Лекція 9. Огляд бібліотеки STL

1. *Базові поняття: контейнер, ітератор, алгоритм, функтор.*
2. *Класифікація ітераторів, моделі ітераторів.*
3. *Класифікація контейнерів, спільні можливості.*

У одній з попередніх лекцій ми вже згадували про бібліотеку STL (Standard Template Library), що є частиною стандарту мови C++, і говорили, що ця бібліотека шаблонів оперує поняттями чотирьох категорій.

1. *Контейнери* – шаблони класів, призначених для зберігання колекцій об'єктів.
2. *Ітератори* – сутності, що приховують структуру контейнера та надають уніфікований інтерфейс доступу до його елементів.
3. *Алгоритми* – шаблони функцій, що розв'язують типові задачі опрацювання контейнерів. Для доступу до контейнерів алгоритми використовують ітератори.
4. *Функтори* – сутності, що вміють опрацьовувати оператор круглі дужки, і слугують для налаштування алгоритмів на певні умови виконання.

Бібліотека STL є прикладом узагальненого програмування. Метою узагальненого програмування є створення програмного коду, незалежного від типів даних. З цією метою в мові C++ використовують шаблони. Ми вже маємо початковий досвід написання і використання шаблонів функцій і шаблонів класів. Наприклад, алгоритми роботи стека, не залежать від типу даних, які зберігають у ньому, тому клас, що моделює стек, можна визначити як шаблонний. Крім використання шаблонів STL має ще одну визначальну рису: в її побудові використано *ієрархію концепцій*. Не ієрархію класів, впорядковану за відношенням наслідування, а ієрархію понять, впорядкованих від загальніших до конкретніших. Наприклад, *колекція → контейнер → послідовність → шаблон vector<T>*. Або *ітератор → Input Iterator → Forward Iterator → Bidirectional Iterator → Random Access Iterator*. Говорять також про *уточнення* понять: ітератор переходу (forward iterator) уточнює поняття ітератора введення, а двосторонній ітератор (bidirectional iterator) уточнює поняття ітератора переходу.

Бібліотека STL містить декілька функціональних можливостей, які неможливо виразити мовою C++. Наприклад, неможливо строго задати, що таке ітератор переходу. Можна створити клас, що *має властивості* ітератора переходу, *є втіленням* поняття (або концепції) ітератора переходу, але неможливо заставити компілятор обмежити певний алгоритм STL використанням лише цього класу. Справа в тому, що ітератор – не тип, а перелік вимог. Такі вимоги може задовольняти інший клас-ітератор, структура, звичайний вказівник. І алгоритм STL працюватиме з кожним таким втіленням ітератора. У літературі, присвяченій STL, *концепція* – це опис переліку вимог до деякої сутності програми. Концепція не має синтаксичного оформлення, оскільки існує на рівні понять, домовленостей. Між концепціями може існувати відношення *уточнення*, подібне до відношення наслідування. Але воно, знову ж таки, не має синтаксичного оформлення. Просто уточнена концепція крім вимог своєї попередниці формулює деякі свої додаткові.

Контейнери

Потреба зберігати сукупність значень виникає з перших програм. Це одна з причин, чому кожна мова програмування володіє хоча б одною вбудованою структурою даних: масиви в C, Fortran, Pascal, Basic; списки в Lisp, Prolog. Об'єктно-орієнтовані мови тяжіють до використання чогось «розумнішого», ніж низькорівневий масив з його проблемами контролю індексу та перерозподілу пам'яті. У цьому курсі програмування ми вже стали авторами кількох таких класів. Пригадайте контейнер фігур *ShArray*, «розумний» масив дійсних чисел *ArrayDb*, простий рядок літер *NewStr*, *List<ValueType>*, цілу низку стеків: клас *stack*, шаблони

класів *Stack<T>*, *DStack<Type>*, *AStack<Type, N>*, *Array<Type>*, *HStack<Type>* – вражаючий перелік. Більшість з них було написано з навчальною метою, щоб зрозуміти внутрішнє влаштування контейнера. У повсякденному житті здебільшого використовують готові класи бібліотечних контейнерів, прилаштовуючи їх до власних потреб.

Концептуально колекція – це довільна іменована сукупність значень у програмі. Вбудований масив C++ є найпростішим втіленням колекції. Якщо втіленням колекції є клас (або шаблон класу), що містить сукупність значень (об'єктів), то ми називаємо його контейнером. Усі контейнери бібліотеки STL задовольняють певні спільні вимоги.

Спільні риси всіх контейнерів

Концепція контейнера бібліотеки STL:

- контейнер є однорідним: він може містити виключно однотипні значення;
- контейнер є власником своїх елементів:
  - містить копії доданих об'єктів;
  - об'єкти, призначені до зберігання в контейнері, повинні володіти конструктором копіювання та оператором присвоєння;
  - при знищенні контейнера знищується весь його вміст;
  - при копіювання контейнера копіюється весь його міст;
- контейнер надає тип ітератора, про доступ до елементів контейнера можна говорити в термінах ітераторів;
- контейнер гарантує високу ефективність своїх методів і не надає тих засобів, які неможливо реалізувати ефективно.

Умовно позначимо оголошення типу контейнера та його екземпляри

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

Тоді спільні риси всіх контейнерів можна зобразити таблицею

Позначення	Що повертає	Час виконання	Примітка

`X<T>::iterator`
`X<T>::const_iterator`
`X<T>::reverse_iterator`
	Тип ітератора, що вказує на T	0	1)

`X<T>::reference` `X<T>::const_reference`	T&	0	2)
`X<T>::value_type`	T	0	2)
`X<T>::size_type`	Тип, здатний відобразити кількість елементів контейнера	0	2)

`X<T> u`
`X<T>()`
`X<T> u(a)` або `X<T> u=a`
	конструктори повертають новостворений контейнер	const
		const
		O(n)
			3)
			4)
			5)

`u.begin()` `u.end()`	`iterator`	const	6)
`u.size()`	`size_t n`	const	7)
`u.swap(a)`	`void`	const	8)
`u==a` або `u!=a`	`bool`	O(n)	9)

1) Час виконання 0 означає, що всі дії виконано на етапі трансляції. Типи ітераторів оголошено всередині контейнера, тому використовують кваліфіковані імена. Константний ітератор використовують для читання елементів незмінного контейнера.
2) Визначено за допомогою typedef, використовується в алгоритмах STL.
3) Конструктор за замовчуванням створює порожній контейнер *u*, який далі використовують у програмі, щоб наповнити даними.
4) Конструктор за замовчуванням створює анонімний (безіменний) порожній контейнер, який передають як аргумент у виклику функції.

5) Конструктор копіювання. Час пропорційний кількості елементів у контейнері *a*.
6) Методи повертають екземпляри ітераторів. Нема іншого простого способу створити ітератор.
7) Розмір контейнера – це кількість елементів у ньому.
8) Інтригуюче! Два однотипні контейнери можуть обмінятися вмістом за константний час! Незалежно від свого розміру. Це можливо тому, що фізичного переміщення вмісту контейнерів не відбувається. Контейнер – своєрідний менеджер пам'яті для розташування елементів, він має вказівник на неї. У методі *swap* контейнери *u* і *a* обмінюються вказівниками.
9) Рівність контейнерів означає поелементну рівність їхнього вмісту.

Кожен контейнер має досконалий набір конструкторів: за замовчуванням, копіювання, з параметрами, – і деструктор. Конструктор копіювання виконує глибоке копіювання, деструктор коректно звільняє всю пам'ять контейнера. Та все ж є певні особливості при наповненні контейнера динамічними об'єктами.

Перша вимога до контейнера – однорідність. Як же зберігати в ньому поліморфну колекцію, як ми це робили, наприклад, у класі *ShArray*? Шаблон контейнера потрібно конкретизувати типом вказівника на базовий клас. Далі такий контейнер наповнюють вказівниками на створені за допомогою *new* динамічні об'єкти. Це означає, що стандартні конструктор копіювання і деструктор працюватимуть не так, як потрібно. Скопійований контейнер міститиме новий набір вказівників на ті самі об'єкти, а деструктор видалить вказівники і «забуде» про об'єкти, на які вони вказували. Таким чином, згаданим процесам доводиться надавати особливу увагу.

Поліморфний контейнер містить вказівники на базовий клас. Щоб його наповнити, потрібно виконати *new* для кожного елемента, щоб спорожнити – *delete* кожного.

Перелік контейнерів

Наведемо стислий перелік контейнерів бібліотеки STL, розділений на категорії. На думку автора, окремі з них мають дещо суперечливі назви, які можуть не збігатися з класичною теорією структур даних, а швидше відображають спосіб їхнього створення.

- *Послідовні* містять послідовність значень, тобто, є початок, закінчення, сусідній елемент, порядок елементів фіксовано:
    - *vector* використовує векторну пам'ять, тому наділений швидким індексуванням елементів; дописувати елементи можна в кінець, що може викликати потребу перерозподілу пам'яті, тому воліє знати наперед, скільки елементів у нього помістять;
    - *list* є втіленням двозв'язного списку, дозволяє перебір елементів у обох напрямках, вставка нового елемента (чи вилучення) за константний час у довільному місці контейнера; не надає доступу за індексом, оскільки він має часову складність $O(n)$;
    - *deque* є компромісом між вектором і списку; творці обіцяють швидке індексування і швидку вставку в довільному місці (deque – перший приклад дивної назви, адже в класиці *дек* – це двосторонній стек, або двостороння черга, а не контейнер з довільним доступом до елементів. Не плутайте, STL не має реалізації класичного дека);
    - *forward_list* – однозв'язний список, такий же гнучкий, як list, витрачає удвічі менше додаткової пам'яті для зберігання вказівників, проте позбавлений можливості перебирати елементи в зворотному порядку.
- *Асоціативні контейнери* – назва (ще одна дивна) об'єднує групу контейнерів зі схожою реалізацією: на основі збалансованого дерева пошуку. Асоціацією в програмуванні називають пару (*ключ*; *значення*), а колекцію асоціацій – словником, наприклад, у Smalltalk, чи Python. Словник гарантує, що всі його ключі – різні. Для швидкого пошуку

ключа будують дерево. Такий самий швидкий пошук корисний і для реалізації множини. Очевидно, від таких асоціацій і пішла назва цієї групи контейнерів:

- *set* – впорядкована колекція унікальних значень (відчуваєте різницю: в математиці множина –це невпорядкована сукупність різних елементів; множину STL з математикою поєднує тільки унікальність значень, а впорядкованість контейнерові дісталася завдяки реалізації на основі дерева);
- *map* – відображення, або словник, зберігає колекцію пар (*ключ*; *значення*), впорядковану за ключами (у цьому контейнері визначено оператор [ ], який дає змогу за ключем отримати доступ до значення; через цю особливість map часто називають асоціативним масивом – але це лише зовнішня схожість! ну який масив використовує дерево в якості пам'яті?);
- *multiset* – мультимножина (дуже дивна річ! Чули про морську свинку? Що не так з назвою цієї милої тваринки? Вона не свинка, і не морська! :) В основі multiset дерево пошуку, тому є впорядкованість, а мульти- означає, що дозволено входження однакових елементів – таким чином від множини не залишилося й сліду) дуже корисна річ – це впорядкована послідовність значень (то які тут асоціації!?);
- *multimap* – відображення, якому дозволено містити різні асоціації з однаковими ключами, за ключем можна отримати одразу діапазон значень.

- *Невпорядковані асоціативні контейнери* – аналоги асоціативних контейнерів, але збудовані на основі ефективних хеш-таблиць. Якщо елементів багато, швидкість доступу набагато вища:
  - *unordered_set* – математична множина;
  - *unordered_map* – хеш-таблиця, словник;
  - *unordered_multiset* – «множина» з повтореннями, зберігає значення і кількість входжень кожного з них;
  - *unordered_multimap* – словник з повтореннями ключів, групи асоціацій з однаковими ключами зберігаються поруч.
- *Контейнерні адаптери* – назва відображає спосіб виконання, а не функціонування. Стек, чергу називають ще динамічними контейнерами (в класичній теорії), оскільки кожне звертання до них змінює їхній стан. Стек можна реалізувати, наприклад, на основі однозв'язного списку, якщо дозволити вставляти і вилучати тільки першу ланку. Творці STL майже так і зробили. Контейнери цієї групи реалізовано на базі послідовних контейнерів. Насправді вони є тонкими обгортками, що містять послідовний контейнер у захищеній частині, і маскують більшість його можливостей, залишаючи доступними тільки ті, що потрібні стекові чи черзі. Звідси і назва адаптер:
  - *stack* – реалізація стека з некласичними назвами методів;
  - *queue* – реалізація черги з некласичними назвами методів;
  - *priority_queue* – пріоритетна черга.
- *«Майже контейнери»* – власна вигадана назва для групи класів, що відрізняються від усіх перерахованих вище своїм спеціальним призначенням (це контейнери, але не на всі випадки життя):
  - *pair* – структура з двох значень, активно використовується, наприклад, асоціативними контейнерами;
  - *complex* – шаблон типу, що реалізує комплексні числа та дії з ними, можна конкретизувати різними дійсними типами залежно від того, яка точність дійсної та уявної частини комплексного числа потрібна;
  - *array* – обгортка навколо вбудованого масиву, що наділяє його рисами справжнього контейнера (ітератори, перевірка індекса тощо);
  - *valarray* – числовий масив та потужна бібліотека ефективних методів для виконання обчислень;
  - *basic_string* – основа всіх рядкових типів;
  - *unique_ptr, shared_ptr, weak_ptr* – «розумні» вказівники;
  - *та ще дещо*.

Різновиди ітераторів

Ітератор – деяка сутність (щось) для навігації по контейнеру, для доступу до його елементів:

- екземпляр допоміжного класу, оголошеного в класі контейнера;
- знає структуру контейнера, приховує від інших;
- operator*() для доступу до елемента;
- operator++() для переміщення контейнером;
- bool operator==(iterator) для перевірки того, чи завершили перебір контейнера.

Ітератор – концепція, використана в STL для маскування конкретної структури контейнера. Користувачеві невідомо, як працює ітератор, але відомо, що він може робити: перебирати елементи контейнера (за допомогою операторів інкременту-декременту), надавати доступ до елементів контейнера (за допомогою оператора розіменування); ітератори можна порівнювати звичайними операторами «==», «!=». Ітератор відокремлює алгоритм опрацювання від структури контейнера. Це дає змогу використовувати декілька ітераторів на один контейнер. Концепція – логічне поняття, її втілення в програмі – модель.

Моделі ітераторів

Якщо трактувати вбудований масив як модель контейнера, то вказівник на елементи масиву – його ітератор (модель концепції ітератор):

```
double a[]={2,9,5,7};     // контейнер
double * arrIter = a;     // ітератор
```

Кожну модель ітератора створено для взаємодії з певним контейнером (структурою певного типу). Наприклад, всередині кожного класу-контейнера бібліотеки STL оголошено його клас ітератор:

```
vector<double> v[5];                        // контейнер
vector<double>::iterator it = v.begin();    // ітератор
```

Завжди можна оголосити і власну модель ітератора. Наприклад, для роботи з лінійним однозв'язним списком:

```
struct Node {                                   // вузол, що є частиною списку
      double item; Node* next;
      Node(double x, Node* link=0):item(x),next(link){}
};

class listIter                                  // «примітивний» ітератор списку
{
      Node* p;
public:
      listIter():p(0){}
      listIter(Node* ptr):p(ptr){}
      double& operator*() { return p->item; }
      listIter& operator++() { p = p->next; return *this; }
      listIter operator++(int) { listIter t=*this; p = p->next; return t; }
      bool operator==(const listIter& a) { return p == a.p; }
      bool operator!=(const listIter& a) { return p != a.p; }
};
```

Роль ітератора в одній з попередніх лекцій зіграв *List<ValueType>::Iter*.

Класифікація ітераторів

У бібліотеці STL використано декілька концепцій ітераторів. Вони утворюють логічну ієрархію від найбільш загальних, до конкретніших, уточнених, з ширшим переліком можливостей.

- Input Iterator (II) – *ітератор введення*: надає доступ для читання елементів контейнера (оператор *); перебирає елементи контейнера: кожен елемент – один раз, але порядок не гарантовано (обидва оператори ++); ітератори можна порівнювати (==, !=). Ітератор для однопрохідних алгоритмів.
- Output Iterator (OI) – *ітератор виведення*: доступ для запису значень в елементи контейнера (*); перебір без гарантування порядку перебору (обидва ++). Ітератор для однопрохідних алгоритмів. Різновидами OI є *ітератор вставляння*, *ітератор дописування*.
- Forward Iterator (FI) – *ітератор переходу*: доступ для читання та запису (*); перебір завжди в одному фіксованому порядку (обидва ++); порівняння ітераторів (= =, !=). Можна використовувати попередні значення ітератора. Ітератор для багатопрохідних алгоритмів.
- Bidirectional Iterator (BI) – *двосторонній ітератор*: доповнює попередню концепцію операторами декременту для пересування контейнером (обидва оператори ––).
- Random Access Iterator (RI) – *ітератор довільного доступу*: дає змогу локалізувати елемент контейнера відносно інших елементів, підтримує операції зменшення-збільшення ітератора на число (на певну кількість елементів контейнера, оператори + та –), обчислення кількості елементів між вказівниками (оператор –), дозволяє порівнювати ітератори за допомогою < та >, визначає оператор індексування []. Використовують в складних багатопрохідних алгоритмах, наприклад, впорядкування.

Ітератори вставляння

Синтаксис

```
// ітератор дописування в кінець контейнера
template<class Container>
back_insert_iterator<Container> back_inserter(
	Container& _Cont
	);
// ітератор вставляння на початок контейнера
template<class Container>
front_insert_iterator<Container> front_inserter(
	Container& _Cont
	);
// ітератор вставляння на вказане місце контейнера
template <class Container>
insert_iterator<Container>
	inserter(
	Container& _Cont,
	typename Container::iterator _Where);
```

Потокові ітератори

Синтаксис

```
// ітератор виведення в потік
template <class Type,
	class CharType = char, class Traits = char_traits <CharType>>
class ostream_iterator
{
	ostream_iterator(
		ostream_type& _Ostr);

	ostream_iterator(
		ostream_type& _Ostr,
		const CharType* _Delimiter);
};
```

Приклади оголошення

```
ostream_iterator<MyClass>(cout," ");
ostream_iterator<int> OsIt(cout,"\n");
```

```
ofstream fout("MyFile.txt");
ostream_iterator<double> FsIt(fout,"\t");
```

Синтаксис

```
// ітератор введення з потоку
template <class Type,
	class CharType = char, class Traits = char_traits<CharType>,
	class Distance = ptrdiff_t >
class istream_iterator
	: public iterator<input_iterator_tag,
	Type, Distance,    const Type *, const Type&>
{
	istream_iterator();

	istream_iterator(
		istream_type& _Istr);
};
```

Приклади оголошення

```
// оголошення початкового ітератора
ifstream fin("MyFile.txt");
istream_iterator<double> FsIt(fin);

// оголошення кінцевого ітератора
istream_iterator<double>();
istream_iterator<double> endFsIt;
```

Приклади використання ітераторів уже наприкінці лекції.

Алгоритми

«Щось», що вміє перетворювати контейнери: копіювати, знаходити, впорядковувати, об'єднувати тощо:

- шаблони функцій;
- забезпечують найвищу обчислювальну ефективність;
- використовують ітератори та функтори.

Написані нами шаблони *FindValInSuccession* та *FindAnyInSuccession* дуже на них схожі.

Є велика кількість стандартних алгоритмів. Ми присвятимо їм окрему лекцію. У цій наведемо приклад одного з них. Він допоможе нам побудувати приклади використання ітераторів.

Алгоритм копіювання

Синтаксис

```
// копіювання інтервалів
template<class _InIt, class _OutIt> inline
_OutIt copy(_InIt _First, _InIt _Last, _OutIt _Dest);
```

_InIt – тип ітератора введення
_First – початок інтервалу копіювання
_Last – закінчення інтервалу
_OutIt – тип ітератора виведення
_Dest – початок місця призначення (місця має вистачити для всього інтервалу, якщо це не так, то можна використовувати ітератори вставляння)

Приклади

```
	// правильно
	int a[] = { 1, 2, 3, 4, 5 };
	const int n = sizeof a / sizeof a[0];
```

```
int b[n];
copy(a, a + n, b);

vector<int> c(n);
copy(a, a + n, c.begin());
cout << c.size() << endl;

// неправильно
vector<int> d;
d.reserve(n);
copy(c.begin(), c.end(), d.begin());
```

Приклади використання ітераторів

Ми могли б використати різноманітні контейнери бібліотеки STL та різні типи даних для їхніх елементів, але зараз важливо зрозуміти дію ітераторів, тому використаємо найпростішу комбінацію: масив, вектор, цілі числа.

```
// Використаємо вбудований масив і послідовний контейнер
// Заради простоти наповнимо їх цілими числами
int a[] = { 1,2,3,4,5,6,7,8 };
const int n = sizeof a / sizeof *a;
int b[n] = { 0 };

// очевидний приклад використання алгоритму копіювання
copy(a, a + n, b);
cout << "\n * Масив b - значення після копіювання *\n";
for (int i = 0; i < n; ++i) cout << '\t' << b[i];
cout << '\n';
```

На екрані отримаємо цілком очікуваний результат – послідовність цілих від 1 до 8. Але виведення масиву – це теж свого роду копіювання. Джерелом є масив, одне з найпростіших втілень послідовності. Приймачем також має бути послідовність, принаймні в термінах ітераторів. Трактувати консоль як послідовність нам допоможе екземпляр класу *ostream_iterator* – ітератор потоку виведення (класичний приклад адаптера). Це шаблон, параметризований типом даних, які потрібно виводити, в конструкторі він приймає екземпляр потоку виведення (на консоль чи до файлу) і розділювач, який буде виведено після кожного значення. Розділювачем має бути рядок, навіть якщо він містить тільки одну літеру.

```
// потоковий ітератор для виведення на консоль
// потрібно задати три параметри:
//  - шаблона - тип даних, які треба виводити
//  - конструктора - потік і рядок-розділювач
ostream_iterator<int> out(cout, "\t");
// виведення на консоль за допомогою копіювання
cout << "\n * Масив a - виведення копіюванням *\n";
copy(a, a + n, out);
cout << "\n * Масив b - виведення копіюванням *\n";
copy(b, b + n, out);
cout << '\n';
```

Отримаємо ту ж послідовність значень, що й попереднього разу, тільки цикл писати не довелося. Створений об'єкт *out* можна використовувати не тільки з алгоритмами. Це ітератор, який «розуміє» оператори розіменування та інкременту, а присвоєння розіменованому ітераторові потоку виведення цілого значення насправді означає виведення цього значення в потік. На перший погляд, несподівано, але цілком логічно в термінах ітераторів.

```
// неочевидне виведення на консоль
cout << "\n --- Демонстрація можливостей потокового ітератора\n";
for (int i = 7; i < 99; i += 10) *out++ = i;
```

Такий самий підхід можна застосувати і до контейнерів. Пригадаємо, що кожен з них має спеціальні методи *begin* та *end*, які повертають ітератори відповідно на початок і кінець контейнера.

```
// виведення контейнера на консоль за допомогою копіювання
// ітератори надасть сам контейнер
vector<int> A = { 10,20,30,40,50,60,70,80 };
cout << "\n * Вектор A - виведення копіюванням *\n";
copy(A.begin(), A.end(), out);
cout << '\n';

// копіювання контейнерів, розмір відомий
vector<int> B(A.size()); // B заповниться нулями
copy(A.begin(), A.end(), B.begin());
cout << "\n * Вектор B - після копіювання *\n";
copy(B.begin(), B.end(), out);
cout << '\n';

// Джерело і приймач можуть бути різних типів
copy(a, a + n, B.begin());
cout << "\n * Вектор B - після копіювання з масиву *\n";
copy(B.begin(), B.end(), out);
cout << '\n';
```

Алгоритм *copy* не переймається, чи вистачить місця у приймача для всіх копійованих даних. Це відповідальність користувача алгоритму. Ми спеціально турбувалися про правильні розміри масиву *b* та вектора *B*. Але на практиці треба бути готовим до того, що задати потрібний розмір з різних причин не вдасться. Чи може *copy* змінити розмір приймача? Ні, але *це може зробити відповідний ітератор*! Використаємо шаблонну функцію *back_inserter*, яка повертає спеціальний об'єкт-адаптер *ітератор дописування в кінець* для послідовного контейнера. Він поводить себе як ітератор, а насправді викликає метод *push_back* контейнера.

```
// Копіювання може змінити розмір приймача, якщо використати спеціальний ітератор
vector<int> C = { 55 };
cout << "\n * Вектор C - початкове значення *\n";
copy(C.begin(), C.end(), out);
copy(A.begin(), A.end(), back_inserter(C));
cout << "\n * Вектор C - після копіювання *\n";
copy(C.begin(), C.end(), out);
cout << '\n';
```

Дописування в кінець зазвичай викликає перебудову пам'яті контейнера: векторові доводиться виділяти нову більшу ділянку і переміщати туди свої елементи. Якщо потрібна копія контейнера, то набагато ефективніше сконструювати його за допомогою ітераторів, що задають копійований інтервал.

```
// ітератори можна використовувати в конструкторах
vector<int> D(C.begin(), C.end());
cout << "\n * Вектор D - після конструювання *\n";
copy(D.begin(), D.end(), out);
cout << '\n';
```

Повернімося до потокових ітераторів. Вони стануть у пригоді для налагодження обміну даними з файловими потоками. Виведення до файла виглядає так само, як на консоль, тільки попередньо створили файловий потік виведення.

```
// файлові ітератори можна використовувати для виведення вмісту контейнера
ofstream fout("Integers.txt");
copy(D.begin(), D.end(), ostream_iterator<int>(fout, "\t"));
fout.close();
system("copy Integers.txt con");
cout << "\n * Вектор D записано до файла *\n";
```

Для виведення потрібен тільки один потоковий ітератор, що задає початок приймача, а для введення – два: початок і закінчення інтервалу, який треба прочитати. Трактувати файл як послідовність допоможуть екземпляри класу *istream_iterator*, що параметризований типом даних, які потрібно читати. В конструкторі ітератор приймає файловий потік введення. За домовленістю ітератор на кінець потоку введення повертає конструктор за замовчуванням. Наперед не відомо, скільки даних записано у файлі, тому в якості приймача використано ітератор дописування.

```
// файлові ітератори можна використовувати для завантаження контейнера з файла
ifstream fin("Integers.txt");
A.clear(); // оголошений раніше контейнер очистимо від старого вмісту
// Для початкового ітератора потрібно задати файловий потік
// Кінцевий ітератор повертає конструктор за замовчуванням
copy(istream_iterator<int>(fin), istream_iterator<int>(), back_inserter(A));
cout << "\n * Вектор A - після завантаження з файла *\n";
copy(A.begin(), A.end(), out);
cout << '\n';
fin.close();
```

Ми вже зазначали, що використання конструктора ефективніше за дописування в контейнер. Виявляється, такий підхід можна використати і для завантаження контейнера з файла. Тільки з одним уточненням. У алгоритмі *copy* ми використали анонімні (безіменні) ітератори: результат виклику конструкторів одразу став значенням формальних параметрів алгоритму. З конструктором контейнера треба вчинити трохи інакше: ітератор на початок потоку введення треба помістити в окрему змінну, а вже її використати як аргумент конструктора.

```
// файлові ітератори можна використовувати для створення контейнера з файла
// Для коректної роботи конструктора обов'язково
// оголосити стартовий ітератор як іменовану змінну
fin.open("Integers.txt");
istream_iterator<int> start_iter(fin), end_iter;
vector<int> E(start_iter, end_iter);
//vector<int> E(start_iter, istream_iterator<int>()); // можна і так
cout << "\n * Вектор E - після створення з файла *\n";
copy(E.begin(), E.end(), out);
cout << '\n';
fin.close();
```

Фантастично! Складний колись процес читання з файла послідовності значень можна зобразити одним рядком коду. До того ж, такий підхід працюватиме не тільки для значень стандартних типів, а й для довільних типів, визначних програмістом, для яких оголошено оператор введення з потоку. Ось, що значить програмування на високому рівні абстракції.

Читачам було б корисно поекспериментувати з наведеними тут фрагментами коду: випробувати різні комбінації контейнерів для копіювання, спробувати копіювати не цілий контейнер, а лише його частину, випробувати інші ітератори вставляння.

У наступній лекції:
Послідовні контейнери: концепція, влаштування, базові операції. Приклади використання контейнерів vector, list, deque. Порівняння ефективності послідовних контейнерів. Адаптація вектора під поліморфну колекцію. Спеціальні операції зі списком.