue.empty())
    {
         printf("%d ", priorQueue.top());
         priorQueue.pop();
    }
         printf("\n") ;
}
```

Результат

```
Порядок заповнення
41 18467 6334 26500 19169
15724 11478 29358 26962 24464
Розмір першої черги =  5
Розмір другої черги =  5
Перша черга
26500 19169 18467 6334 41
Друга черга
29358 26962 24464 15724 11478
```

Примітка. Незважаючи на те що порядок заповнення черги із пріоритетом був випадковим, вивід елементів із черги відбувається в порядку, певному щаблонным параметром `less<int>`, тобто в порядку зростання.

13.4. Асоціативні контейнери

Асоціативні контейнери — це структури даних, що забезпечують швидкий пошук даних по ключі. У стандартній бібліотеці шаблонів передбачено чотири різновиди асоціативних контейнерів: `set` (множина), `multiset` (мультимножина), `map` (асоціативний масив) і `multimap` (асоціативний мультимассив).

Терміни “мультимножина” й “мультимассив” означають, що ці контейнери допускають дублікати елементів, на відміну від множини й масиву, у яких елементи повинні бути унікальними.

Асоціативні контейнери залежать від двох параметрів — `Key` (ключ) і `Compare` (відношення повного впорядкування по ключі `Key`). Класи `set` й `multiset` являють собою “вырожденные” асоціативні контейнери, у яких значення не важливі — всі операції здійснюються через ключі. Контейнер класу `set` повинен складатися з унікальних ключів, а об'єкт класу `multiset` допускає дублікати. У класах `map` й `multimap` зберігаються пари, що складаються з об'єктів довільного типу `T`, пов'язаних з об'єктами класу `Key`.

Ітератор асоціативного контейнера є двунаправленным. Операція вставки не впливає на коректність ітераторов і посилань на елементи контейнера, а операція видалення анулює лише ітератори й посилання, установлені на вилучені елементи. Крім того, ітератори асоціативних контейнерів переміщаються по контейнері в порядку зростання ключів. Операції, загальні для всіх асоціативних контейнерів, перераховані в табл. 13.3.

Таблиця 13.3. Операції, загальні для всіх асоціативних контейнерів

Конструктори	
`AssociativeContainer()`	Створює порожній контейнер
`AssociativeContainer(const Comparison& c)`	Створює порожній контейнер із зазначеним

	відношенням порівняння
`AssociativeContainer(i,j)`	Створює контейнер, инициализированный діапазоном `[i,j)`
`AssociativeContainer(i,j,c)`	Створює контейнер, инициализированный діапазоном `[i,j)` із зазначеним відношенням порівняння
Доступ до елементів контейнера	
`iterator find(key)` `const_iterator find(key)`	Виконує пошук елемента по заданому ключі. Повертає ітератор, установлений на шуканий елемент
`iterator lower_bound(key)` `const_iterator lower_bound(key)`	Повертає ітератор, установлений на перший елемент, що відповідає заданому ключу
`iterator upper_bound(key)` `const_iterator upper_bound(key)`	Повертає ітератор, установлений на перший елемент, ключ якого більше заданого
`pair<iterator,iterator> equal_range(key)`	Повертає пари ітераторов, установлених на перший й останній елементи, що мають заданий ключ
`size_t count(key)`	Повертає кількість елементів, що мають заданий ключ
Вставка й видалення	
`pair<iterator,bool> insert(t)`	Вставляє елемент `t`. Повертає пару, що складається з ітератора, установленого на вставлений елемент, і логічного значення (`true` — вставка виконана, `false` — вставка не виконана)
`void insert(i,j)`	Вставляє в контейнер діапазон `[i,j)`
`void erase(pos)`	Видаляє елемент, на який посилається ітератор `pos`
`size_t erase(key)`	Видаляє елемент, що має ключ `key`
`void erase(i,j)`	Видаляє діапазон `[i,j)`

Розглянемо стандартні шаблонові класи, що описують властивості асоціативних контейнерів.

13.4.1 Множина

Клас `set`, передбачений стандартом мови C++, являє собою асоціативний контейнер, що містить унікальні ключі. Він допускає застосування ітератора довільного доступу, що посилається на об'єкт класу `T`. Тип `value_type` визначений у класі `set` як тип `key_type`. Це зроблено для того, щоб підкреслити наступність класу `set`, що є вырожденным варіантом асоціативного контейнера, тобто містить лише ключі, а не пари значення/ключ.

За замовчуванням для порівняння ключів використається відношення “менше”. Із цієї причини переміщення ітераторов відбувається в напрямку зростання ключів.

13.4.1.1 Конструктори й деструктор

У класі `set` передбачені наступні конструктори й деструктор.

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

Створює порожня множина.

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

Створює множина, ініціалізуючи його елементами з діапазону, обмеженого ітераторами `first` й `last`.

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

Створює множина, ініціалізуючи його елементами об'єкта `x`.

- `~set();`

Знищує множина.

13.4.1.2 Операція присвоювання

Присвоювання об'єктів класу `set` забезпечується наступної операторной функцією-членом.

- `set<Key,T,Compare,Allocator>& operator=(const set<Key,T,Compare,Allocator>& x);`

Присвоює викликаючому об'єкту елементи множини `x`.

13.4.1.3 Зміна й визначення розміру об'єкта

Маніпуляції з розміром об'єкта класу `set` виконуються наступними функціями-членами.

- `size_type size() const;`

Повертає кількість елементів, що втримуються в множини.

- `size_type max_size() const;`

Повертає максимальна кількість елементів у множини.

- `bool empty() const;`

Якщо об'єкт порожній, повертає значення `true`, якщо немає — значення `false`.

13.4.1.4 Операція доступу

Для доступу до елемента множини передбачений наступна операторная функція-член.

- `operator[](const key_type& x);`

Оператор індексованого доступу до елемента множини.

13.4.1.5 Операції вставки

Вставка елементів в об'єкт класу `set` здійснюється наступними функціями-членами.

- `pair<iterator, bool> insert(const value_type& x);`

Вставляє елемент і повертає пару, що складається з ітератора й булевого значення. Оскільки об'єкт класу `set` не допускає дублікатів, що повертає пара дозволяє визначити, наскільки успішно виконана вставка. Якщо в множини не існувало елемента із зазначеним ключем, булева змінна буде мати значення `true`, а ітератор буде встановлений на вставлений елемент. Якщо вставка не відбулася, вона буде мати значення `false`. Інші варіанти функції `insert()` не дозволяють з'ясувати, чи виконана вставка.

- `iterator insert(iterator position, const value_type& x);`

Вставляє елемент перед елементом, на який посилається ітератор `position`.

- `template <class InputIterator> void insert(InputIterator first, InputIterator last);`

Вставляє діапазон елементів, заданий ітераторами `first` й `last`.

13.4.1.6 Операції видалення

Видалення елементів з об'єкта класу `set` здійснюється наступними функціями-членами.

- `void erase(iterator position);`

Видаляє елемент, на який посилається ітератор `position`.

- `size_type erase(const key_type& x);`

Видаляє елемент по заданому ключі.

- `void erase(iterator first, iterator last);`

Видаляє елементи із зазначеного діапазону.

- `void clear();`

Видаляє всі елементи множини.

- `void swap(map<Key,T,Compare,Allocator>&);`

Міняє місцями елементи викликаючого об'єкта й аргументу.

13.4.1.7 Функторы порівняння

Функторы, що задають правила порівняння елементів об'єкта класу `set`, повертаються наступними функціями.

- `key_compare key_comp() const;`

Повертає функтор, що порівнює ключі.

- `value_compare value_comp()const;`

Повертає функтор, що порівнює значення.

13.4.1.8 Операції пошуку

Пошук елементів в об'єктах класу `set` виконується наступними функціями-членами.

- `const_iterator find(const key_type& x) const;`

Функція-член `find()` повертає ітератор, установлений на шукане значення. Якщо шукане значення не знайдене, вона повертає ітератор, установлений на позамежне значення контейнера. Пошук здійснюється по ключі.

- `size_type count(const key_type& x) const;`

Повертає кількість елементів, що мають заданий ключ.

- `iterator  lower_bound(const key_type& x) const;`

Повертає ітератор, установлений на початок підпослідовності елементів, що мають заданий ключ.

- `iterator  upper_bound(const key_type& x) const;`

Повертає ітератор, установлений на кінець підпослідовності елементів, що мають заданий ключ.

- `pair<iterator,iterator> equal_range(const key_type& x) const;`

Повертає пару, що складається з ітераторов, установлених на початок і кінець послідовності, що складає з елементів із заданим ключем.

13.4.1.9 Операції порівняння

Порівняння елементів в об'єктах класу `set` виконується наступними функціями-членами.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const set<Key,T,Compare, Allocator>& x,
                    const set<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох множин. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з однакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator< (const set<Key,Compare, Allocator>& x,
                    const set<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох множин. Повертає значення `true`, якщо множина `x` менше множини `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator!=(const set<Key,Compare, Allocator>& x,
                    const set<Key,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох множин. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з неоднакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator> (const set<Key,Compare, Allocator>& x,
                    const set<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох множин. Повертає значення `true`, якщо об'єкт `x` більше об'єкта `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator>=(const set<Key,Compare, Allocator>& x,
                    const set<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох множин. Повертає значення `true`, якщо об'єкт `x` більше об'єкта `y` або дорівнює йому. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator<=(const set<Key,Compare, Allocator>& x,
                    const set<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох множин. Повертає значення `true`, якщо об'єкт `x` менше об'єкта `y` або дорівнює йому. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
      void swap(set<Key,Compare,Allocator>& x,
                set<Key,Compare,Allocator>& y);
  ```

Міняє місцями елементи множин `x` й `y`.

Приклад множини

```
#include <iostream>
#include <set>

using namespace std ;

void print(set<int, less<int> > &Set);
void reverse_print(set<int, less<int> > &Set);

int main()
{
   const int SIZE = 5;
   int i;

   set<int, less<int> >::iterator where, newWhere;

   // Порожні множини

   set<int, less<int> > Set, newSet;

   // Заповнюємо множини за допомогою функції-члена insert()
   for (i = 0; i < SIZE; i++)
   {
       Set.insert(i);
       newSet.insert(i+1);
   }

   printf("Множина Set\n");
   print(Set);
   printf("Множина newSet\n");
   print(newSet);

   // Виводимо множини у зворотному порядку
   printf("Множина Set у зворотному порядку\n");
   reverse_print(Set);
   printf("Множина newSet у зворотному порядку\n");
   reverse_print(newSet);


   // Виводимо на печатку розміри множин

   int SetSize = Set.size();
   printf("Розмір можества Set     =  %d\n", SetSize);

   SetSize = newSet.size();
   printf("Розмір множини newSet =  %d\n", SetSize);

   // Виводимо на печатку максимальні розміри множин

   SetSize = Set.max_size();
   printf("Максимальний розмір можества Set      =  %d\n", SetSize);

   SetSize = newSet.max_size();
   printf("Максимальний розмір множини newSet  =  %d\n", SetSize);

   // Виводимо на печатку кількість п'ятірок
```
```

```
    SetSize = Set.count(5);
    printf("Кількість п'ятірок у множини Set     =  %d\n", SetSize);

    SetSize = newSet.count(5);
    printf("Кількість п'ятірок у множини newSet =  %d\n", SetSize);


    // Шукаємо й видаляємо трійку в множини Set
    where = Set.find(3);
    Set.erase(where);
    printf("Множина Set після видалення трійки\n");
    print(Set);

    // Видаляємо два останніх елементи множини Set
    where = Set.find(2);
    Set.erase(where, Set.end());
    printf("Множина Set після видалення двох останніх елементів\n");
    print(Set);

    // Спустошуємо множина Set за допомогою функції clear()
    Set.clear();
    printf("Спроба вивести порожню множина\n");
    print(Set);

    // Заповнюємо множина Set за допомогою функції-члена insert().
    // Дублікати не допускаються!
    for (i = 0; i < SIZE; i++)
    {
        if(i<2) Set.insert(5); else    Set.insert(10);
    }

    printf("Нова множина Set\n");
    print(Set);

    // Перестановка множин
    swap(Set, newSet);
    printf("Множина Set після обміну\n");
    print(Set);
    printf("Множина newSet після обміну\n");
    print(newSet);

    // Порівняння множин

    if(Set == newSet) printf("Set == newSet\n");
    if(Set >  newSet) printf("Set >  newSet\n");
    if(Set >= newSet) printf("Set >= newSet\n");
    if(Set <  newSet) printf("Set <  newSet\n");
    if(Set <= newSet) printf("Set <= newSet\n");

    return 0;
}

void print(set<int, less<int> > &Set)
{
    if (Set.empty())
    {
        printf("%s \n","Множина порожньо");
        return;
    }

    set<int, less<int> >::iterator i;
    for(i=Set.begin(); i!=Set.end(); i++)
```

```
        printf("%d ",*i);
         printf("\n") ;
}

void reverse_print(set<int, less<int> > &Set)
{
   if (Set.empty())
   {
       printf("%s \n","Множина порожня");
       return;
   }

   set<int, less<int> >::reverse_iterator i;
   for(i=Set.rbegin(); i!=Set.rend(); i++)
       printf("%d ",*i);
        printf("\n") ;
}
```

Результат

```
Множина Set
0 1 2 3 4
Множина newSet
1 2 3 4 5
Множина Set у зворотному порядку
4 3 2 1 0
Множина newSet у зворотному порядку
5 4 3 2 1
Розмір множини Set     =  5
Розмір множини newSet  =  5
Максимальний розмір множини Set      =  1073741823
Максимальний розмір множини newSet   =  1073741823
Кількість п'ятірок у множини Set     =  0
Кількість п'ятірок у множини Set     =  1
Множина Set після видалення трійки
0 1 2 4
Множина Set після видалення двох останніх елементів
0 1
Спроба вивести порожню множина
Множина порожньо
Нова множина Set
5 10
Множина Set після обміну
1 2 3 4 5
Множина newSet після обміну
5 10
Set <  newSet
Set <= newSet
```

13.4.2 Мультимножина

Шаблонний клас `multiset` описує властивості асоціативного контейнера, що дозволяє зберігання однакових ключів, тобто допускающего дублікати. Доступ до елемента контейнера забезпечується ітератором довільного доступу, що посилається на об'єкт класу T. Основні відмінності між контейнерами класів `multiset` й `set` полягають у реалізації функцій-членів `equal_range()`, `lower_bound()` і `upper_bound()`.

13.4.2.1 Конструктори й деструктор

Клас `multiset` містять наступні конструктори й деструктор.

- `explicit multiset(const Compare& comp = Compare(),`
  `const Allocator&= Allocator());`

Створює порожню мультимножину.

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

Створює мультимножину, ініціалізуючи його елементами з діапазону, обмеженого ітераторами `first` й `last`.

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

Створює мультимножину, ініціалізуючи його елементами об'єкта `x`.

- `~multiset();`

Знищує мультимножину.

13.4.2.2 Операція присвоювання

Присвоювання об'єктів класу `multiset` виконується наступним функцією-членом.

- `multiset<Key,T,Compare,Allocator>&`
  `operator=(const multiset<Key,T,Compare,Allocator>& x);`

Присвоює викликаючому об'єкту елементи мультимножини `x`.

13.4.2.3 Зміна й визначення розміру об'єкта

Для маніпуляцій з розмірами об'єктів класу `multiset` призначені наступні функції-члени.

- `size_type size() const;`

Повертає кількість елементів, що втримуються в мультимножине.

- `size_type max_size() const;`

Повертає максимальна кількість елементів у мультимножине.

- `bool empty() const;`

Якщо об'єкт порожній, повертає значення `true`, якщо немає — значення `false`.

13.4.2.4 Операції вставки

Вставка елементів в об'єкт класу `multiset` здійснюється наступними функціями-членами.

- `iterator insert(const value_type& x);`

Вставляє значення `x`.

- `iterator insert(iterator position, const value_type& x);`

Вставляє елемент перед елементом, на який посилається ітератор `position`.

- `template <class InputIterator>`
  `void insert(InputIterator first, InputIterator last);`

Вставляє діапазон елементів, заданий ітераторами `first` й `last`.

13.4.2.5 Операції видалення

Видалення елементів з об'єкта класу `multiset` здійснюється наступними функціями-членами.

- `void erase(iterator position);`

Видаляє елемент, на який посилається ітератор `position`.

- `size_type erase(const key_type& x);`

Видаляє елемент по заданому ключі.

- `void erase(iterator first, iterator last);`

Видаляє елементи із зазначеного діапазону.

- `void clear();`

Видаляє всі елементи мультимножини.

- `void swap(multiset<Key,T,Compare,Allocator>&);`

Міняє місцями елементи викликаючого об'єкта й аргументу.

13.4.2.6 Функторы порівняння

Функторы, що визначають правила порівняння елементів мультимножини, повертаються наступними функціями-членами.

- `key_compare key_comp() const;`

Повертає функтор, що порівнює ключі.

- `value_compare value_comp()const;`

Повертає функтор, що порівнює значення.

13.4.2.7 Операції пошуку

Пошук елементів у мультимножині забезпечується наступними функціями-членами.

- `iterator find(const key_type& x) const;`

Повертає ітератор, установлений на шукане значення. Якщо шукане значення не знайдене, повертає ітератор, установлений на позамежне значення контейнера. Пошук здійснюється по ключі.

- `size_type count(const key_type& x) const;`

Повертає кількість елементів, що мають заданий ключ.

- `iterator  lower_bound(const key_type& x) const;`

Повертає ітератор, установлений на початок підпослідовності елементів, що мають заданий ключ.

- `iterator  upper_bound(const key_type& x) const;`

Повертає ітератор, установлений на кінець підпослідовності елементів, що мають заданий ключ.

- `pair<iterator,iterator> equal_range(const key_type& x) const;`

Повертає пару, що складається з ітераторов, установлених на початок і кінець послідовності, що складає з елементів із заданим ключем.

13.4.2.8 Операції порівняння

Порівняння елементів у мультимножині виконується наступними функціями-членами.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator==(const multiset<Key,Compare, Allocator>& x,
                    const multiset<Key,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох мультимножин. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з однакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator< (const multiset<Key,Compare, Allocator>& x,
                    const multiset<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох мультимножин. Повертає значення `true`, якщо об'єкт `x` менше об'єкта `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator!=(const multiset<Key,Compare, Allocator>& x,
                    const multiset<Key,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох мультимножин. Повертає значення `true`, якщо об'єкти `x` й `y` складаються з неоднакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator> (const multiset<Key,Compare, Allocator>& x,
                    const multiset<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох мультимножин. Повертає значення `true`, якщо об'єкт `x` більше об'єкта `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key,Compare, Allocator>& x,
                    const multiset<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох мультимножин. Повертає значення `true`, якщо об'єкт `x` більше об'єкта `y` або дорівнює йому. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
    bool operator<=(const multiset<Key,Compare, Allocator>& x,
                    const multiset<Key,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох мультимножин. Повертає значення `true`, якщо об'єкт `x` менше об'єкта `y` або дорівнює йому. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class Compare, class Allocator>
     void swap(multiset<Key,Compare,Allocator>& x,
               multiset<Key,Compare,Allocator>& y);
  ```

Міняє місцями елементи мультимножин `x` й `y`.

Приклад мультимножини

```
#include <iostream>
#include <set>

using namespace std ;

void print(multiset<int, less<int> > &Multiset);
void reverse_print(multiset<int, less<int> > &Multiset);

int main()
{
   const int SIZE = 5;
   int i;

   multiset<int, less<int> >::iterator where, newWhere;

   // Порожні множини

   multiset<int, less<int> > Multiset, newMultiset;


   // Заповнюємо множини за допомогою функції-члена insert()
   for (i = 0; i < SIZE; i++)
   {
       Multiset.insert(i);
       newMultiset.insert(i+1);
   }

   printf("Множина Multiset\n");
   print(Multiset);
   printf("Множина newMultiset\n");
   print(newMultiset);

   // Виводимо множини у зворотному порядку
   printf("Множина Multiset у зворотному порядку\n");
   reverse_print(Multiset);
   printf("Множина newMultiset у зворотному порядку\n");
   reverse_print(newMultiset);


   // Виводимо на печатку розміри множин

   int MultisetSize = Multiset.size();
   printf("Розмір можества Multiset     =  %d\n", MultisetSize);

   MultisetSize = newMultiset.size();
   printf("Розмір множини newMultiset =  %d\n", MultisetSize);


   // Виводимо на печатку максимальні розміри множин

   MultisetSize = Multiset.max_size();
   printf("Максимальний розмір можества Multiset      =  %d\n",
            MultisetSize);

   MultisetSize = newMultiset.max_size();
```
```

```
printf("Максимальний розмір множини newMultiset  =  %d\n",
        MultisetSize);

// Виводимо на печатку кількість п'ятірок

MultisetSize = Multiset.count(5);
printf("Кількість п'ятірок у множини Multiset     =  %d\n",
        MultisetSize);

MultisetSize = newMultiset.count(5);
printf("Кількість п'ятірок у множини newMultiset =  %d\n",
        MultisetSize);

// Шукаємо й видаляємо трійку в множини Multiset
where = Multiset.find(3);
Multiset.erase(where);
printf("Множина Multiset після видалення трійки\n");
print(Multiset);

// Видаляємо два останніх елементи множини Multiset
where = Multiset.find(2);
Multiset.erase(where, Multiset.end());
printf("Множина Multiset після видалення двох останніх елементів\n");
print(Multiset);

// Спустошуємо множина Multiset за допомогою функції clear()
Multiset.clear();
printf("Спроба вивести порожню множина\n");
print(Multiset);

// Заповнюємо множина Multiset за допомогою функції-члена insert()
// Дублікати дозволені!
for (i = 0; i < SIZE; i++)
{
    if(i<2) Multiset.insert(5); else    Multiset.insert(10);
}

printf("Нова множина Multiset\n");
print(Multiset);

// Перестановка множин
swap(Multiset, newMultiset);
printf("Множина Multiset після обміну\n");
print(Multiset);
printf("Множина newMultiset після обміну\n");
print(newMultiset);

// Діапазон, що складається з десяток
printf("Діапазон, що складається з десяток\n");
for (where=newMultiset.lower_bound(10);
     where!=newMultiset.end();
     where++)
     printf("%d ",*where);
        printf("\n");

// Діапазон, що складається з п'ятірок
printf("Діапазон, що складається з п'ятірок\n");
 for (where=newMultiset.begin();
      where!= newMultiset.upper_bound(5);
      where++)
      printf("%d ",*where);
         printf("\n");
```

```
    // Діапазон, що складається із сімок
    printf("Діапазон, що складається із сімок\n");
    for (where=newMultiset.equal_range(7).first;
        where!= newMultiset.equal_range(7).second;
         where++)
         printf("%d ",*where);
        printf("\n");

    // Порівняння множин

    if(Multiset == newMultiset) printf("Multiset == newMultiset\n");
    if(Multiset >  newMultiset) printf("Multiset >  newMultiset\n");
    if(Multiset >= newMultiset) printf("Multiset >= newMultiset\n");
    if(Multiset <  newMultiset) printf("Multiset <  newMultiset\n");
    if(Multiset <= newMultiset) printf("Multiset <= newMultiset\n");

    return 0;
 }

 void print(multiset<int, less<int> > &Multiset)
 {
    if (Multiset.empty())
    {
        printf("%s \n","Множина порожньо");
        return;
    }

    multiset<int, less<int> >::iterator i;
    for(i=Multiset.begin(); i!=Multiset.end(); i++)
        printf("%d ",*i);
         printf("\n") ;

 }

 void reverse_print(multiset<int, less<int> > &Multiset)
 {
    if (Multiset.empty())
    {
        printf("%s \n","Множина порожньо");
        return;
    }

    multiset<int, less<int> >::reverse_iterator i;
    for(i=Multiset.rbegin(); i!=Multiset.rend(); i++)
        printf("%d ",*i);
         printf("\n") ;

 }
```

Результат

```
Множина Multiset
 0 1 2 3 4
 Множина newMultiset
 1 2 3 4 5
 Множина Multiset у зворотному порядку
4 3 2 1 0
 Множина newMultiset у зворотному порядку
 5 4 3 2 1
 Розмір множини Multiset     =  5
 Розмір множини newMultiset  =  5
 Максимальний розмір множини Multiset      =  1073741823
 Максимальний розмір множини newMultiset   =  1073741823
 Кількість п'ятірок у множини Multiset     =  0
```

```
 Кількість п'ятірок у множини Multiset     =  1
 Множина Multiset після видалення трійки
0 1 2 4
 Множина Multiset після видалення двох останніх елементів
 0 1
 Спроба вивести порожню множина
 Множина порожньо
Нова множина Multiset
 5 5 7 7 10 10 10 10
Множина Multiset після обміну
1 2 3 4 5
Множина newMultiset після обміну
5 5 7 7 10 10 10 10
Діапазон, що складається з десяток
10 10 10 10
Діапазон, що складається з п'ятірок
5 5
Діапазон, що складається із сімок
7 7
Multiset <  newMultiset
Multiset <= newMultiset
```

13.4.3. Асоціативний масив

Клас `map` описує властивості асоціативного контейнера, що не допускає ключів-дублікатів. Він забезпечує швидкий пошук об'єктів іншого типу `T`, пов'язаних із ключами. На відміну від об'єктів класів `set` й `multiset`, що зберігають ключі, контейнери класів `map` й `multimap` містять пари (значення, ключ). Ключі являють собою константи, тому їх неможливо змінити. Однак через ключі можна одержати доступ до пов'язаним з ними значенням і модифікувати ці значення. Кожен ключ в асоціативному масиві унікальний, але може бути пов'язаний з декількома значеннями. Перебір елементів асоціативного масиву виробляється по зростанню ключів. Доступ забезпечується ітератором довільного доступу.

13.4.3.1.Конструктори й деструктор

Клас `map` містять наступні конструктори й деструктор.

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

Створює порожній асоціативний масив.

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

Створює асоціативний масив, задаючи початок і кінець діапазону, а також функцію порівняння елементів.

- `map<Key,T,Compare,Allocator>&`
  `operator=(const map<Key,T,Compare,Allocator>& x);`

Створює асоціативний масив, ініціалізуючи його елементами масиву `x`.

- `~map();`

Знищує асоціативний масив.

13.4.3.2. Зміна й визначення розміру об'єкта

Для маніпуляцій з розмірами об'єктів класу `map` призначені наступні функції-члени.

- `size_type size() const;`

Визначає кількість елементів в асоціативному масиві.

- `size_type max_size() const;`

Задає максимальна кількість елементів, що зберігаються в масиві.

- `bool empty() const;`

Якщо масив порожній, повертає значення `true`, якщо немає — значення `false`.

13.4.3.3. Операція доступу

Доступ до елементів асоціативного масиву забезпечується наступним функцією-членом.

- `operator[](const key_type& x);`

Оператор доступу до елемента масиву.

13.4.3.4. Операції вставки

Вставка елементів в асоціативний масив виконується наступними функціями-членами.

- `pair<iterator, bool> insert(const value_type& x);`

Вставляє елемент і повертає пару, що складається з ітератора й булевого значення. Оскільки об'єкт класу `map` не допускає дублікатів, що повертає пара дозволяє визначити, наскільки успішно виконана вставка. Якщо в асоціативному масиві не існувало елемента із зазначеним ключем, булева змінна буде мати значення `true`, а ітератор буде встановлений на вставлений елемент. Якщо вставка не відбулася, вона буде мати значення `false`. Інші варіанти функції `insert()` не дозволяють з'ясувати, чи виконана вставка.

- `iterator insert(iterator position, const value_type& x);`

Вставляє елемент перед елементом, на який посилається ітератор `position`.

- ```
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
  ```

Вставляє діапазон елементів, заданий ітераторами `first` й `last`.

13.4.3.5. Операції видалення

Видалення елементів з асоціативного масиву виконується наступними функціями-членами.

- `void erase(iterator position);`

Видаляє елемент, на який посилається ітератор `position`.

- `size_type erase(const key_type& x);`

Видаляє елемент по заданому ключі.

- `void swap(map<Key,T,Compare,Allocator>&);`

Міняє місцями елементи викликаючого масиву й аргументу.

- `void clear();`

Видаляє всі елементи масиву.

13.4.3.6. Функторы порівняння

Функторы, що визначають правила порівняння елементів асоціативного масиву, повертаються наступними функціями-членами.

- `key_compare key_comp() const;`

Повертає функтор, що порівнює ключі.

- `value_compare value_comp()const;`

Повертає функтор, що порівнює значення.

13.4.3.7. Операції пошуку

Пошук елементів в асоціативному масиві здійснюється наступними функціями-членами.

- `iterator find(const key_type& x);`
- `const_iterator find(const key_type& x) const;`

Функція-член `find()` повертає ітератор, установлений на шукане значення. Якщо шукане значення не знайдене, вона повертає ітератор, установлений на позамежне значення контейнера. Пошук здійснюється по ключі.

- `size_type count(const key_type& x) const;`

Повертає кількість елементів, що мають заданий ключ.

- `iterator lower_bound(const key_type& x);`
- `iterator lower_bound(const key_type& x) const;`

Повертає ітератор, установлений на початок підпослідовності елементів, що мають заданий ключ.
  ```

- `iterator upper_bound(const key_type& x);`
- `iterator upper_bound(const key_type& x) const;`

Повертає ітератор, установлений на кінець підпослідовності елементів, що мають заданий ключ.

- `pair<iterator,iterator> equal_range(const key_type& x);`
- `pair<iterator,iterator> equal_range(const key_type& x) const;`

Повертає пару, що складається з ітераторов, установлених на початок і кінець послідовності, що складає з елементів із заданим ключем.

13.4.3.8. Операції порівняння

Порівняння елементів асоціативного масиву виконується наступними функціями-членами.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key,T,Compare, Allocator>& x,
                    const map<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох асоціативних масивів. Повертає значення `true`, якщо масиви `x` й `y` складаються з однакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const map<Key,T,Compare, Allocator>& x,
                    const map<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних масивів. Повертає значення `true`, якщо масив `x` менше масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const map<Key,T,Compare, Allocator>& x,
                    const map<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох асоціативних масивів. Повертає значення `true`, якщо масиви `x` й `y` складаються з неоднакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator> (const map<Key,T,Compare, Allocator>& x,
                    const map<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних масивів. Повертає значення `true`, якщо масив `x` більше масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const map<Key,T,Compare, Allocator>& x,
                    const map<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних масивів. Повертає значення `true`, якщо масив `x` більше або дорівнює масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const map<Key,T,Compare, Allocator>& x,
                    const map<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних масивів. Повертає значення `true`, якщо масив `x` менше або дорівнює масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
     void swap(map<Key,T,Compare,Allocator>& x,
               map<Key,T,Compare,Allocator>& y);
  ```

Міняє місцями елементи масивів `x` й `y`.

Приклад асоціативного масиву

```
#include <iostream>
#include <map>
#include <string>

using namespace std ;

typedef map<int, char*, less<int> > MAP;

void print(MAP &Map);
void reverse_print(MAP &Map);

int main()
{
```
```

```
const int SIZE = 5;
char* NUMBERS[] = {"One", "Two", "Three", "Four", "Five"};
int i;

MAP::iterator where, newWhere;

// Порожні масиви

MAP Map, newMap;

// Заповнюємо масиви за допомогою функції-члена insert()
for (i = 0; i < SIZE; i++)
{
    Map.insert(MAP::value_type(i,NUMBERS[i]));
    newMap.insert(MAP::value_type(i,NUMBERS[i]));
}

printf("Масив Map\n");
print(Map);

printf("Масив newMap\n");
print(newMap);

// Виводимо масиви у зворотному порядку
printf("Масив Map у зворотному порядку\n");
reverse_print(Map);
printf("Масив newMap у зворотному порядку\n");
reverse_print(newMap);

// Виводимо на печатку розміри множин

int MapSize = Map.size();
printf("Розмір масиву Map     =  %d\n", MapSize);

MapSize = newMap.size();
printf("Розмір масиву newMap =  %d\n", MapSize);

// Виводимо на печатку максимальні розміри множин

MapSize = Map.max_size();
printf("Максимальний розмір масиву Map      =  %d\n", MapSize);

MapSize = newMap.max_size();
printf("Максимальний розмір масиву newMap  =  %d\n", MapSize);

// Виводимо на печатку кількість п'ятірок

MapSize = Map.count(5);
printf("Кількість п'ятірок у масиві Map     =  %d\n", MapSize);

MapSize = newMap.count(4);
printf("Кількість четвірок у масиві newMap =  %d\n", MapSize);

// Шукаємо й видаляємо трійку в масиві Map
where = Map.find(3);
Map.erase(where);
printf("Масив Map після видалення трійки\n");
print(Map);

// Видаляємо два останніх елементи масиву Map
where = Map.find(2);
Map.erase(where, Map.end());
printf("Масив Map після видалення двох останніх елементів\n");
```

```
    print(Map);

    // Спустошуємо масив Map за допомогою функції clear()
    Map.clear();
    printf("Спроба вивести порожній масив\n");
    print(Map);

    // Заповнюємо масив Map за допомогою функції-члена insert()
    // Дублікати не допускаються!
    for (i = 0; i < 10; i++)
    {
        if(i<2) Map.insert(MAP::value_type(2,"Two"));
        if(i<2 && i< 5)    Map.insert(MAP::value_type(7,"Seven"));
        if(i>5) Map.insert(MAP::value_type(10,"Ten"));
    }

    printf("Новий масив Map\n");
    print(Map);

    // Перестановка масивів
    swap(Map, newMap);
    printf("Масив Map після обміну\n");
    print(Map);
    printf("Масив newMap після обміну\n");
    print(newMap);

        // Діапазон, що складається з десяток
    printf("Діапазон, що складається з десяток\n");
    for (where=newMap.lower_bound(10); where!=newMap.end();where++)
      printf("(%d, %s) ",(*where).first,(*where).second);
    printf("\n");

    // Діапазон, що складається із двійок
    printf("Діапазон, що складається із двійок\n");
    for (where=newMap.begin(); where!= newMap.upper_bound(2); where++)
      printf("(%d, %s) ",(*where).first,(*where).second);
    printf("\n");

    // Діапазон, що складається із сімок
    printf("Діапазон, що складається із сімок\n");
    for (where=newMap.equal_range(7).first;
         where!= newMap.equal_range(7).second; where++)
      printf("(%d, %s) ",(*where).first,(*where).second);
    printf("\n");

    // Звертання до елементів асоціативного масиву
    // за допомогою функції-члена operator[]()
    printf("Operator[]\n");
    for(where=newMap.begin(); where!=newMap.end(); where++)
    printf("(%d, %s) ",*where, newMap[(*where).first]);
    printf("\n");

    // Порівняння масивів

    if(Map == newMap) printf("Map == newMap\n");
    if(Map >  newMap) printf("Map >  newMap\n");
    if(Map >= newMap) printf("Map >= newMap\n");
    if(Map <  newMap) printf("Map <  newMap\n");
    if(Map <= newMap) printf("Map <= newMap\n");

    return 0;
}
```

```
 void print(MAP &Map)
 {
    if (Map.empty())
    {
        printf("%s \n","Масив порожній");
        return;
    }

    MAP::iterator i;
    for(i=Map.begin(); i!=Map.end(); i++)
         printf("(%d, %s) ",(*i).first,(*i).second);
         printf("\n") ;
 }

 void reverse_print(MAP &Map)
 {
    if (Map.empty())
    {
        printf("%s \n","Масив порожній");
        return;
    }

    MAP::reverse_iterator i;
    for(i=Map.rbegin(); i!=Map.rend(); i++)
        printf("(%d, %s) ",(*i).first,(*i).second);
         printf("\n") ;
 }
```

Результат

```
Масив Map
 (0, One) (1, Two) (2, Three) (3, Four) (4, Five)
 Масив newMap
 (0, One) (1, Two) (2, Three) (3, Four) (4, Five)
 Масив Map у зворотному порядку
 (4, Five) (3, Four) (2, Three) (1, Two) (0, One)
 Масив newMap у зворотному порядку
 (4, Five) (3, Four) (2, Three) (1, Two) (0, One)
 Розмір масиву Map     =  5
 Розмір масиву newMap  =  5
 Максимальний розмір Map      =  1073741823
 Максимальний розмір newMap   =  1073741823
 Кількість п'ятірок у масиві Map     =  0
 Кількість четвірок у масиві newMap =  1
 Множина Map після видалення трійки
(0, One) (1, Two) (2, Three) (4, Five)
 Множина Map після видалення двох останніх елементів
 (0, One) (1, Two)
 Спроба вивести порожній масив
 Масив порожній
 Новий масив Map
 (2, Two) (7, Seven) (10, Ten)
 Масив Map після обміну
 (0, One) (1, Two) (2, Three) (3, Four) (4, Five)
 Масив newMap після обміну
 (2, Two) (7, Seven) (10, Ten)
 Діапазон, що складається з десяток
(10, Ten)
 Діапазон, що складається із двійок
(2, Two)
 Діапазон, що складається із сімок
 (7, Seven)
 Operator[]
 (2, Two) (7, Seven) (10, Ten)
```

```
Map <  newMap
Map <= newMap
```

13.4.4. Асоціативний мультимассив

Клас `multimap` описує властивості асоціативного контейнера, що допускає дублювання ключів. В іншому його функціональні можливості не відрізняються від функціональних можливостей класу `map`.
Визначення шаблонового класу `multimap` у стандарті мови C++ виглядає в такий спосіб.

13.4.4.1. Конструктори й деструктор

Клас `multimap` містять наступні конструктори й деструктор.

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

Створює порожній асоціативний мультимассив.

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

Створює асоціативний мультимассив, задаючи початок і кінець діапазону, а також функцію порівняння елементів.

- `~multimap();`

Знищує асоціативний мультимассив.

13.4.4.2. Операція присвоювання

Присвоювання об'єктів класу `multimap` виконується наступним функцією-членом.

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

Присвоює викликаючий мультимассиву елементи масиву `x`.

13.4.4.3. Зміна й визначення розміру об'єкта

Для маніпуляцій з розмірами об'єктів класу `multimap` призначені наступні функції-члени.

- `size_type size() const;`

Визначає кількість елементів в асоціативному мультимассиве.

- `size_type max_size() const;`

Задає максимальна кількість елементів, що зберігаються в мультимассиве.

- `bool empty() const;`

Якщо мультимассив порожній, повертає значення `true`, якщо немає — значення `false`.

13.4.4.4. Операції вставки

Вставка елементів в об'єкти класу `multimap` виконується наступними функціями-членами.

- `iterator insert(const value_type& x);`

Вставляє елемент і повертає ітератор, установлений на нього.

- `iterator insert(iterator position, const value_type& x);`

Вставляє елемент перед елементом, на який посилається ітератор `position`.

- ```
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
  ```

Вставляє діапазон елементів, заданий ітераторами `first` й `last`.

13.4.4.5. Операції видалення

Видалення елементів з об'єктів класу `multimap` виконується наступними функціями-членами.

- `void erase(iterator position);`

Видаляє елемент, на який посилається ітератор `position`.

- `size_type erase(const key_type& x);`

Видаляє елемент по заданому ключі.

- `void swap(multimap<Key,T,Compare,Allocator>&);`

Міняє місцями елементи викликаючого асоціативного мультимассива й аргументу.

- `void clear();`

Видаляє всі елементи масиву.

13.4.4.6. Функторы порівняння

Функторы, що визначають правила порівняння елементів асоціативного мультимассива, повертаються наступними функціями-членами.

- `key_compare key_comp() const;`

Повертає функтор, що порівнює ключі.

- `value_compare value_comp()const;`

Повертає функтор, що порівнює значення.

13.4.4.7. Операції пошуку

Пошук елементів в асоціативному мультимассиве здійснюється наступними функціями-членами.

- `iterator find(const key_type& x);`
- `const_iterator find(const key_type& x) const;`

функція-член `find()` повертає ітератор, установлений на шукане значення. Якщо шукане значення не знайдене, повертає ітератор, установлений на позамежне значення контейнера. Пошук здійснюється за ключем.

- `size_type count(const key_type& x) const;`

Повертає кількість елементів, що мають заданий ключ.

- `iterator lower_bound(const key_type& x);`
- `iterator lower_bound(const key_type& x) const;`

Повертає ітератор, установлений на початок підпослідовності елементів, що мають заданий ключ.

- `iterator upper_bound(const key_type& x);`
- `iterator upper_bound(const key_type& x) const;`

Повертає ітератор, установлений на кінець підпослідовності елементів, що мають заданий ключ.

- `pair<iterator,iterator> equal_range(const key_type& x);`
- `pair<iterator,iterator> equal_range(const key_type& x) const;`

Повертає пару, що складається з ітераторов, установлених на початок і кінець послідовності, що складає з елементів із заданим ключем.

13.4.4.8. Операції порівняння

Порівняння елементів асоціативного мультимассива виконується наступними функціями-членами.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const multimap<Key,T,Compare, Allocator>& x,
                    const multimap<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох асоціативних мультимассивов. Повертає значення `true`, якщо масиви `x` й `y` складаються з однакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator< (const multimap<Key,T,Compare, Allocator>& x,
                    const multimap<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних мультимассивов. Повертає значення `true`, якщо масив `x` менше масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator!=(const multimap<Key,T,Compare, Allocator>& x,
                    const multimap<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки рівності двох асоціативних мультимассивов. Повертає значення `true`, якщо масиви `x` й `y` складаються з неоднакових елементів. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator> (const multimap<Key,T,Compare, Allocator>& x,
                    const multimap<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних мультимассивов. Повертає значення `true`, якщо масив `x` більше масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator>=(const multimap<Key,T,Compare, Allocator>& x,
                    const multimap<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних мультимассивов. Повертає значення `true`, якщо масив `x` більше або дорівнює масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
    bool operator<=(const multimap<Key,T,Compare, Allocator>& x,
                    const multimap<Key,T,Compare, Allocator>& y);
  ```

Оператор перевірки нерівності двох асоціативних мультимассивов. Повертає значення `true`, якщо масив `x` менше або дорівнює масиву `y`. Операція порівняння задається функтором `Compare`.

- ```
  template <class Key, class T, class Compare, class Allocator>
     void swap(multimap<Key,T,Compare,Allocator>& x,
               multimap<Key,T,Compare,Allocator>& y);
  ```

Міняє місцями елементи масивів `x` й `y`.

Клас `multimap` не підтримує доступ до елементів мультимассива за допомогою оператора `[]`, оскільки наявність декількох однакових ключів створює неоднозначність. Крім того, оскільки асоціативний мультимассив допускає дублікати, контролювати процес вставки не обов'язково. Із цієї причини функція `insert()` завжди повертає ітератор, установлений на вставлений елемент.

Приклад асоціативного мультимассива

```
#include <iostream>
#include <map>

using namespace std ;

typedef multimap<int, char*, less<int> > MULTIMAP;

void print(MULTIMAP &Multimap);
void reverse_print(MULTIMAP &Multimap);

int main()
{
   const int SIZE = 5;
   char* NUMBERS[] = {"One", "Two", "Three", "Four", "Five"};
   int i;

   MULTIMAP::iterator where, newWhere;

   // Порожні множини

   MULTIMAP Multimap, newMultimap;

   // Заповнюємо множини за допомогою функції-члена insert()
   for (i = 0; i < SIZE; i++)
   {
       Multimap.insert(MULTIMAP::value_type(i,NUMBERS[i]));
       newMultimap.insert(MULTIMAP::value_type(i,NUMBERS[i]));
   }

   printf("Множина Multimap\n");
   print(Multimap);

   printf("Множина newMultimap\n");
   print(newMultimap);

   // Виводимо множини у зворотному порядку
   printf("Множина Multimap у зворотному порядку\n");
   reverse_print(Multimap);
   printf("Множина newMultimap у зворотному порядку\n");
   reverse_print(newMultimap);
```
```

```
// Виводимо на печатку розміри множин

int MultimapSize = Multimap.size();
printf("Розмір можества Multimap     =  %d\n", MultimapSize);

MultimapSize = newMultimap.size();
printf("Розмір множини newMultimap =  %d\n", MultimapSize);


// Виводимо на печатку максимальні розміри множин

MultimapSize = Multimap.max_size();
printf("Максимальний розмір можества Multimap      =  %d\n",
         MultimapSize);

MultimapSize = newMultimap.max_size();
printf("Максимальний розмір множини newMultimap  =  %d\n",
        MultimapSize);

// Виводимо на печатку кількість п'ятірок

MultimapSize = Multimap.count(5);
printf("Кількість п'ятірок у множини Multimap     =  %d\n",
         MultimapSize);

MultimapSize = newMultimap.count(3);
printf("Кількість трійок у множини newMultimap =  %d\n",
         MultimapSize);

// Шукаємо й видаляємо трійку в множини Multimap
where = Multimap.find(3);
Multimap.erase(where);
printf("Множина Multimap після видалення трійки\n");
print(Multimap);

// Видаляємо два останніх елементи множини Multimap
where = Multimap.find(2);
Multimap.erase(where, Multimap.end());
printf("Множина Multimap після видалення двох останніх елементів\n");
print(Multimap);

// Спустошуємо множина Multimap за допомогою функції clear()
Multimap.clear();
printf("Спроба вивести порожню множина\n");
print(Multimap);

// Заповнюємо множина Multimap за допомогою функції-члена insert()
// Дублікати дозволені!
for (i = 0; i < 7; i++)
{
    if(i<2) Multimap.insert(MULTIMAP::value_type(2,"Two"));
    if(i>=2 && i< 5)    Multimap.insert(MULTIMAP::value_type(7,"Seven"));
    if(i>=5) Multimap.insert(MULTIMAP::value_type(10,"Ten"));
}

printf("Нова множина Multimap\n");
print(Multimap);

// Перестановка множин
swap(Multimap, newMultimap);
printf("Множина Multimap після обміну\n");
print(Multimap);
```

```
    printf("Множина newMultimap після обміну\n");
    print(newMultimap);

    // Діапазон, що складається з десяток
    printf("Діапазон, що складається з десяток\n");
    for (where=newMultimap.lower_bound(10); where!=newMultimap.end();where++)
      printf("(%d, %s) ",(*where).first,(*where).second);
    printf("\n");

    // Діапазон, що складається з п'ятірок
    printf("Діапазон, що складається з п'ятірок\n");
    for (where=newMultimap.begin(); where!= newMultimap.upper_bound(2); where++)
      printf("(%d, %s) ",(*where).first,(*where).second);
    printf("\n");

    // Діапазон, що складається із сімок
    printf("Діапазон, що складається із сімок\n");
    for (where=newMultimap.equal_range(7).first;
         where!= newMultimap.equal_range(7).second; where++)
      printf("(%d, %s) ",(*where).first,(*where).second);
    printf("\n");

    // Порівняння множин

    if(Multimap == newMultimap) printf("Multimap == newMultimap\n");
    if(Multimap >  newMultimap) printf("Multimap >  newMultimap\n");
    if(Multimap >= newMultimap) printf("Multimap >= newMultimap\n");
    if(Multimap <  newMultimap) printf("Multimap <  newMultimap\n");
    if(Multimap <= newMultimap) printf("Multimap <= newMultimap\n");

    return 0;
}

 void print(MULTIMAP &Multimap)
 {
    if (Multimap.empty())
    {
        printf("%s \n","Множина порожня");
        return;
    }

    MULTIMAP::iterator i;
    for(i=Multimap.begin(); i!=Multimap.end(); i++)
         printf("(%d, %s) ",(*i).first,(*i).second);
         printf("\n") ;
 }

 void reverse_print(MULTIMAP &Multimap)
 {
    if (Multimap.empty())
    {
        printf("%s \n","Множина порожньо");
        return;
    }

    MULTIMAP::reverse_iterator i;
    for(i=Multimap.rbegin(); i!=Multimap.rend(); i++)
        printf("(%d, %s) ",(*i).first,(*i).second);
         printf("\n") ;
 }
```

Результат

```
Масив Multimap
 (0, One) (1, Two) (2, Three) (3, Four) (4, Five)
 Масив newMultimap
 (0, One) (1, Two) (2, Three) (3, Four) (4, Five)
 Масив Multimap у зворотному порядку
 (4, Five) (3, Four) (2, Three) (1, Two) (0, One)
 Масив newMultimap у зворотному порядку
 (4, Five) (3, Four) (2, Three) (1, Two) (0, One)
 Розмір масиву Multimap     =  5
 Розмір масиву newMultimap  =  5
 Максимальний розмір Multimap      =  1073741823
 Максимальний розмір newMultimap   =  1073741823
 Кількість п'ятірок у масиві Multimap     =  0
 Кількість четвірок у масиві newMultimap =  1
 Множина Multimap після видалення трійки
(0, One) (1, Two) (2, Three) (4, Five)
 Множина Multimap після видалення двох останніх елементів
 (0, One) (1, Two)
 Спроба вивести порожній масив
 Масив порожній
 Новий масив Miltimap
 (2, Two) (2, Two) (7, Seven) (7, Seven) (7, Seven) (10, Ten) (10, Ten)
Масив Multimap після обміну
 (0, One) (1, Two) (2, Three) (3, Four) (4, Five)
 Масив newMultimap після обміну
 (2, Two) (2, Two) (7, Seven) (7, Seven) (7, Seven) (10, Ten) (10, Ten)
 Діапазон, що складається з десяток
(10, Ten) (10, Ten)
 Діапазон, що складається із двійок
(2, Two) (2, Two)
 Діапазон, що складається із сімок
 (7, Seven) (7, Seven) (7, Seven)
 Map <  newMultimap
 Map <= newMultimap
```

13.5 Бітова множина

Контейнер, що містить біти, являє собою бітова множина. Ця множина має функціональні можливості векторів і множин, але не підтримує ітератори. Бітова множина — це компактний масив двійкових значень, що дозволяє виконувати логічні операції над ними.

Шаблонний клас `bitset<N>` описує об'єкт, у якому зберігається `N` біт. Кожен біт дорівнює 0 або 1 і може *перемикатися*. Крім того, кожен біт має *позицію*, задану параметром `pos`.

13.5.1. Конструктори й деструктор

У класі `bitset<N>` визначені наступні конструктори й деструктор.

- `bitset();`

Конструктор без параметрів инициализирует масив битов нулями.

- `bitset(unsigned long val);`

Другий конструктор инициализирует масив бітами целочисленного аргументу. Якщо довжина аргументу менше `N`, інші біти прирівнюються до нуля.

- ```
  template <class char, class traits, class Allocator>
    explicit bitset(
       const basic_string<char, traits, Allocator>& str,
       typename basic_string<char, traits, Allocator>::size_type pos = 0,
       typename basic_string<char, traits, Allocator>::size_type
         n = basic_string<char, traits, Allocator>::npos);
  ```

Заповнює бітовий масив елементами рядка типу `basic_string`, преобразуя символи `'0'` й `'1'` у числа 0 й 1. Якщо рядок містить інші символи, генерується виняткова ситуація `invalid_argument`. Якщо довжина бітового масиву перевершує довжину рядка, генерується виняткова ситуація `out_of_range`.
  ```

13.5.2. Логічні операції

У класі `bitset<N>` передбачені наступні логічні операції.

- `bitset<N>& operator&=(const bitset<N>& rhs);`

Виконує операцію `*this &= rhs` (логічна операція “И”).

- `bitset<N>& operator|=(const bitset<N>& rhs);`

Виконує операцію `*this |= rhs` (логічна операція “АБО”).

- `bitset<N>& operator^=(const bitset<N>& rhs);`

Виконує операцію `*this ^= rhs` (логічна операція “ щовиключає АБО”).

- `bitset<N>& operator<<=(const bitset<N>& rhs);`

Виконує операцію `*this <<= rhs` (зрушення розрядів уліво).

- `bitset<N>& operator>>=(const bitset<N>& rhs);`

Виконує операцію `*this >>= rhs` (зрушення розрядів вправо).

- `bitset<N>& set();`

Установлює всі біти об'єкта `*this` (присвоює їм одиницю).

- `bitset<N>& set(size_t pos, int val = true);`

Виконує операцію `*this[pos] = 1`.

- `bitset<N>& reset();`

Обнуляет всі біти.

- `bitset<N>& reset(size_t pos);`

Обнуляет біт у заданій позиції.

- `bitset<N>& operator~() const;`

Виконує операцію логічного заперечення над всім бітовим масивом.

- `bitset<N>& flip();`

Інвертує всі біти масиву.

- `bitset<N>& flip(size_t pos);`

Інвертує біт у позиції `pos`.

- `bitset<N> operator<<(size_t pos) const;`

Виконує операцію `bitset<N>(*this) >> = pos`.

- `bitset<N> operator>>(size_t pos) const;`

Виконує операцію `bitset<N>(*this) << = pos`.

- ```
  template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&);
  ```

Виконує операцію `bitset<N>(lhs) &= rhs`.

- ```
  template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&);
  ```

Виконує операцію `bitset<N>(lhs) |= rhs`.

- ```
  template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&);
  ```

Виконує операцію `bitset<N>(lhs) ^= rhs`.

13.5.3. Операції перетворення

Перетворення об'єктів класу `bitset<N>` виконується наступними функціями-членами.

- `unsigned long to_ulong() const;`

Перетворить бітовий масив у ціле число типу `long`.

- ```
  template <class char, class traits, class Allocator>
      basic_string<char, traits, Allocator> to_string() const;
  ```

Перетворить бітовий масив у рядок.

13.5.4. Операції пошуку й визначення розміру

Пошук елементів в об'єкті класу `bitset<N>` і визначення його розміру виконуються наступними функціями-членами.

- `size_t count() const;`

Підраховує кількість установлених битов (тобто одиниць) у бітовому масиві.

- `size_t size() const;`

Обчислює розмір бітового масиву.

13.5.5. Операції порівняння

Порівняння елементів об'єкта класу `bitset<N>` забезпечується наступними функціями-членами.

- `bool operator==(const bitset<N>& rhs) const;`

Повертає ненульове значення, якщо кожен біт об'єкта `*this` дорівнює відповідному біту аргументу `rhs`.

- `bool operator!=(const bitset<N>& rhs) const;`

Повертає ненульове значення, якщо `!(*this == rhs)`.

- `bool test(size_t pos) const;`

Перевіряє коректність заданої позиції біта. Якщо позиція виходить за межі припустимого діапазону, генерується виняткова ситуація `out_of_range`. Якщо позиція коректна, повертається аргумент `pos`.

- `bool any() const;`

Перевіряє, чи містить бітовий масив хоча б одну одиницю.

- `bool none() const;`

Повертає значення `true`, якщо бітовий масив не містить ні однієї одиниці.

13.5.6. Операції вставки

Вставка елементів у бітову множина виконується наступними функціями-членами.

- ```
  template <class char, clas traits, size_t N>
    basic_istream<char, traits>&
    operator>>(basic_istream<char, traits>& is, bitset<N>& x);
  ```

Витягає з потоку `is` до `N` символів, зберігаючи їх у тимчасовому рядку `str`, а потім виконує операцію `x = bitset<N>(str)`. Уведення триває, поки не будуть уведені всі `N` символів, не буде виявлений кінець файлу або не зустрінуться символ, відмінний від 0 й 1.

- ```
  template <class char, clas traits, size_t N>
    basic_istream<char, traits>&
    operator<<(basic_istream<char, traits>& os, bitset<N>& x);
  ```

Виконує вставку об'єкта `x.template to_string<char,traits,allocator<char> >` у потік виводу `os`.

Приклад бітової множини

```
#include <iostream>
#include <bitset>
#include <string>

using namespace std ;

void print(bitset<16> &BitSet);

int main()
{
   int i;

   // Порожня бітова множина
   bitset<16> BitSet;

   // Заповнюємо вектор за допомогою функції-члена operator[]()
   for (i = 0; i < BitSet.size(); i++)  BitSet[i]=(bool)(i%2);

   // Ініціалізація бітової множини

   bitset<16> NewSet("0000000011111100");

   // Виводимо бітові множини на екран
   cout << "Бітова множина BitSet" << endl;
   print(BitSet);
```

```
cout << "Бітова множина NewSet" << endl;
print(NewSet);

// Перевіряємо,  чи встановлений у множини BitSet хоча б один біт
if(BitSet.any())
     cout << "У множини BitSet є одиниця" << endl;
else cout << "У множини BitSet всі біти скинуті" << endl;

// Знаходимо перший установлений біт
i=0;
while (!NewSet.test(i)) i++;
cout << "Перша одиниця множини NewSet перебуває в розряді"
      << i << endl;

// Установлюємо біт у першому розряді
NewSet.set(1);
cout <<"Множина NewSet після установки біта в першому розряді"
     << endl;
print(NewSet);

// Скидаємо біт у першому розряді
NewSet.reset(1);
cout <<"Множина NewSet після скидання біта в першому розряді" << endl;
print(NewSet);

// Скидаємо всі встановлені біти
NewSet.reset();
cout <<"Множина NewSet після скидання всіх битов" << endl;
print(NewSet);

// Перемикаємо біт у другому розряді
NewSet.flip(2);
cout <<"Множина NewSet після перемикання біта в другому розряді"
     << endl;
print(NewSet);

// Перемикаємо всі біти
NewSet.flip();
cout <<"Множина NewSet після перемикання всіх битов" << endl;
print(NewSet);

// Підраховуємо кількість установлених битов
cout << "Кількість установлених битов у множини NewSet = "
     << NewSet.count() << endl;

// Перетворимо в довге ціле число
NewSet.reset();
NewSet.set(1);
NewSet.set(2);
print(NewSet);
unsigned long LongValue = NewSet.to_ulong();
cout << "Еквівалент long = " << LongValue << endl;

// Перетворимо в рядок
string s = NewSet.to_string();
cout << "Строковий еквівалент = " << s << endl;

// Поразрядные операції
cout << "  Заперечення " << endl;
print(NewSet);
cout << "~ " << endl;
NewSet=~NewSet;
cout << "----------------" << endl;
```

```
    print(NewSet);
    cout << endl;

    cout << "  Поразрядное И" << endl;
    print(NewSet);
    cout << "&" << endl;
    print(BitSet);
    NewSet=NewSet & BitSet;
    cout << "----------------" << endl;
    print(NewSet);
    cout << endl;

    cout << "  Поразрядное АБО" << endl;
    print(NewSet);
    cout << "|" << endl;
    print(BitSet);
    NewSet=NewSet | BitSet;
    cout << "----------------" << endl;
    print(NewSet);
    cout << endl;

    cout << "  Поразрядное що виключає АБО" << endl;
    print(NewSet);
    cout << "^" << endl;
    print(BitSet);
    NewSet=NewSet ^ BitSet;
    cout << "----------------" << endl;
    print(NewSet);
    cout << endl;

    cout << "  Зрушення вліво на n позицій" << endl;
    int n = 5;
    print(BitSet);
    cout << "<< " << endl;
    BitSet=BitSet << n ;
    cout << "----------------" << endl;
    print(BitSet);
    cout << endl;

    cout << "  Зрушення вправо на n позицій" << endl;
    print(BitSet);
    cout << ">> " << endl;
    BitSet=BitSet >> n ;
    cout << "----------------" << endl;
    print(BitSet);
    cout << endl;

    return 0;
}

void print(bitset<16> &BitSet)
{
    cout << "  ";
    for (int i = BitSet.size()-1; i>=0; i--)
         cout << BitSet[i];
         cout << endl;
}
```

Результат

```
Бітова множина BitSet
 1010101010101010
Бітова множина NewSet
 0000000011111100
```

```
У множини BitSet є одиниця
Перша одиниця множини NewSet перебуває в розряді 2
Множина NewSet після установки біта в першому розряді
 0000000011111110
Множина NewSet після скидання біта в першому розряді
 0000000011111100
Множина NewSet після скидання всіх битов
 0000000000000000
Множина NewSet після перемикання біта в другому розряді
 0000000000000100
Множина NewSet після перемикання всіх битов
 1111111111111011
Кількість установлених битов у множини NewSet = 15
0000000000000110
Еквівалент long = 6
Строковий еквівалент = 0000000000000110
 Заперечення
 0000000000000110
~
 1111111111111001

 Поразрядное И
 1111111111111001
&
 1010101010101010
----------------
 1010101010101000

 Поразрядное АБО
 1010101010101000
|
 1010101010101010
----------------
 1010101010101010

 Поразрядное що виключає АБО
 1010101010101010
^
 1010101010101010
----------------
 0000000000000000

 Зрушення вліво на 5 позицій
 1010101010101010
<<
----------------
 0101010101000000

 Зрушення вправо на 5 позицій
 0101010101000000
>>
----------------
 0000001010101010
```

Відзначимо кілька особливостей, властивому класу `bitset<N>`.

По-перше, при виводі об'єкта класу `bitset<N>` виникають утруднення, оскільки функція `printf()` не має убудованого прапора формату для булевых величин. Окремі булевые величини трактуються як цілі числа й, відповідно, виводяться по форматі `%d`, займаючи 4 або 8 байт, залежно від розрядності операційної системи. Хоча дотепер ми без проблем виводили дані, цей приклад показує обмеженість процедурно-процедурно-орієнтованої системи вводу-виводу, успадкованої від мови C. Із цієї причини для виводу бітової множини застосовується клас `cout` й оператор `<<`, що у даному контексті є оператором вставки даних у потік виводу.

По-друге, оскільки бітові множини записуються у зворотному порядку, функція `print()` виводить біти в природному порядку, так щоб молодші біти перебували праворуч, хоча в самій множини молодші біти зберігаються в перших позиціях.

По-третє, при виконанні перетворення `to_string` передбачається, що результатом є об'єкт стандартного класу `string`, а не вказівник на рядок `char*`.