Лекція 10. Послідовні контейнери STL

1. *Спільні риси та специфічні можливості. Особливості внутрішнього влаштування*
2. *Загальні способи використання послідовних контейнерів.*
3. *Допоміжний «говіркий» клас. Порівняння ефективності.*
4. *Спеціальні операції зі списком.*

У попередній лекції ми обговорювали засадничі концепції побудови бібліотеки STL. Зокрема, йшла мова про контейнери. Одним з уточнень концепції контейнера є *послідовний контейнер*. Втілення концепції уточнює її до кінця: реалізує спільні властивості, додає свої. Тут розглянемо конкретні втілення послідовних контейнерів.

Спільні риси

Спочатку пригадаємо спільні властивості *всіх* контейнерів, а тоді подивимося, які властивості *послідовних* доповнюють цей перелік. На завершення розглянемо особливості кожного з конкретних контейнерів. Як і раніше, в таблицях використаємо умовні позначення для оголошення типу контейнера та його екземплярів.

```
template <typename ValueType> class X { ... };
X<T> u, a;
```

Тоді спільні риси всіх контейнерів можна зобразити таблицею

Позначення	Що повертає	Час виконання

`X<T>::iterator`
`X<T>::const_iterator`
`X<T>::reverse_iterator`
	Тип ітератора, що вказує на T	0

`X<T>::reference` `X<T>::const_reference`	T&	0
`X<T>::value_type`	T	0
`X<T>::size_type`	Тип, здатний відобразити кількість елементів контейнера	0

`X<T> u`
`X<T>()`
`X<T> u(a) або X<T> u=a`
	конструктори повертають новостворений контейнер	const
		const
		O(n)

`u.begin()` `u.end()`	`iterator`	const
`u.size()`	`size_t n`	const
`u.swap(a)`	`void`	const
`u==a або u!=a`	`bool`	O(n)

Кожен контейнер має методи *begin* та *end*, тому ми не дублюватимемо їх у наступних таблицях. Просто пам'ятатимемо, що в кожного послідовного контейнера вони є. Давайте уточнимо, що вони повертають. З першим все досить просто: метод *begin* повертає ітератор на перший елемент контейнера. Наприклад, якщо є такі оголошення:

```
vector<int> A {10, 20, 30};
vector<int>::iterator it = A.begin();
```

то **it* є першим елементом вектора *A*, **it* == 10, а присвоєння **it* = 15 змінить *A*[0] – воно стане рівне 15.

Метод *end* повертає ітератор на «елемент, наступний після останнього». Ці дивні слова справді мають сенс. У векторній пам'яті кінець одного елемента є початком наступного. Якщо

до вказівника на останній елемент масиву додати 1, отримаємо вказівник на кінець масиву, або на кінець останнього елемента, або на початок наступного після останнього, якби він там був. У світі ітераторів «наступний після останнього» – це нормально, це така узагальнена назва для закінчення останнього елемента контейнера, або спосіб завжди говорити про початок, а не про кінець.

У бібліотеці STL пару ітераторів *vector<int>::iterator first = A.begin(), last = A.end();* використовують для задання інтервалів [*first*; *last*) – початок належить до інтервалу, а закінчення – ні. З такими інтервалами працюють конструктори, методи, узагальнені алгоритми.

Послідовні контейнери (*vector*, *deque*, *list*, *forward_list*) мають додаткові можливості, характерні саме для послідовностей. Наприклад, конструктори дають змогу наповнити новостворений контейнер вказаною послідовністю значень. Кожна послідовність має початок і кінець, тому послідовні контейнери мають методи дописування на початок і в кінець: *push_front* і *push_back* відповідно; послідовність фіксує порядок своїх елементів, тому в параметрах методів вставляння і вилучення вказують місце (в контейнері) виконання дії. Спільні риси послідовних контейнерів (притаманні хоча б трьом з них) відображено в наступній таблиці. У ній використано такі позначення:

```
X – тип контейнера, наприклад X == vector<T>
T t; size_t n; // значення для занесення в контейнер; розмір контейнера
X::iterator p, q, i, j;
// ітератори на цей або інший контейнер, задають інтервал [i; j)
```

Метод	Тип	vector	deque	list	forward_list
Конструктори					
`X a(n,t); X (n,t);` `X a(i,j);`	Новостворений контейнер	+	+	+	+
Модифікація					
`a.push_front(t)` `a.pop_front()`	void	–	+	+	+
`a.push_back(t)` `a.pop_back()`	void	+	+	+	–

`a.assign(i,j)`
`a.assign(n,t)`
`a.assign(init_list)`
	void	+	+	+	+

Конструювання					
`a.emplace_front(t)`	void	–	+	+	+
`a.emplace(p, t)`	iterator	+	+	+	`emplace_after`
`a.emplace_back(t)`	void	+	+	+	–
Вставляння					

`a.insert(p,t)`
`a.insert(p,n,t)`
`a.insert(p,i,j)`
	iterator
	void
	void
		+	+	+	`insert_after`

Вилучення, очищення					

`a.erase(p)`
`a.erase(p,q)`
`a.clear()`
	iterator
	iterator
	void
		+	+	+	`erase_after(p)`
					`erase_after(p,q)`
					+

Доступ					
`a.front()` `a.back()`	T&	+	+	+	+ –
`a.cbegin()` `a.cend()`	const_iterator	+	+	+	+
`a.rbegin()` `a.rend()`	reverse_iterator	+	+	+	–

Метод	Тип	vector	deque	list	forward_list
`a.crbegin()` `a.crend()`	const_reverse_iterator	+	+	+	–
Розмір					
`a.empty()`	bool	+	+	+	+
`a.maxsize()`	size_t	+	+	+	+
`a.resize(n)` `a.resize(n,t)`	void	+	+	+	+

Мінус в таблиці означає, що контейнер не підтримує вказаного методу. Окремі методи змінили свої назви в новому контейнері *forward_list*: у них з'явився суфікс *_after*.

Наведемо стисле позначення методів:

- Конструктор *X a(n,t);* створює контейнер *a*, заповнений *n* значеннями *t*; конструктор *X a(i,j);* створює контейнер *a*, копію інтервалу [*i*; *j*).
- Методи *a.push_front(t); a.pop_front();* змінюють початок контейнера, а *a.push_back(t); a.pop_back();* – кінець: *push* заносить значення *t* в контейнер і збільшує його розмір, а *pop* – вилучає, розмір зменшує.
- Метод *assign* повністю змінює вміст контейнера. Витирає всі старі елементи і наповнює контейнер значеннями, вказаними в параметрах. Тут *init_list* –ініціалізацтор такий, як використовують для ініціалізації масиву.
- Методи *emplace* – це ефективна заміна методам *push* та *insert*, якщо ви додаєте до контейнера цілком новий об'єкт. Метод *a.push_back(MyHavyObj(params));* викликає два конструктори: з параметрами (для створення тимчасового об'єкта) і копіювання (як звичайно при додаванні елемента до контейнера) та деструктор для знищення тимчасового об'єкта. Натомість *a.emplace_back(params);* викличе лише один конструктор – з параметрами. Можна сказати, що методи *emplace* конструюють об'єкти «на місці». Розмір контейнера збільшується.
- Методи *insert* вставляють одне значення, або *n* значень, або копію вказаного діапазона на вказане ітератором місце. Розмір контейнера збільшується.
- Методи *erase* вилучають вказаний ітератором елемент, або діапазон, заданий парою ітераторів. Розмір контейнера зменшується. Метод *clear* вилучає всі елементи контейнера. У всіх випадках вилучення для елементів автоматично викликаються деструктори (якщо елементи – об'єкти).
- Методи *a.front(), a.back()* повертають посилання на перший і останній елементи контейнера відповідно. Корисні, якщо потрібно прочитати-змінити виключно один – перший чи останній – елементи контейнера. Наприклад, *a.front() = 5;* заміняє перший елемент контейнера на 5.
- На додачу до методів *begin(), end()* є ще шість, що повертають ітератори: *cbegin(), cend()* – константні, що не дозволяють змінювати вміст контейнера; *rbegin(), rend()* – зворотні, що дозволяють перебирати контейнер задом наперед; *crbegin(), crend()* – константні зворотні.
- Метод *empty()* відповість, чи правда, що контейнер не містить ні одного елемента.
- Результат виклику методу *maxsize()* найбільше залежить від технічних параметрів вашого комп'ютера і повідомляє, якого найбільшого розміру міг би досягти контейнер (якщо ніякі інші сутності програми не споживатимуть пам'ять).
- У будь-який момент виконання програми можна налаштувати розмір послідовного контейнера за допомогою методу *resize*. Якщо новий розмір більший від поточного, то контейнер буде доповнений з кінця елементами зі значенням за замовчуванням, або заданими значеннями *t*. Якщо ж новий розмір менший, то зайві з кінця елементи буде вилучено.

Така велика кількість спільних рис послідовних контейнерів означає, що в багатьох випадках можна писати програми не турбуючись про те, який конкретно послідовний контейнер обрати: може підійти будь-який. Зазвичай так і роблять. Починають, наприклад, з улюбленого вектора, а згодом після тестувань і експериментів з оцінкою швидкодії заміняють його іншим, що демонструє кращі результати (або ні).

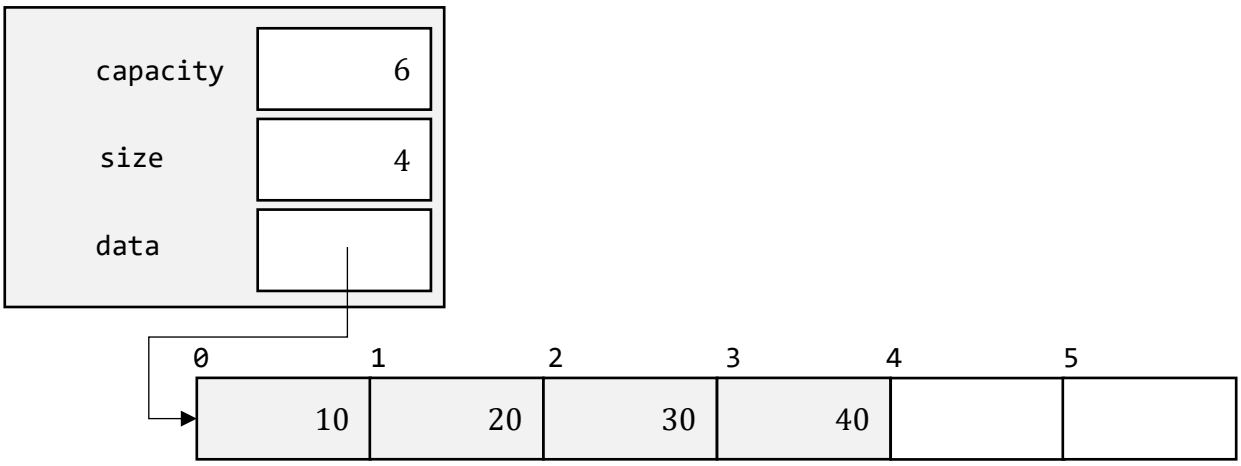
Відмінності в можливостях послідовних контейнерів зумовлена їхнім внутрішнім влаштуванням. Що ж, спробуємо заглянути всередину.

Особливості послідовних контейнерів

Вектор

https://docs.microsoft.com/en-us/cpp/standard-library/vector-class?view=vs-2019

Клас інкапсулює динамічний масив. Схематично його можна зобразити так:



Швидке дописування нових елементів можливе на вільне місце в кінці ділянки *data*. Коли воно вичерпується, контейнер мусить виділити нову ділянку більшого розміру (вдвічі більшу), перекопіювати туди попередній вміст, а тоді дописати нове значення. Тому, в загальному випадку, час дописування елемента *O(size)*. Тому для зменшення затрат на розподіл пам'яті стараються створювати вектор потрібного розміру. Так само переміщень елементів потребує вставляння значення в середину вектора. Ці проблеми і особливості влаштування нам відомі, бо ми вже робили власноруч схожий контейнер.

Звісно, не буде катастрофою, якщо не задати наперед розмір вектора. Він може автоматично збільшуватися в результаті вставляння (чи дописування) елементів. Просто треба пам'ятати про затрати на виконання цих методів. До речі, метод *push_front* не реалізовано для вектора, оскільки він завжди гарантовано потребує *size* переміщень елементів. Якщо дуже треба вставити значення на початок вектора, доводиться використовувати метод *insert* з ітератором на початок контейнера.

Вектор має декілька додаткових методів. Безперечною його перевагою є наявність оператора (і методу) індексування.

```
reference operator[](size_type position);
const_reference operator[](size_type position) const;

reference at(size_type position);
const_reference at(size_type position) const;
```

Оператор зручніший в написанні і швидший у виконанні, але, якщо *position* ≥ *size*, то результат невизначений. Метод *at* надійніший, оскільки перевіряє значення *position* і сигналізує винятком про помилку.

Ще декілька методів допомагають налаштовувати реалізацію (динамічний масив).

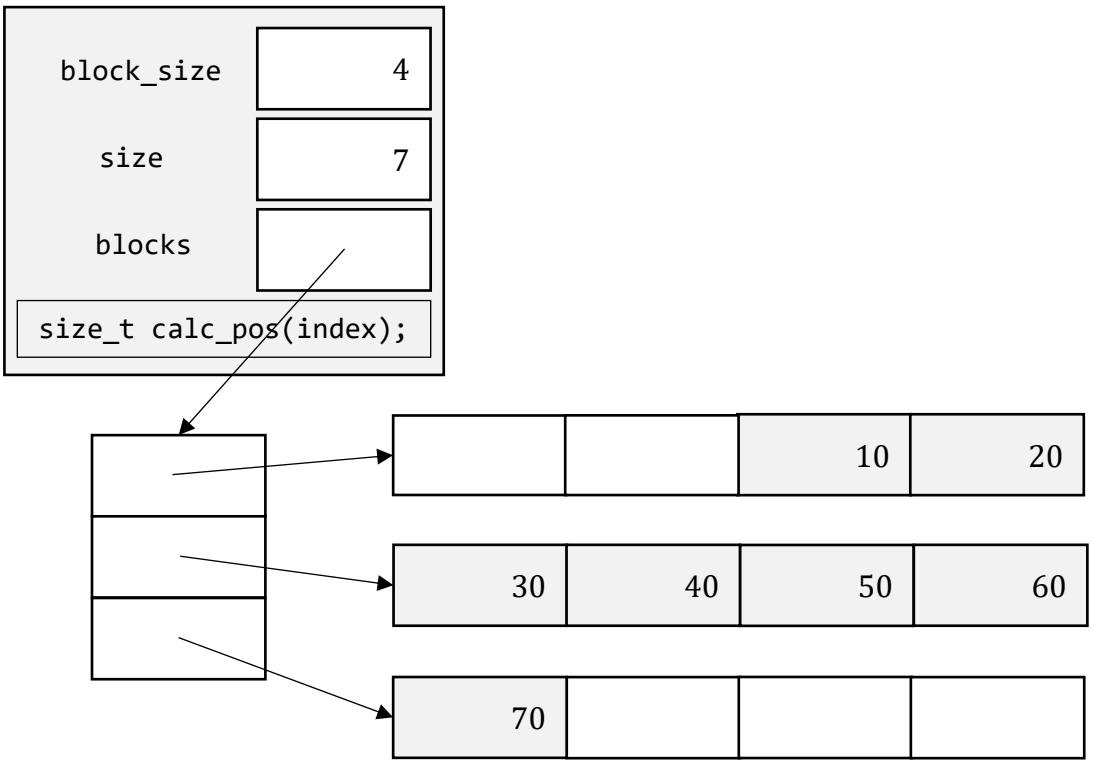
```
// повідомляє розмір зарезервованої ділянки
size_type capacity() const;

// виділяє ділянку розміром count, змінює capacity, не впливає на size
void reserve(size_type count);

// звільняє надлишок зарезервованої пам'яті, залишає лише зайняті елементи
// є сенс викликати тоді, коли вектор перестав зростати
void shrink_to_fit();

// повертає вказівник на перший елемент вектора (на динамічний масив)
// стане в пригоді, коли виникне потреба доступатися до елементів вектора
// через вказівники, а не через ітератори чи індекси (але, коли таке буде треба?)
const_pointer data() const;
pointer data();
```

Дек

https://docs.microsoft.com/en-us/cpp/standard-library/deque-class?view=vs-2019

У попередній лекції вже були висловлені застереження щодо назви цього контейнера. Вона не має нічого спільного з класичною двосторонньою чергою, бо в класичному деку доступ до елементів дозволено *лише* з кінців, а у дека STL доступитися можна до *будь-якого* елемента, як у вектора. Мабуть, *deque* можна розуміти як *vector*, для якого дозволено *push_front* і *pop_front*. Покликання *deque* – зберегти швидкість індексування елементів, як у вектора, і позбутися його затрат на вставляння значень.

Внутрішнє влаштування дека складніше. Він зберігає послідовність блоків векторної пам'яті. Дуже схематично його можна зобразити так (показано частину реалізації, розмір блока 4 вибрано умовно):



За такої реалізації вільне місце залишається і в кінці, і на початку послідовності. Якщо пам'яті бракує, достатньо виділити ще один блок (перебудовується пам'ять вказівників на блоки, а самі блоки – ні). Для вставляння всередину послідовності достатньо зсунути меншу частину послідовності до кінця чи до початку. Час вставляння залишиться лінійним, але вдвічі меншим, ніж для вектора.

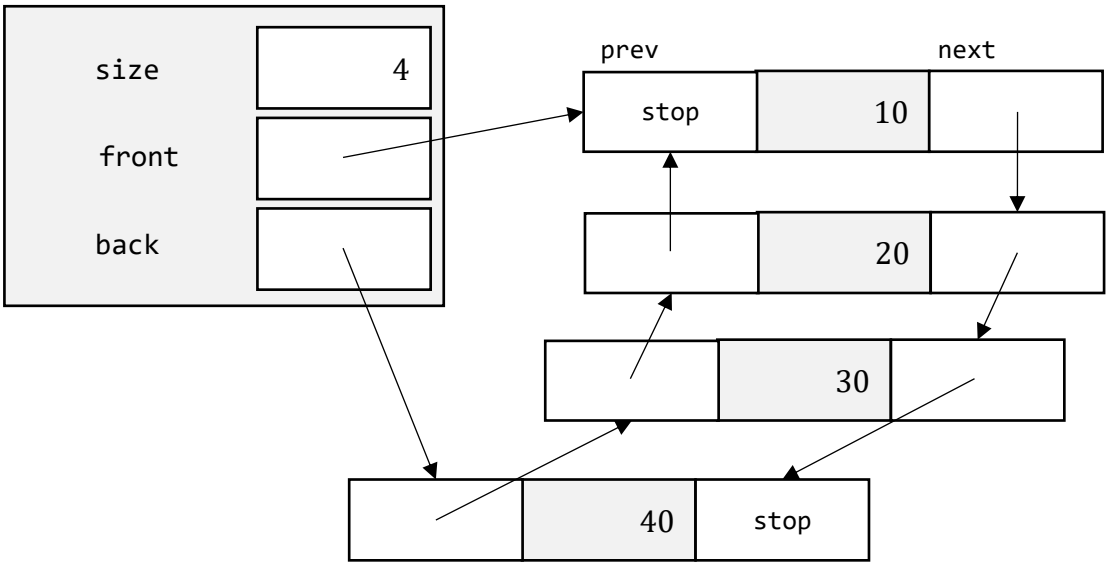
Дек, як і вектор, має додаткові методи

```
reference operator[](size_type position);
const_reference operator[](size_type position) const;

reference at(size_type position);
const_reference at(size_type position) const;

void shrink_to_fit();
```

Двозв'язний список

https://docs.microsoft.com/en-us/cpp/standard-library/list-class?view=vs-2019

Контейнер *list* поміщає свої елементи у ланки двозв'язного списку, тому не потребує виділення пам'яті «про запас». Завдяки вказівникам на першу й останню ланки списку його можна перебирати в обох напрямках. Вставляння нового елемента можливе за константний час в будь-якому місці списку, на яке встановлено ітератор. Розплатою за гнучкість структури є додаткові витрати пам'яті: поруч з кожним елементом зберігаються ще два вказівники на сусідні ланки. Як відомо, список є структурою *послідовного* доступу: щоб отримати значення *k*-го елемента, потрібно перебрати всі попередні $k - 1$. Тому контейнер *list* не підтримує індексування своїх елементів і не містить методу *at* чи *operator[ ]*.



Серед послідовних контейнерів список, мабуть, найбагатший на додаткові можливості. Його методи вміють виконувати роботу загальних алгоритмів з більшою ефективністю, оскільки мають доступ до структури списку. Наприклад, метод *reverse* міняє порядок елементів списку за допомогою обміну лише двох вказівників (на першу й останню ланку), тоді як алгоритм *reverse* обмінює місцями кожну пару відповідних елементів контейнера.

Отож перелічимо ці методи:

- обертання списку;
  ```
  void reverse();
  ```
- вилучення елементів, що рівні заданому значенню, або задовольняють задану умову:
  ```
  void remove(const Type& val);

  template <class Predicate> void remove_if(Predicate pred)
  ```
- вставляння елементів одного списку у вказане місце іншого (самі елементи не змінюють свого розташування в пам'яті, списки тільки обмінюються вказівниками);
  ```
  // insert the entire source list
  void splice(const_iterator Where, list<Type, Allocator>& Source);
  ```

```
// insert one element of the source list
void splice(const_iterator Where,
            list<Type, Allocator>& Source, const_iterator Iter);
// insert a range of elements from the source list
void splice(const_iterator Where, list<Type, Allocator>& Source,
                                  const_iterator First, const_iterator Last);
```

- впорядкування за зростанням, шаблонний метод для порівняння елементів замість оператора порівняння використовує бінарний предикат-компаратор;

```
void sort();

template <class Traits> void sort(Traits comp);
```

З упорядкованими списками можна виконувати корисні операції:

- об'єднання (злиття) двох упорядкованих списків в один, список-аргумент методу передає всі свої елементи списку-отримувачу повідомлення;

```
void merge(list<Type, Allocator>& right);

template <class Traits> void merge(list<Type, Allocator>& right, Traits comp);
```

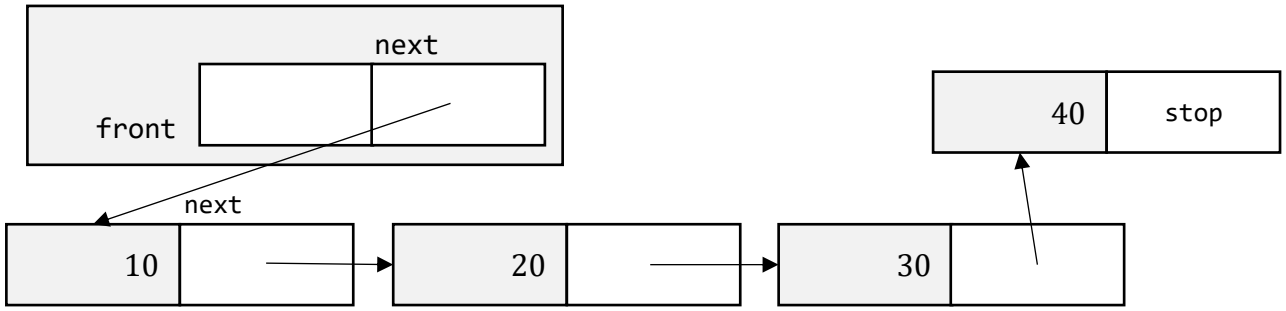
- вилучення повторних входжень однакових елементів упорядкованого списку (замість групи однакових сусідів залишається один елемент);

```
void unique();

template <class BinaryPredicate> void unique(BinaryPredicate pred);
```

Однозв'язний список

https://docs.microsoft.com/en-us/cpp/standard-library/forward-list-class?view=vs-2019

Контейнер *forward_list* з'явився в бібліотеці порівняно недавно. Має більшість можливостей контейнера *list* і використовує удвічі менше додаткової пам'яті, оскільки кожна ланка списку зберігає тільки один вказівник – на наступну ланку. Розробники так турбувалися про швидкодію контейнера, що вважали за краще вилучити з нього підтримку обчислення кількості елементів. Він, чи не єдиний з контейнерів, не має методу *size*.



Зрозуміло, що завдяки такому влаштуванню однозв'язний список не можна перебирати в зворотному порядку. Назви декількох методів отримали суфікс *_after*, який підкреслює ту обставину, що зміни відбуватимуться з тією ланкою, що розташована *після* позначеної ітератором. Наприклад, щоб вилучити ланку зі значенням 20, потрібно мати ітератор на ланку зі значенням 10 – на попередню.

Додаткові можливості *forward_list* такі ж, як у *list*, тільки метод *splice* замінено методом *splice_after*. Також є два додаткових методи, що повертають ітератори на ланку, яка передує першій ланці списку (на заголовну ланку):

```
iterator before_begin();

const_iterator cbefore_begin() const;
```

Далі розглянемо приклади використання послідовних контейнерів.

Загальний спосіб використання *vector, list, deque*

Послідовні контейнери такі схожі за своїми можливостями, що базові операції з ними можна описати єдиним шаблоном. Знайдіть у тексті функції *basicOps* усі конструктори, оператори присвоєння, випадки використання ітераторів, методів дописування, вставляння, присвоєння тощо. Запустіть програму на виконання і поясніть для себе отримані результати.

файл BasicOps.cpp

```
#include <iostream>
#include <vector>
#include <deque>
#include <list>
using namespace std;

// Шаблон функції для виведення повної інформації про довільний
// контейнер (у цій програмі - послідовний)
template<typename Container>
void print(Container& c, char* title = "")
{
	cout << title << ':' << endl;
	if (c.empty())
	{
		cout << "(empty)" << endl;
		return;
	}
	// typename - обов'язкова підказка компіляторові рпо те,
	// що за іменем Container::iterator ховається саме тип,
	// а не статичний член класу
	typename Container::iterator it;
	for (it = c.begin(); it != c.end(); ++it)
		cout << *it << " ";
	cout << endl;
	cout << "size() " << c.size() << " max_size() "<< c.max_size()
		<< " front() " << c.front() << " back() " << c.back() << endl;
	cin.get();
}

// Шаблон функції, що демонструє використання базових можливостей
// довільного послідовного контейнера
template<typename ContainerOfInt>
void basicOps(char* s)
{
	cout << "-------- " << s << " --------" << endl;
	typedef ContainerOfInt Ci;
	Ci c;
	print(c, "c after default constructor");
	Ci c1{ 10, 20, 30, 45 }; // безпосередня ініціалізація
	print(c1, "c1 after initialization");
	typename Ci::iterator iter = c1.begin();
	c1.emplace(++iter, 13);
	print(c1, "c1 after emplace");
	Ci c2(10, 1);          // 10 елементів = 1
	print(c2, "c2 after constructor(10,1)");
	int ia[] = { 1, 3, 5, 7, 9 };
	const int iasz = sizeof(ia)/sizeof(*ia);
// Ініціалізація за допомогою початкового та кінцевого ітераторів
	Ci c3(ia, ia + iasz);
```

```
	print(c3, "c3 after constructor(iter,iter)");
	Ci c4(c2);              // Конструктор копіювання
	print(c4, "c4 after copy-constructor(c2)");
	c = c2;                 // Оператор присвоєння
	print(c, "c after operator=c2");
	c.assign(10, 2);        // 10 елементів зі значенням 2
	print(c, "c after assign(10,2)");
// Присвоєння за допомогою ітераторів
	c.assign(ia, ia + iasz);
	print(c, "c after assign(iter,iter)");
	cout << "c using reverse iterators:" << endl;
	typename Ci::reverse_iterator rit = c.rbegin();
	while (rit != c.rend()) cout << *rit++ << " ";
	cout << endl;
	c.resize(4);
	print(c, "c after resize(4)");
	c.push_back(47);
	print(c, "c after push_back(47)");
	c.pop_back();
	print(c, "c after pop_back()");
	typename Ci::iterator it = c.begin();
	++it; ++it;
	c.insert(it,74);
	print(c, "c after insert(it,74)");
	it = c.begin(); ++it;
	c.insert(it,3,96);
	print(c, "c after insert(it,3,96)");
	it = c.begin(); ++it;
	c.insert(it,c3.begin(),c3.end());
	print(c, "c after c.insert(it,c3.begin(),c3.end())");
	it = c.begin(); ++it;
	c.erase(it);
	print(c, "c after erase(it)");
	typename Ci::iterator it2 = it = c.begin();
	++it;
	++it2; ++it2; ++it2; ++it2; ++it2;
	c.erase(it,it2);
	print(c, "c after erase(it,it2)");
	c.swap(c2);
	print(c, "c after swap(c2)");
	c.clear();
	print(c, "c after clear()");
}

int main()
{
	// при виклику потрібно явно вказувати тип спеціалізації
	basicOps< vector<int> >("vector");
	basicOps< deque<int> >("deque");
	basicOps< list<int> >("list");
	return 0;
}
```

Допоміжний клас *Noisy*

Для того, щоб краще зрозуміти, як працюють контейнери, стане в пригоді спеціальний клас, у якого кожен конструктор, деструктор, оператор присвоєння виводять на консоль

повідомлення про своє виконання. Екземпляри можна відрізнити за їхніми ідентифікаційними номерами. Джерелом номерів є внутрішній лічильник створених об'єктів або значення, задані користувачем у конструкторах з параметрами. Об'єкти *Noisy* можна порівнювати і виводити в потік. Ніякої іншої функціональності клас не надає, але вона і не потрібна. Екземпляри такого класу допоможуть нам простежити, як саме вектор і дек виконують перебудову пам'яті, як функціонують списки тощо.

файл noisy.h

```
#include <iostream>
using std::cout;
using std::ostream;

// - - - клас для моделювання елементів контейнера
// Конструктори, деструктор, оператор присвоєння виводять на консоль
// повідомлення про свою роботу для того, щоб користувач міг
// спостерігати за процесом створення/знищення екземплярів контейнера

class Noisy
{
	// лічильники виконаних дій з екземплярами
	static long create, assign, copy, destroy;
	long id;
public:
	// конструктор за замовчуванням
	Noisy() : id(create)
	{
		cout << "default: [ "<< id << " ]\n";
		++create;
	}
	// конструктор з одним параметром
	Noisy(int k) : id(k)
	{
		cout << "one par: [ " << id << " ]\n";
		++create;
	}
	// конструктор з двома параметрами
	Noisy(int a, int b) : id(a * 100 + b)
	{
		cout << "two par: [ " << id << " ]\n";
		++create;
	}
	// конструктор копіювання
	Noisy(const Noisy& r) : id(r.id)
	{
		cout << "   copy: [ " << id << " ]\n";
		++copy;
	}
	// оператор присвоєння
	Noisy& operator=(const Noisy& r)
	{
		cout << " assign: ( " << id << " ) = [ " << r.id << " ]\n";
		id = r.id;
		++assign;
		return *this;
	}
	// деструктор
	~Noisy()
	{
```

```
            cout << "destruct~[ " << id << " ]\n";
            ++destroy;
      }
      friend bool operator<(const Noisy& a, const Noisy& b)
      {
            return a.id < b.id;
      }
      friend bool operator==(const Noisy& a, const Noisy& b)
      {
            return a.id == b.id;
      }
      friend ostream& operator<<(ostream& os, const Noisy& r)
      {
            return os << "Noisy_" << r.id;
      }
      friend class NoisyReport;
};

// Тип, до екземпляра якого можна звертатися як до генератора
// екземплярів класу Noisy
struct NoisyGen
{
      Noisy operator()()
      {
            return Noisy();
      }
};

// - - - Синглетний клас для автоматичного виведення статистики
class NoisyReport
{
      static NoisyReport nr;
      NoisyReport() {}
      NoisyReport(const NoisyReport&);
      NoisyReport& operator=(NoisyReport&);
public:
      ~NoisyReport()
      {
            cout<<"\n-----------------------------------------\n"
                  <<"  Noisy creations: "<<Noisy::create
                  <<"\nCopy-Constructions: "<<Noisy::copy
                  <<"\nAssigniments: "<<Noisy::assign
                  <<"\nDestructions: "<<Noisy::destroy<<'\n';
            std::cin.get();
      }
};
```

Допоміжний клас *NoisyReport* має єдиний екземпляр – статичний об'єкт класу. Його деструктор автоматичне виведе статистичну інформацію про клас *Noisy* перед самим завершенням програми. Клас цікавий сам по собі, оскільки є дружнім до *Noisy*, щоб мати доступ до його приватних статичних полів. Він також є прикладом реалізації синглетного класу (синглет, або синглтон, або одинак – один з патернів програмування), але це вже тема окремої розмови.

Щоб зрозуміти, як користуватися цим інструментом, напишемо невелику тестову програму, поки що ніяк не пов'язану з контейнерами, і порівняємо її текст з отриманим виведенням. Для зручності обидва тексти наведено в сусідніх стовпцях.

файл testNoisy.cpp

```
#include <Windows.h>
#include "noisy.h"
using std::cin;
// визначення статичних змінних
long Noisy::create = 0, Noisy::assign = 0,
     Noisy::copy = 0, Noisy::destroy = 0;
NoisyReport NoisyReport::nr;

int main()
{
       SetConsoleOutputCP(1251);
       cout << "Створення: конструктори за"
" замовчуванням\n   звичайні змінні\n";
       Noisy a, b, c;
       cin.get();
       cout << "   масив\n";
       Noisy N[2];
       cin.get();
       cout << "Виведення створених об'єктів:\n"
           << '\t' << a << '\t' << b << '\t' << c
       << "\n\t" << N[0] << '\t' << N[1] << '\n';
       cin.get();
       cout << "Динамічний об'єкт: ";
       Noisy * p = new Noisy();
       cin.get();
       cout << "його значення:" << *p
              << "\nA тепер - знищення: ";
       delete p;
       cin.get();
       cout
        << "Створення: конструктори копіювання\n";
       Noisy f(a); Noisy g = b;
       cin.get();
       cout << "Оператор присвоєння:\n";
       N[0] = f; N[1] = b;
       cout << "\nоновлений масив\t"
            << N[0] << '\t' << N[1];
       cin.get();
       cout <<
        "Створення: конструктори з параметрами\n";
       Noisy o(17), t(5, 3);
       cout << "\nстворені об'єкти\t" << o
            << '\t' << t << '\n';
       cin.get();
       cout << "Завершення\n";
       return 0;
}
```

```
Створення: конструктори за замовчуванням
   звичайні змінні
default: [ 0 ]
default: [ 1 ]
default: [ 2 ]

   масив
default: [ 3 ]
default: [ 4 ]

Виведення створених об'єктів:
        Noisy_0 Noisy_1 Noisy_2
        Noisy_3 Noisy_4

Динамічний об'єкт: default: [ 5 ]

його значення:Noisy_5
A тепер - знищення: destruct~[ 5 ]

Створення: конструктори копіювання
   copy: [ 0 ]
   copy: [ 1 ]

Оператор присвоєння:
 assign: ( 3 ) = [ 0 ]
 assign: ( 4 ) = [ 1 ]

оновлений масив Noisy_0 Noisy_1
Створення: конструктори з параметрами
one par: [17 ]
two par: [503 ]

створені об'єкти   Noisy_17   Noisy_503

Завершення
destruct~[ 503 ]
destruct~[ 17 ]
destruct~[ 1 ]
destruct~[ 0 ]
destruct~[ 1 ]
destruct~[ 0 ]
destruct~[ 2 ]
destruct~[ 1 ]
destruct~[ 0 ]

----------------------------------------
  Noisy creations: 8
Copy-Constructions: 2
Assignments: 2
Destructions: 10
```

Вставляння в послідовні контейнери

Тепер дослідимо, як працюють методи *push_back* та *emplace_back*. Для цього вистачить кількох операторів.

```
vector<Noisy> vi, a, b;
a.reserve(5); b.reserve(5);
cout << "\n~~~ Порівняння push_back і emplace_back\n\n";
a.push_back(Noisy()); cout << '\n';
b.emplace_back();
cout << "  .................\n";
a.push_back(Noisy(17)); cout << '\n';
b.emplace_back(17);
cout << "  .................\n";
a.push_back(Noisy(5, 5)); cout << '\n';
b.emplace_back(5, 5);
```

```
	cout << "  .................\n";
	cin.get();
```

Цей фрагмент дасть таке виведення на консоль:

```
~~~ Порівняння push_back і emplace_back

default: [ 0 ]
   copy: [ 0 ]
destruct~[ 0 ]

default: [ 1 ]
  .................
one par: [ 17 ]
   copy: [ 17 ]
destruct~[ 17 ]

one par: [ 17 ]
  .................
two par: [ 505 ]
   copy: [ 505 ]
destruct~[ 505 ]

two par: [ 505 ]
  .................
```

Очевидно, що методи *emplace_back* утричі ефективніший: виклик одного конструктора змість двох конструкторів і деструктора.

Цікаво також спостерігати за перебудовами пам'яті послідовних контейнерів. Для цього потрібні будуть дещо складніші процедури.

```
// Джерело даних для наповнення контейнерів
const int n = 7;
Noisy a[n];

// Дописування в кінець послідовного контейнера
template<class Cont>
void backInsertion(Cont& ci, char* name)
{
	cout << "\n*** BI *** Дописування в кінець " << name << " \n";
	//copy(a, a + n, back_inserter(ci));  // можна використати замість циклу
	for (typename Cont::value_type * p = a; p != a + n; ++p)
	{
		ci.push_back(*p);
		cout << "  .................\n";
	}
	cout<<"\nОтримали:\n";
	copy(ci.begin(), ci.end(),
		ostream_iterator<typename Cont::value_type>(cout, " "));
	cout<<'\n'; cin.get();
}
// Вставляння всередину. Позиція вставляння вибрана довільно.
template<class Cont>
void midInsertion(Cont& ci, char* name)
{
	typename Cont::iterator it = ci.begin();
	++it; ++it; ++it;
	cout<<"\n*** MI *** Вставляння всередину "<<name<<" (після третього)\n";
	copy(a, a + n / 2, inserter(ci, it));
```

```
	cout<<"\nОтримали:\n";
	copy(ci.begin(), ci.end(),
		ostream_iterator<typename Cont::value_type>(cout," "));
	cout<<'\n'; cin.get();
}
```

Викликати їх можна, наприклад, так:

```
	cout<<"\nСтворюємо порожні контейнери\n";
	deque<Noisy> di;
	list<Noisy> li;
	vector<Noisy> vi;
	cout << "\n+++ Дописування в кінець контейнера\n\n";
	backInsertion(vi, "VECTOR");
	backInsertion(di, "DEQUE");
	backInsertion(li, "LIST");
```

Кожен виклик дасть різні результати. Особливо багатослівним він виявиться для вектора. Варто поекспериментувати з програмою, яку ви знайдете у файлі «*Inserters.cpp*». Там є ці та інші процедури вставляння до послідовних контейнерів. Постарайтеся зрозуміти, чому виведення саме таке.

Окремої уваги заслуговує виведення контейнера на консоль, застосоване у наведених вище процедурах. Пригадайте, в попередній лекції ми розбиралися, як оголошують потокові ітератори. Щоб його правильно оголосити, потрібно знати тип даних, призначених до виведення. Шаблон *backInsertion* не знає навіть конкретний тип контейнера, з яким йому доведеться працювати, то звідки ж взяти тип даних, що в ньому зберігаються? Саме для таких потреб всередині кожного контейнера бібліотеки STL оголошено декілька корисних *typedef*. Один з них – *value_type* – означає те, що нам треба, тому виведення на консоль має саме такий вигляд:

```
	copy(ci.begin(), ci.end(),
		ostream_iterator<typename Cont::value_type>(cout, " "));
```

Порівняння ефективності послідовних контейнерів

Давайте влаштуємо змагання на швидкодію між вектором і деком. Списки зачіпати не будемо, бо в них нема засобів індексування елементів. А в цьому змаганні ми перевірятимемо і його. Отже, завдання таке: завантажити до послідовного контейнера рядків вміст достатньо великого текстового файла за допомогою *push_back*. Тоді за допомогою *operator[ ]* змінити кожен рядок контейнера: дописати на початок рядка його порядковий номер. Отриманий змінений текст зберегти в новому файлі, контейнер при цьому перебрати за допомогою ітераторів.

файл Compare.cpp

```
int main()
{
	ifstream in("testV.txt"); // 1.76 MB
	vector<string> vstrings;
	deque<string> dstrings;
	string line;

/* З ТЕКСТУ ПРОГРАМИ ВИЛУЧЕНО РЕЄСТРАЦІЮ ЧАСУ ПОЧАТКУ І ЗАВЕРШЕННЯ РОБОТИ АЛГОРИТМІВ
ЗАРАДИ ЕКОНОМІЇ МІСЦЯ*/
// Завантаження даних до вектора:
	while (getline(in, line)) vstrings.push_back(line);
	cout << "Read into vector: " << elapsed_ms.count() << " ms ellapsed\n";
	in.close();
// Те саме для дека:
```

```
	ifstream in2("testD.txt"); // 1.76 MB
	while (getline(in2, line)) dstrings.push_back(line);
	cout << "Read into deque: " << elapsed_ms.count() << " ms ellapsed\n";
	in2.close();

// Тепер випробуємо індексування для вектора
	for (size_t i = 0; i<vstrings.size(); ++i)
	{
		ostringstream ss;
		ss << i;
		vstrings[i] = ss.str()+": "+vstrings[i];
	}
	cout << "Indexing vector: " << elapsed_ms.count() << " ms ellapsed\n";
// i для дека
	for (size_t j = 0; j<dstrings.size(); ++j)
	{
		ostringstream ss;
		ss << j;
		dstrings[j] = ss.str()+": "+dstrings[j];
	}
	cout << "Indexing deque: " << elapsed_ms.count() << " ms ellapsed\n";

// Порівняння перебору для вектора
	ofstream tmp1("tmp1.txt"), tmp2("tmp2.txt");
	copy(vstrings.begin(), vstrings.end(), ostream_iterator<string>(tmp1, "\n"));
	cout << "Iterating vector: " << elapsed_ms.count() << " ms ellapsed\n";
// i для дека
	copy(dstrings.begin(), dstrings.end(), ostream_iterator<string>(tmp2, "\n"));
	cout << "Iterating deque: " << elapsed_ms.count() << " ms ellapsed\n";
	tmp1.close(); tmp2.close();

// Завантаження в контейнер копіюванням
	ifstream fin("testV.txt");
	istream_iterator<string> FsIt(fin);
	deque<string> newcont(FsIt, istream_iterator<string>());
	cout << "Loading deque by copy: " << elapsed_ms.count() << " ms ellapsed\n";
	fin.close();
	return 0;
}
```

Дек виграв один етап змагання з трьох, але з величезним відривом (дві спроби):

```
Read into vector: 707 ms ellapsed
Read into deque: 167 ms ellapsed
Indexing vector: 182 ms ellapsed
Indexing deque: 183 ms ellapsed
Iterating vector: 27 ms ellapsed
Iterating deque: 28 ms ellapsed
Loading deque by copy: 1725 ms ellapsed

Read into vector: 678 ms ellapsed
Read into deque: 169 ms ellapsed
Indexing vector: 182 ms ellapsed
Indexing deque: 188 ms ellapsed
Iterating vector: 25 ms ellapsed
Iterating deque: 26 ms ellapsed
Loading deque by copy: 2199 ms ellapsed
```

І, як виявилося, зовсім недоцільно передавати конструкторові контейнера пару потокових ітераторів: завантаження триває в рази довше.

Спеціальні операції зі списком

Наступна програма говорить сама за себе. Поекспериментуйте з нею, і ви переконаєтеся, що списки набагато краще обходяться власними засобами, ніж загальними. Спробуйте переробити її для *forward_list*.

файл ListDeeling.cpp

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include "noisy.h"
using namespace std;

// Налаштування лічильників для ведення статистики
// створення/знищення екземплярів Noisy
long Noisy::create = 0, Noisy::assign = 0,
      Noisy::copy = 0, Noisy::destroy = 0;
NoisyReport NoisyReport::nr;

int main()
{
      list<Noisy> L;
      ostream_iterator<Noisy>out(cout," ");
      // Наповнення списку сімома об'єктами
      generate_n(back_inserter(L),7,NoisyGen());
      // Друк початкового списку
      cout<<"\n Printing the list:\n";
      copy(L.begin(),L.end(),out);
      // Обертання списку і друк
      cout<<"\n Reversing the list:\n";
      L.reverse();
      copy(L.begin(),L.end(),out);
      // Впорядкування і друк
      cout<<"\n Sortinging the list:\n";
      L.sort();
      copy(L.begin(),L.end(),out);
      // Обмін двох елементів списку зовнішніми засобами
      cout<<"\n Swaping two elements:\n";
      list<Noisy>::iterator it1, it2;
      it1 = it2 = L.begin();
      ++it2;
      swap(*it1,*it2);
      copy(L.begin(),L.end(),out);
      // Обеотання списку зовнішніми засобами
      cout<<"\n Using genering reverse:\n";
      reverse(L.begin(),L.end());
      copy(L.begin(),L.end(),out);
      cin.get(); cout << endl;

      // спеціальні операції зі списками
      list<Noisy> A;
      generate_n(back_inserter(A), 5, NoisyGen());
      list<Noisy> B;
      generate_n(back_inserter(B), 7, NoisyGen());
      cout << "\nTwo lists were generated:\n";
      cout << " A: "; copy(A.begin(), A.end(), out); cout << endl;
      cout << " B: "; copy(B.begin(), B.end(), out); cout << endl;
```

```
        cin.get();
        // врізка списку
        cout << "\nLet's splice B into A.begin()+3\n";
        list<Noisy>::iterator i = A.begin();
        ++i; ++i; ++i;
        A.splice(i, B);
        cout << "\n   result\n";
        cout << " A: "; copy(A.begin(), A.end(), out); cout << endl;
        cout << " B: "; copy(B.begin(), B.end(), out); cout << endl;
        cin.get();
        // врізка підсписку, вставляння елемента
        cout << "\nLet's splice [A; A+2) into B, then push into front of B:\n";
        B.splice(B.begin(), A, A.begin(), ++(++(A.begin()))); cout << endl;
        B.push_front(Noisy()); cout << endl;
        cout << "\n   result\n";
        cout << " A: "; copy(A.begin(), A.end(), out); cout << endl;
        cout << " B: "; copy(B.begin(), B.end(), out); cout << endl;
        cin.get();
        // впорядкування
        cout << "\n Sort the lists!\n";
        A.sort();
        B.sort();
        cout << "\n   result\n";
        cout << " A: "; copy(A.begin(), A.end(), out); cout << endl;
        cout << " B: "; copy(B.begin(), B.end(), out); cout << endl;
        cin.get();
        // впорядковане злиття
        cout << "\n A.merge(B)\n";
        A.merge(B);
        cout << "\n   result\n";
        cout << " A: "; copy(A.begin(), A.end(), out); cout << endl;
        cout << " B: "; copy(B.begin(), B.end(), out); cout << endl;
        cin.get();
        cout << "\n Cleanup\n";
        return 0;
}
```

Переконайтеся, що методи *sort, splice, merge* не спричиняють фізичного переміщення елементів списку.

Адаптація вектора під поліморфну колекцію

Цей матеріал розглянемо в одній з наступних лекцій.