Введення в стандартну бібліотеку шаблонів

Цей розділ досліджує один з найбільш важливих складових C++ , яким є ***Стандартна бібліотека шаблонів*** STL. Її включення на мові C++ була одним з основних досягнень стандартизації бібліотеки містить загальний шаблон класів функцій, що реалізують багато широко використані алгоритми та структури даних, наприклад, вона визначає майстер-класи вектори, списки, списки очікування та стеки, а також численні процедури, щоб працювати з ними, бо STL бібліотеки складається з шаблонних класів, своїх алгоритмів та структур даних, які можуть бути застосовані до майже будь-яких типів даних.

З точки зору розробки ПЗ стандартна бібліотека шаблонів є дуже складний проект, в якому використовуються найскладніші властивості C++. Для того, щоб зрозуміти і правильно використовувати цю бібліотеку, потрібно вивчити повністю мову C+ + і прекрасно розуміти вказівники, посилання та шаблони. Чесно кажучи, синтаксис шаблонних класів, які описані в бібліотеці STL, виглядає дуже страшним, хоча це дійсно не так складно, як здається на перший погляд.

Перегляд бібліотеки STL.

Незважаючи на те, що стандартна бібліотека шаблонів велика, а її синтаксис — це складно, це досить легко працює, якщо правильно розуміти, з чого вона складається і яким чином працює. Відповідно, перш ніж заглибитись в прикладах, ви повинні зробити короткий огляд всієї бібліотеки.

Бібліотеки STL лежить на трьох основних компоненти: ***контейнери, алгоритми*** і ***ітератори.*** Взаємодія цих елементів забезпечує стандартні зв'язки дуже широкого переліку задач програмування.

Контейнери

Контейнери- це об'єкти, які містять інші об'єкти всередині. Є кілька типів контейнерів, наприклад, клас **vector** визначає динамічний масив, клас **deque** створює двосторонню чергу, а клас **list** дає змогу працювати з лінійним списком. Ці контейнери називаються ***послідовними*** (sequence container), оскільки за термінологією бібліотеки STL послідовність – це лінійний список. Крім основних контейнерів в стандартній бібліотеці шаблонів є ***асоціативні контейнери*** (associative container), що дають змогу ефективно отримувати значення з ключа. Наприклад, клас **map** забезпечує доступ до значень, які мають унікальні ключі. Таким чином, **map** зберігає пари ключ-значення і дає змогу отримати значення за указаним ключем.

Кожен контейнерний клас визначає набір функцій, які можуть бути застосовані до контейнера. Наприклад, клас **list** містить функції вставити, видалити, і об'єднати елементи, а клас **stack** надає функції для виштовхування і заштовхування елементів.

Алгоритми

Алгоритми застосовуються до контейнерів. Вони дають змогу керувати вмістом контейнера: ініціалізувати, сортування, пошуку та перетворення вмісту з контейнера. багато алгоритмів застосовуються до всього ***діапазон*** (gagŝe) елементи, які є у контейнері. ***(Діапазон*** називають послідовності, що дозволяє випадковий доступ.- ***ред.)***

Ітератори

Ітератори — це об'єкти, які нагадують вказівників. Вони дозволяють переміщатися по вмісту контейнера, як переміщається по елементах масиву. Є п'ять типів ітераторів.

Ітератор	***Вид доступу***
Вибірковий ітератор	Зберігає і отримує значення. Дозволяє довільного доступу до елементів.
Двонаправлений ітератор	Зберігає і отримує значення. Рухається вперед і назад.
Прямий літератор	Зберігає і отримує значення. Рухається тільки вперед.
Ітератор вводу	Отримує, але не зберігає елементи. Рухається тільки вперед.

Ітератор виводу	Зберігає, але не отримання значень. Рухається тільки вперед.

Як правило, ітератори, що володіють широкими можливостями, можуть бути використані замість слабших ітераторів. Наприклад, замість ітератора вводу, можна застосувати прямий ітератор.

Ітератори діють як вказівники. Їх можна збільшити, зменшити і розіменувати за допомогою оператора "*". Ітератори, тобто об'єкти, які мають тип **iterator,** оголошуються в різних контейнерах.

В бібліотеці STL є також ***зворотні ітератори*** (reverse iterator). Вони можуть бути двонаправленими, або вибірковими ітераторами. В обох випадках, вони повинні бути в змозі пересуватися в контейнері в протилежному напрямку. Таким чином, якщо зворотний ітератор посилається на кінець послідовності, його збільшення призведе до переміщено до попереднього елемента.

Згадки про типи ітератори використовується в описах шаблони, ми будемо використовувати таку термінологію.

Значення терміну	***Значення***
BiIter	Двонаправлений ітератор
ForIter	Прямий ітератор
InIter	Ітератор вводу
OutIter	Ітератор виводу
RandIter	Вибірковий ітератор

Інші елементи бібліотеки STL

Крім контейнерів, ітераторів та алгоритмів в бібліотеці STL присутні інші стандартні компоненти. Найбільш важливим серед них є розділювачі (allocators), предикати (predicates), функції порівняння (comparison function) і функтори (function objects). Іноді функтори називають ***об'єктами функцій.***

Кожен ітератор має ***розділювач,*** підібраний спеціально для нього. Розділювачі управляють виділенням пам'яті для контейнера. Розділювач, за замовчуванням, є об'єкт класу **allocator.** Ви можете знайти свій власний розподільника, зосереджений на конкретній програмі, але в більшості випадків це можна керувати розділювачем за замовчуванням.

Деякі алгоритми і контейнери використовують особливий тип функції, яка називається ***предикат.*** Є два типи предикатів: унарних та бінарних. ***Унарний предикат*** має один з аргументів, та ***бінарний –*** два. Ці функції повертають булеві значення **true** або **false. У**мови, що впливають на ці значення, програміст може сформулювати для себе самостійно. Надалі унарні предикати, ми будемо відносити до типу **UnPred** і бінарні є типу **Bin-Pred.** Аргументів для двійкових предикатів завжди відображаються по порядку: *first, secon.* Аргументи як бінарних, так і унарних предикатами є об'єкти, збережені у контейнері.

Деякі алгоритмі і класи використовувати спеціальні види бінарних предикатів, призначених для порівняння двох елементів — функції, які повертають значення порівняння. **true,** якщо перший аргумент менше, ніж другий. Функції порівняння мають тип **Comp.**

На додаток до заголовків специфічних для різних шаблонних класів бібліотеки STL, в ній використовуються заголовки **<utility>** і **< functional >.** Наприклад, шаблонний клас **pair,** що містить пари значень, визначається в заголовку **<utility>.**

Шаблони, визначені в заголовку **< functional >,** дають змогу створювати об'єкти, які визначають оператора функції **operator().** Такі об'єкти називають *функторами* (*function objects*). Їх часто використовується замість вказівників на функції. В заголовку **< functional>,** є кілька вбудованих функторів.

plus	**minus**	**multiplies**
negate	**equal_to**	**not_equal_to**
less	**less_equal**	**logical_and**

divides	**logical_or**	**greater_equal**
greater	**modulus**	**logical_not**

Найбільш широко використовується функтор **less,** що визначає відносини між об'єктами. Ви можете використовувати функтори. замість вказівників функцій в шаблонних алгоритмах, що використовують функтори замість вказівників на функції збільшує ефективність коду.

В стандартній бібліотеці є два різновиди функторів: ***редактори посилання*** (biders) та ***інвертори*** (negators). Редактори посилання зв`язують аргумент з функтором. Інвертор повертає заперечення предиката.

Залишилося дати визначення *адаптера*. Простіше кажучи, адаптер перетворює одну сутність в іншу. Наприклад, контейнер **queue**, що створює стандартну чергу, є адаптером по відношенню до контейнера **deque.**

Контейнерні класи

Як ми вже казали, контейнери представляють об'єкти, які зберігають дані. Контейнери, визначені стандартною бібліотекою, перераховані в таблиці.

Контейнер	Опис	Заголовок
bitset	Набір бітів	**<bitset>**
deque	Двостороння черга	**<deque>**
list	Лінійний список	**<list>**
map	Зберігає пари ключ-значення, в котрих кожному ключу відповідає лише одне значення	**<map>**
multimap	Зберігає пари ключ-значення, в котрих кожному ключу можу відповідати декілька значень	**<map>**
multiset	Множина, в яку кожен елемент може входити декілька раз	**<set>**
priority_queue	Черга з пріоритетами	< **queue**>
queue	Черга	< **queue**>
set	Множина, в яку кожен елемент може входити тільки раз	**<set>**
stack	Стек	<**stack**>
vector	Динамічний масив	**<vector>**

Оскільки імена узагальнених типів, що використовуються в якості параметрів шаблонних класів, вибираються довільно, в контейнерних класах вони перевизначаються за допомогою оператора **typedef**. Це дозволяє конкретизувати імена цих типів. Перечислимо імена, які найчастіше перевизначаються в допомогою оператора **typedef**

size_type	Деякий різновид целочисленного типу
reference	Посилання на елемент
const_reference	Константне посилання на елемент
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 ключа
value_compare	Тип функції, що порівнює два значення

Загальні принципи роботи

Незважаючи на складний внутрішній пристрій бібліотеки STL, використовувати її досить легко. По-перше, слід вибрати тип з контейнера, кожен з них має переваги і недоліки. Наприклад, клас **vector** хороший при роботі з об'єктами, що забезпечують прямий доступ, масивами. Клас **list** дозволяє вставляти та видаляти елементи, але знижує продуктивність програми. Клас **map** створює асоціативний контейнер, але за додаткову плату пам'яті.

Вибір контейнерного типу визначає функції-члени, які будуть використовуватись для вставлення, модифікації та видалення елементів, які зберігаються в ньому. За винятком класу **bitset,** всі контейнери автоматично збільшується, коли елементи додаються та зменшуються при видаленні.

Є багато способів для вставки і видалення елементи-контейнера. Наприклад, послідовні **(vextor, list і deque)** та асоціативні контейнери **(map, multimap, set і multiset)** містять функцію член **insert(),** яка вставляє елементи в контейнер і функція-член **erase(),** вилучає їх з контейнера. Крім того, послідовні контейнери містять функцію-член **push_back()**, що додає новий елемент в кінец послідовності, і функцію-член **pop_back()**, що видаляє його звідти. Ці функції використовуються для вставки і видалення елементів частіше інших. Контейнери **list** І **degue** також містять функцію-член **push_front()**, що додає новий елемент в початок послідовності, і функцію-член **pop_front**, що видаляє його звідти.

Як правило, доступ до елементів контейнера забезпечується за допомогою ітераторів. Послідовності і асоціативні контейнери містять функції-члени **begin()** і **end()**, які повертають ітератори, що посилаються відповідно на початок і кінець контейнера. Ці ітератори дуже зручні для доступу до елементів масиву. Наприклад, викликавши функцію-член **begin()**, можна отримати итератор, встановлений на початок масиву, а потім збільшувати його, поки його значення не співпаде зі значенням функції **end()**.

Асоціативні контейнери містять функцію-член **find()**, яка виявляє елемент контейнера по заданому ключу. Оскільки асоціативні контейнери пов'язують ключ з його значенням, функція **find()** дозволяє визначити, як розміщені елементи контейнера.

Клас **vector** є динамічним масивом, тому він підтримує звичайні синтаксичні конструкції, що застосовуються для індексації його елементів.

Інформація, що зберігається в контейнері, може оброблятися декількома алгоритмами. Ці алгоритми дозволяють не тільки змінювати вміст масиву, а й перетворювати один тип послідовності в інший.

Вектори

Найбільш універсальним контейнерним класом є клас **vector**, що представляє собою динамічний масив, розміри якого можуть змінюватися в міру необхідності. Як відомо, у мові С ++ розмір масиву фіксується на етапі компіляції. Цей спосіб визначення масивів найбільш ефективний. Однак він пов'язаний з серйозними обмеженнями, оскільки розмір масиву в ході програми змінюватися не може. Клас **vector** вирішує цю проблему, виділяючи стільки динамічної пам`яті, скільки буде потрібно. Незважаючи на те, що вектор є динамічним масивом, для доступу до його елементів можна використовувати звичайний спосіб індексації.

Шаблонна специфікація класу **vector** виглядає наступним чином.

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

Тут символ **T** вказує тип даних, які зберігаються в контейнері і клас **Allocator** визначає розподільника пам'яті. За замовчуванням використовується стандартний розподільник пам'яті. Клас **vector** містить наступні конструктори.

```
explicit vector (const Allocator &a = Allocator());
explicit vector(size_type num, const T &val=T(),
                const Allocator &a = Allocator());
vector(const vector<T, Allocator &ob);
template <class InIter> vector(InIter start, InIter end,
                const Allocator &a = Allocator());
```

Перший конструктор створює порожній вектор. Другий конструктор створює вектор, що складається з *num* елементів, що мають значення *val*, яке можна задавати за замовчуванням.

Третій конструктор створює вектор, що містить елементи вектора *ob*. Четвертий конструктор створює вектор, що складається з елементів, що лежать в діапазоні, визначеному ітераторами *start* і *end*.

Щоб забезпечити максимальну гнучкість і машинонезалежнсть, будь-який об'єкт, що зберігається у векторі, повинен визначати конструктор за замовчуванням, а також оператори "<" і "==". Деякі компілятори накладають на такі об'єкти додаткові обмеження. (Точна інформація про ці обмеження міститься в технічній документації компілятора.) Всі вбудовані типи за замовчуванням задовольняють ці умови.

Хоча шаблонні синтаксичні конструкції виглядають досить складними, вектор оголошується просто. Розглянемо кілька прикладів.

```
vector <int> iv;              //Створюється порожній вектор типу int.
vector <char> cv(5);          //Створюється вектор типу char;
                              //з п'ять елементів.
vector <char> cv(5, 'x');     //Ініціалізується вектор типу char;
                              //з п'ять елементів.
vector <int> iv2(iv);         //Створюється вектор типу int, який
                              //ініціалізується іншим
                              //цілочисельним вектором..
```

В класі **vector** визначенітакі оператори порівняння:

==, <, <=, !=, >, >=

Крім того, в класі **vector** визначений оператор "[ ]". Це дозволяє звертатися до елементів вектора за допомогою стандартної індексації масиву.

Функції-члени, визначені в класі **vector**, перераховані в табл. 24.2.. До найбільш поширених функцій - членам класу **vector** належать функції-члени **size(), begin(), end(), push_back(), insert()** і **erase()**. Функція **size()** повертає поточний розмір вектора. Ця функція досить корисна, оскільки з її помо¬щью можна визначити розмір вектора в ході виконання програми. Нагадаємо, що вектори можуть змінювати свої розміри в міру необхідності, тому їх розмір визначається під час виконання програми, а не на етапі компіляції.

Функція **begin()** повертає ітератор, установлений на початок вектора. Функція **end()** повертає ітератор, установлений на кінець вектора.

Функція **push_back()** записує значення в кінець вектора. Якщо вектор заповнений, при записі його розмір збільшується. За допомогою функції **insert()** значення можна записати в середину вектора. Крім того, вектор можна ініціалізувати. У будь-якому випадку для доступу до елементів вектора можна застосовувати звичайні обозначе¬нія індексації. Для видалення елемента з вектора призначена функція **erase()**.

Таблиця 24.2. Функції, визначені в в класі vector

Функція-член	***Опис***
reference_back(); const_ reference_back() const;	повертає посилання на останній елемент вектора
iterator begin(); const_iterator begin() const;	повертає ітератор до першого елемента
void clear();	видаляє всі елементи з вектор
bool empty const;	повертає значення true**,** якщо вектор пустий, або повертає значенням false
iterator end(); const_iterator end() const;	повертає ітератор на перший елемент набору
iterator erase(iterator *i*);	Видаляє елемент, який посилається ітератор *i*.
iterator erase	

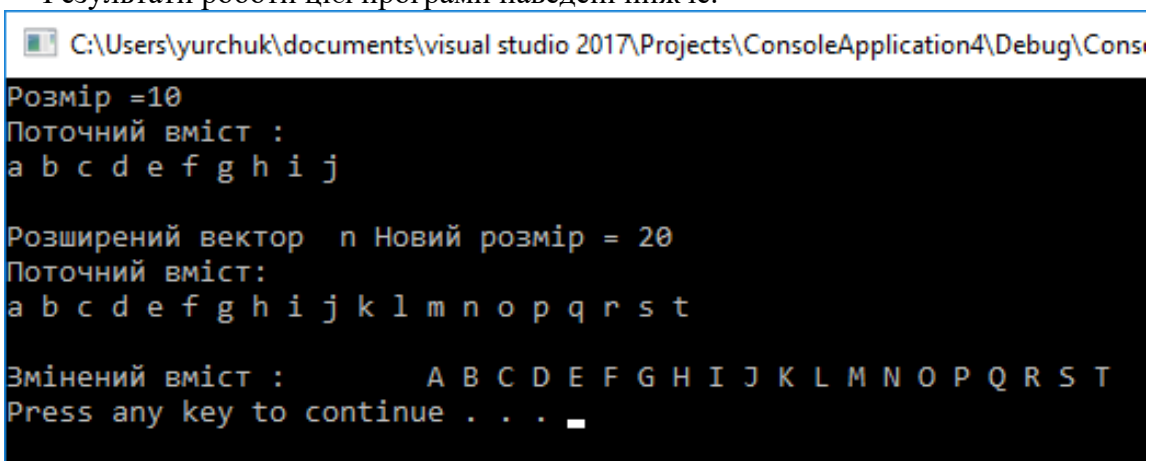
(iterator *start*, iterator *end*);	Видаляє елементи з діапазону вказано ітератори *start* і *end* . Повертає ітератор до елементу наступного за останнім видаленим
reference front(); const_reference front() const;	Повертає посилання на перший елемент вектор
iterator insert (iterator *I*, const T &*val*);	Вставляє елемент *val* перед елемент, на який посилається ітератор *i* Повертає ітератор, встановлений на елемент *val*.
void insert(iterator *i*, size_type *num*, const T &*val*);	вставляє *num* копій елемента *val* перед елементом, на який посилається ітератор
template<class inIter> void insert(iterator *i*, InIter *start*, InIter *end*);	вставляє перед елементом, на який посилається ітератор *i*, послідовність елементів, вказану ітераторами *start* і *end*
reference operator[] (size_type *i*) const;	Повертає посилання на вказані ітератор
const_refererance operator[] (size_type *i*);	
void pop_back(); void push_back(const T &*val*);	Видаляє останній елемент вектора додає елемент *val* в кінці вектора
size_type size() const;	Повертає поточне число елементів вектора

Давайте розглянемо приклад, що ілюструє основні операцій над векторами.

```
#include "windows.h"

#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	vector<char> v(10); // Створюємо вектор з 10 елементів.
	int i;

	//виводимо на екран вихідний розмір вектора v
	cout << "Розмір =" << v.size() << endl;

	//Присвоюємо елементам вектора певні значення.
	for (i = 0; i< 10; i++) v[i] = i + 'a';

	//Виводимо на екран вміст вектора.
	cout << "Поточний вміст : \n";
		for (i = 0; i<v.size(); i++) cout << v[i] << " ";
	cout <<"\n\n";

	cout << "Розширений вектор \ n ";
	/*Записуємо нові елементи наприкінці вектора, при цьому його розмір збільшується.*/
		for (i = 0; i< 10; i++)  v.push_back(i + 10 + 'a');
	//Виводимо поточний розмір вектора V.
	cout << "Новий розмір = " << v.size() << endl;
	//Відображення на екрані вмісту вектор.
	cout << "Поточний вміст:\n";
	for (i = 0; i< v.size(); i++) cout << v[i] << " ";
	cout << "\n\n";
	//Змінимо вміст вектора.
	for (i = 0; i< v.size(); i++)  v[i] = toupper(v[i]);
	cout << "Змінений вміст : \t";
		for (i = 0; i< v.size(); i++) cout << v[i] << " ";
```

```
        cout << endl;

        system("pause");
        return 0;
}
```

Результати роботи цієї програми наведені нижче.



Розглянемо цю програму уважніше. У функції **main()** створюється вектор символів **v**, довжина якого дорівнює 10. Цей факт підтверджується викликом функції-члена **size()**. Потім 10 елементів вектора **v** ініціалізуються символами, починаючи з букви **a** і завершуючи буквою **j**. Зверніть увагу на те, що при цьому застосовується стандартна індексація. Потім за допомогою функції **push_back()** в кінець вектора **v** додається ще 10 елементів. Для того щоб записати ці 10 елементів, розмір вектора **v** збільшується. Як свідчать результати роботи програми, після цієї операції розмір масиву стає рівним 20. У підсумку значення елементів вектора **v** змінюються, причому для доступу до них використовується стандартна індексація.

У цій програмі є ще один цікавий момент. Зверніть увагу на те, що цикли, які відображатимуть на екрані вміст масиву V, використовують як верхньої межі значення функції **v.size().** Одна з переваг векторів над звичайними масивами полягає в тому, що поточний розмір вектора можна визначити в будь-який момент. Легко переконатися, що це властивість виявляється досить корисним у багатьох ситуаціях.

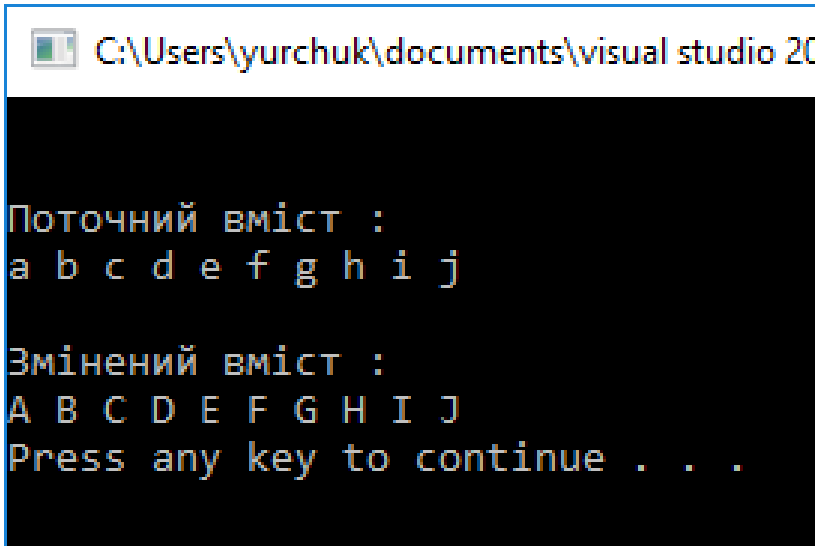
Доступ до елементів вектора за допомогою ітератора

Як відомо, масиви і вказівників тісно пов'язані між собою. Доступ до елемента масиву здійснюється або через вказівник, або за індексом. В бібліотеці STL' спостерігається аналогічна ситуація, тільки ролі масивів і покажчиків грають вектори і ітератори. Доступ до елементів вектора забезпечується за індексом або через итератор. Розглянемо відповідний приклад

```
//Доступ до елементів вектора  за допомогою ітератора.
#include <iostream>
#include "windows.h"
#include <vector>
#include <cctype>
using namespace std;

int main()
{
        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
        vector<char> v(10); // Створюємо вектор з 10 елементів.
        vector<char>::iterator p; //Створюємо ітератор.
        int i;

        //Присвоюємо елементам вектора певні значення.
        p = v.begin();
        i = 0;
        while (p != v.end()) {
```

```
            *p = i + 'a';
            p++;
            i++;
        }
        cout << "\n\n";

        //Виводимо на екран вміст вектора.
        cout << "Поточний вміст : \n";
            p = v.begin();
        while (p != v.end()) {
            cout << *p << " ";
            p++;
        }
        cout << "\n\n";

        //Змінюємо вміст вектора
        p = v.begin();
        while (p != v.end()) {
            *p = toupper(*p);
            p++;
        }
        //Виводимо вміст векторана екран .
        cout << "Змінений вміст : \n";
            p = v.begin();
        while (p != v.end()) {
            cout << *p << " ";
            p++;
        }
        cout << endl;

        system("pause");
        return 0;
}
```

Результати роботи цієї програми наведені нижче.



Зверніть увагу на те, як що, як повідомлений ітератор p. Тип **iterator** визначено в контейнерному класі. Таким чином, отримати ітератор для елементів конкретного контейнера, необхідно використовувати оголошення, аналогічне показаному в прикладі: просто вказувати ім'я контейнера і специфікатор **iterator**. У даній програмі итератор **p** встановлений в початок вектора. Для цього використовується функція **begin()**. Ця процедура аналогічна застосування покажчиків для доступу до елементів масиву. Для того щоб розпізнати кінець вектора, викликається функція **end().** Ця функція повертає ітератор, установлений на елемент, наступний за останнім елементом вектора. Таким чином, якщо ітератор **p** дорівнює значенню **v.end()**, значить, виявлений кінець вектора.

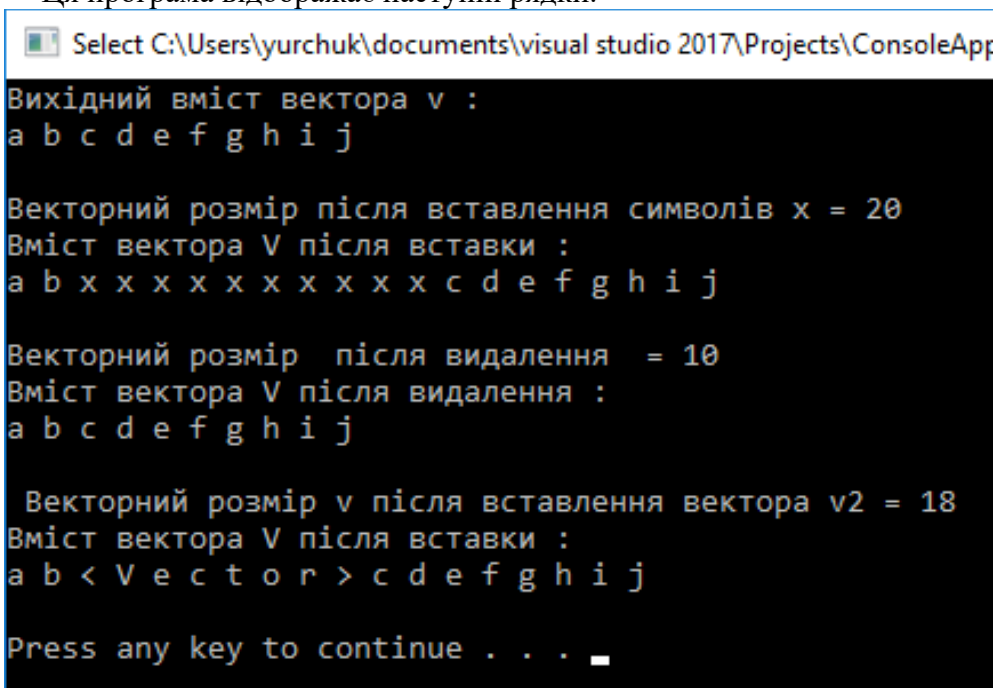
Вставка і видалення елементів вектора

Отже, ми вже знаємо, як вставити нові значення в кінець вектора. Тепер подивимось, як їх записати в середину. Для цього застосовується функція **insert().**-Видалити елементи з вектора можна за допомогою функції **erase()**. Розглянемо програму, яка ілюструє застосування функцій **insert()** і **erase()**.

```
// Ілюстрація функцій insert і erase

#include "windows.h"
#include <iostream>
#include <vector>
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	vector<char> v(10);
	vector<char> v2;
	char str[] = "<Vector>";
	int i;

	//Ініціалізуємо вектор
	for (i = 0; i< 10; i++) v[i] = i + 'a';

	//Скопіюємо символів із рядка str у вектор v2
	for (i = 0; str[i]; i++) v2.push_back(str[i]);

	//Відображення на екрані вихідного вмісту вектора
	cout << "Вихідний вміст вектора v : \n";
		for (i = 0; i< v.size(); i++) cout << v[i] << " ";
	cout << "\n\n";

	vector<char>:: iterator p = v.begin();
	p += 2; // встановлюємо літератор на третій елемент.

			// Вставляємо 10 елементів ‘x’ в вектор v.
	v.insert(p, 10, 'x');

	//Виводимо на екран вміст вектора після вставки.
	cout << "Векторний розмір після вставлення символів x = "
		<< v.size() << endl;
	cout << "Вміст вектора V після вставки : \n";
	for (i = 0; i< v.size(); i++) cout << v[i] << " ";
	cout << "\n\n";

	//Видалити ці елементи.
	p = v.begin();
	p += 2; // Встановлюємо ітератор на третій елементю.
	v.erase(p, p + 10); // Видалити наступних 10 елементів.
					//Відображення на екрані вмісту вектора після
видалення.
	cout << "Векторний розмір  після видалення  = "
		<< v.size() << endl;
	cout << "Вміст вектора V після видалення : \n";
	for (i = 0; i< v.size(); i++) cout << v[i] << " ";
	cout << "\n\n";

	//Вставка  вектора v2 у  вектор v.
	p = v.begin();
	p += 2;
	v.insert(p, v2.begin(), v2.end());

	cout << " Векторний розмір v після вставлення вектора v2 = ";
	cout << v.size() << endl;
	cout << "Вміст вектора V після вставки : \n";
```

```
        for (i = 0; i< v.size(); i++) cout << v[i] <<" ";
        cout << "\n\n";
        system("pause");
        return 0;
}
```

Ця програма відображає наступні рядки.



Ця програма демонструє застосування двох видів функції **insert()**. В першому випадку вона вставляє в вектор 10 символів ‘x’. У другому - вставляє в вектор v вміст вектора v2. Друга форма функції **insert()** набагато цікавіше. Вона має три аргументи, які є ітераторами. Перший з них задає позицію, з якої починається вставка елементів в контейнер. Наступні два аргументи зада¬ют початок і кінець вставляемой послідовності.

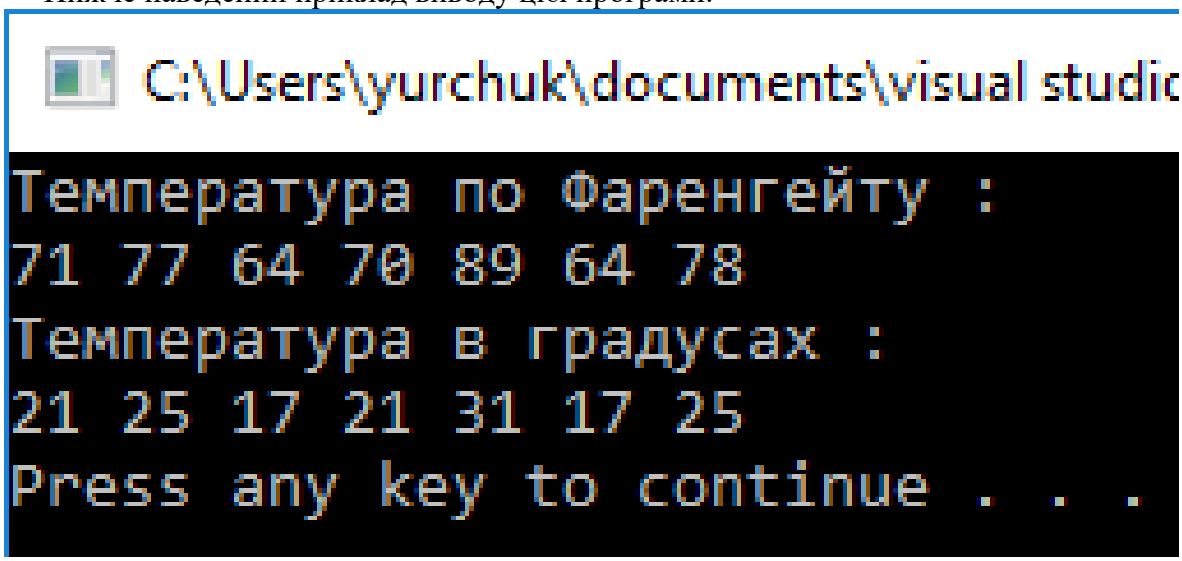
Вектор, що містить об'єкти класу

У попередніх прикладах вектор містив об'єкти вбудованих типів, однак клас **vector** володіє більш широкими можливостями. Вектор може складатися з об'єктів любого типу, включаючи об'єкти класів, визначених програмістом. Розглянемо приклад, в якому вектор зберігає дані про температуру повітря, виміряної протягом тижня. зверніть увагу на те, що клас **DailyTemp** має конструктор за замовчуванням, а також перевантажені версії операторів "<" і "==". Пам'ятайте, що в залежності від конкретної реалізації бібліотеки STL перевантаження операторів порівняння може бути необов'язковим.

```
// Вектор, який містить об'єкти класу.
#include <iostream>
#include <vector>
#include <cstdlib>
#include "windows.h"
using namespace std;

class DailyTemp {
        int temp;
public:
        DailyTemp() { temp = 0; }
        DailyTemp(int x) { temp = x; }

        DailyTemp &operator=(int x) {
                temp = x; return*this;
        }

        double get_temp() { return temp; }

};
```

```
bool operator<(DailyTemp a, DailyTemp b)
{
	return a.get_temp() < b.get_temp();
}

bool operator==(DailyTemp a, DailyTemp b)
{
	return a.get_temp() == b.get_temp();
}

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	vector<DailyTemp> v;
	int i;

	for (i = 0; i<7; i++)
	v.push_back(DailyTemp(60 + rand() % 30));
	cout << "Температура по Фаренгейту : \n";
		for (i = 0; i<v.size(); i++)
	cout << v[i].get_temp() << " ";

	cout << endl;

	//Перетворення шкали Фаренгейта в шкалу Цельсія.
	for (i = 0; i<v.size(); i++)
	v[i] = (v[i].get_temp() - 32) * 5 / 9;
	cout << "Температура в градусах : \n";
	for (i = 0; i<v.size(); i++)
	cout << v[i].get_temp() << " ";
	cout << endl;
	system("pause");
	return 0;
	}
```

Нижче наведений приклад виводу цієї програми.



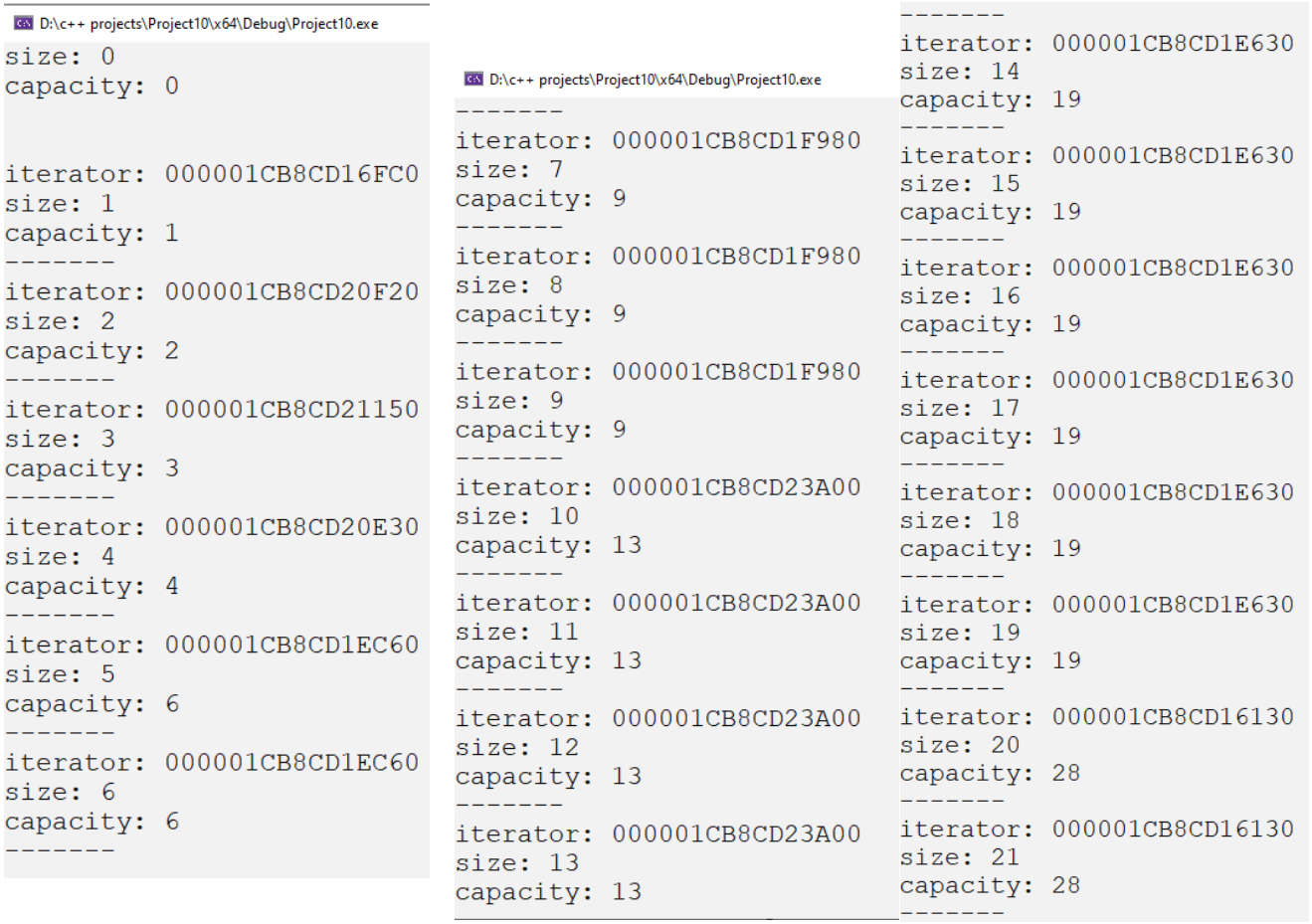
Векторам притаманна велика міць, безпека і гнучкість, але, на жаль, вони менш ефективні, ніж звичайні масиви. Отже, в багатьох ситуаціях треба віддати перевагу звичайним масивам. Однак потрібно мати на увазі, що в деяких випадках переваги класу **vector** перевищують його недоліки.

Особливості роботи з vector

Розглянемо код:

```
vector <int> s ;
	cout << "size: " << s.size()<<endl;
	cout << "capacity: " << s.capacity() << endl;

	for (int i = 0; i < 25; i++) {
		s.push_back(7);
		cout << "iterator: " << &(* s.begin()) << endl;
		cout << "size: " << s.size() << endl;
		cout << "capacity: " << s.capacity() << endl;
		cout << "-------" << endl;
	}
```

Ось що ми зможемо побачити при виконанні коду:



size() vs capacity() у std::vector

- **`size()`** — скільки елементів **фактично зберігає** вектор.
- **`capacity()`** — скільки елементів вектор **може зберігати, не перевиділяючи пам'ять**.

Іншими словами:

- `size()` — "скільки вже є"
- `capacity()` — "скільки готовий вмістити, перш ніж розширюватись"

```
vector<int> s;
```

```
cout << "size: " << s.size() << endl;
cout << "capacity: " << s.capacity() << endl;
```

Після `push_back()` — `size` зростає на 1 щоразу, а `capacity` збільшується **ривками** — не кожного разу, а коли не вистачає місця.

Стандарт не вимагає точної формули, **але зазвичай** реалізації (наприклад, в GCC або MSVC) збільшують `capacity()` за принципом **подвоєння**, щоби мінімізувати кількість realok'а:

```
Capacity: 0 → 1 → 2 → 3 → 4 → 6 → 9 → 13...
```

Тобто, коли вектору не вистачає місця — він:

1. Виділяє новий блок пам'яті (більший).
2. Копіює туди старі елементи.
3. Звільняє стару пам'ять.

Це дорога операція, тому й росте "стрибками", щоб не робити цього часто.

`capacity()` потрібен, щоб оптимізувати:

```
vector<int> v;
v.reserve(100); // одразу виділяє місце на 100 елементів
```

Це зручно, коли ти **знаєш наперед**, скільки елементів буде, — тоді не буде зайвих перевизначень.

Таким чином

```
for (int i = 0; i < 25; i++) {
    s.push_back(7);
    cout << "iterator: " << &(* s.begin()) << endl;
    cout << "size: " << s.size() << endl;
    cout << "capacity: " << s.capacity() << endl;
    cout << "-------" << endl;
}
```

На кожному кроці:

- `size` збільшується на 1.
- `capacity` стрибає: `0 → 1 → 2 → 3 → 4 → 6 → 9 → 13`
- Адреса `s.begin()` **змінюється**, коли вектор перевизначає пам'ять.

Відмінності `insert()` vs `emplace()` у `std::vector`

Характеристика	`insert()`	`emplace()`
Аргументи	Готовий об'єкт (`T`)	Аргументи конструктора (`T(args...)`)

Характеристика	`insert()`	`emplace()`
Принцип дії	Копіює або переміщує об'єкт у вектор	Створює об'єкт **на місці** всередині вектора
Продуктивність	Може викликати копіювання або переміщення	Часто **швидше**: об'єкт створюється одразу у векторі
Виклики конструкторів	Конструктор → (копіювання або переміщення)	Лише **конструктор**, без копій/переміщень
Коли використовувати	Коли об’єкт уже створений	Коли об’єкт ще не створений (передаєш аргументи)

Приклад для `vector`:

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

struct User {
    User(int id, std::string name) {
        std::cout << "Constructor called: " << name << std::endl;
    }
};

int main() {
    std::vector<User> users;

    // insert: створюємо об’єкт заздалегідь
    User u1(1, "Volodar");
    users.insert(users.begin(), u1); // Може скопіювати

    // emplace: створення напряму у векторі
    users.emplace(users.begin(), 2, "Sviatoslav"); // Швидше, без копій

    return 0;
}
```

Вивід покаже, що `emplace` не викликає копіювання, а лише конструктор.

Підсумок:

- `insert(obj)` — вставляє **вже створений об'єкт**
- `emplace(args...)` — створює **новий об'єкт прямо у векторі**

Асоціативні контейнери

Клас **map** створює асоціативний контейнер, в якому кожному ключу відповідає єдине значення. По суті, ключ являє собою ім'я, за допомогою якого можна отримати необхідне значення. Якщо в контейнері зберігається деяке значення, доступ до нього можливий тільки через ключ. Таким чином, асоціативний контейнер фактично зберігає пари ключ-значення. Перевага асоціативних масивів полягає в доступі до значень за їх ключам. Наприклад, можна створити асоціативний контейнер, в якому ключем є ім'я людини, а значенням - номер його телефону. В даний час асоціативні контейнери отримують все більш широке застосування.

Як вказувалося раніше, асоціативні контейнери можуть містити лише унікальние ключі. Дублікати не допускаються. Якщо необхідно створити асоціативний контейнер, в якому можна зберігати дублікати, слід застосовувати клас **multimap**.

Шаблонна специфікація класу **map** має наступний вигляд.

template <class Key, class T, class Comp = less<Key>,
class Allocator = allocator<pair<const key,T> > class map

Тут клас **Key** визначає тип ключа, шаблонний параметр T задає тип даних, що зберігаються в асоціативному масиві, а функція **Comp** дозволяє порівнювати два ключа. За умовчанням як функції **Comp** застосовується стандартний функтор **less()**.. Розподільник пам'яті задається класом **Allocator**, причому за замовчуванням використовується стандартний клас **allocator**.

Клас **map** має наступні конструктори.

explicit map(const Comp &*cmpfn*= Comp(),
const Allocator &*a* = Allocator());
map(const map<Key, T, Comp, Allocator> &*ab*);
template <class InIter> list(InIter *start,*InIter *end,*
const Comp &*cmpfn*= Comp (),
const Allocator &*a* = Allocator());

Перша версія конструктора створює порожній асоціативний масив. Друга - ассоціатівний контейнер, що містить елементи об'єкта оЬ. Третій варіант конструктора створює асоціативний масив, що складається з елементів, що лежать в діапазоні, заданому ітераторами *start* і *end*. Функція *cmpfn* визначає порядок проходження елементів масиву.

Як правило, будь-який об'єкт, що використовується як ключа, повинен опреде¬лять конструктор за замовчуванням, а також оператор "<" та інші оператори порівняння. Специфічні вимоги, пропоновані до ключам, залежать від компілятора.

У класі **map** визначені наступні оператори порівняння:

==, <. <-, 1 =,>,> =

Деякі функції-члени, визначені в класі кулю, перераховані в табл. 24.4. В цій таблиці клас **key_type** являє собою тип ключа, а клас **value_type** визначає тип раіг <Key, T>.

Таблиця 24.4..Функції, визначені в класі map

Функція-член	**Опис**
iterator begin(); const_iterator begin() const;	Повертає ітератор, установлений на перший елемент асоціативного масиву
void clear();	Видаляє з асоціативного масиву всі елементи
size_type count (const key_type &*k*) const;	Повертає поточну кількість дублікатів елемента із значенням до в асоціативному масиві
bool empty() const;	Повертає значення true, якщо асоціативний масив порожній, в іншому випадку повертає значення false
iterator end();	Повертає ітератор, установлений на перший елемент асоціативного масиву

const_iterator end() const;	
void erase(iterator i);	Видаляє елемент, на який посилається ітератор *i*. Повертає ітератор, установлений на елемент, наступний за видаленим
void erase (iterator start, iterator end);	Видаляє елементи з діапазону, заданого ітераторами *start* і *end*. Повертає ітератор, установлений на елемент, наступний за останнім віддаленим
size_type erase (const key_type &k);	Видаляє з асоціативного масиву елементи, що мають значення *k*
iterator find(const key_type &k); const_iterator find (const key_type &k);	Повертає ітератор, установлений на вказаний ключ. Якщо ключ не знайдений, повертається итератор, встановлений на кінець масиву
iterator insert (iterator i, const value_type &val);	Вставляє елемент зі значенням *val* на місце або відразу після елементу, на який посилається ітератор *i*. Повертається итератор, встановлений на цей елемент.
template <class inIter> void insert(InIter start, InIter end);	Вставляє елементи з діапазону, заданого ітератоами *start* і *end*.
pair<iterator,bool> insert(const value_type &val);	Вставляє елемент з значенням *val* на місце або відразу після елемента, на який посилається ітератор. Елемент вставляється, тільки якщо його ще не було є асоціативному масиві. У разі успіху повертається об'єкт класу **pair<iterator, true>.** В іншому випадку повертається об'єкт класу **pair<iterator, false>.**
mapped_type& operator[] (const key_type &i);	Повертає посилання на елемент, заданий ітератором *i*. Якщо елемента в масиві не було, він вставляється туди.
size_type size() const;	Повертає поточну кількість елементів вектора

Пари ключ-значення зберігаються в асоціативному масиві як об'єкти типу **pair**. Його шаблонна специфікація має наступний вигляд.

```
template <class Ktype, class Vtype> struct pair {
      typedef Ktype first_type;  // Тип ключа
      typedef Vtype second_type; // Тип значення
      Ktype first;                     // Містить ключ
      Vtype second;                     // Містить значення

                                                        // Конструктори
      pair();
      pair(const Ktype &k, const Vtype &v);
      template<class A, class B> pair(const<A, B> &ob);
   };
```

Як підказують коментарі, значення, що зберігається в поле **first**, містить ключ, а поле **second** зберігає значення, відповідне цьому ключу.

Пару можна створити, викликавши або конструктори класу **pair**, або функцію **make_pair()**, яка створює об'єкти класу **pair** на основі інформації про тип параметрів. Функція **make_pair()** є узагальненою. Її прототип наведено нижче.

```
template <class Ktype, class Vtype>
  pair<Ktype, Vtype> make_pair(const Ktype &k, Vtype &v);
```

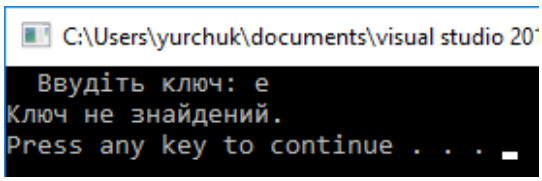
Як бачимо, ця функція повертає об'єкт класу **pair**, що складається з пари значень типів *Ktype* і *Vtype*. Перевага функції **make_pair()** полягає в тому, що типи зберігаються об'єктів визначаються компілятором автоматично, і задавати його явно не потрібно.

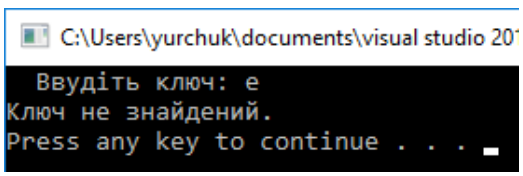
Наступна програма ілюструє основні операції над асоціативним масивам, в якому зберігаються пари значень, що визначають взаємну відповідність між прописними буквами і їх ASCII-кодами. Таким чином, ключем є символ, а значенням - ціле число. Пари ключ-значення мають наступний вигляд:

A 65
B 66
C 67

Коли користувач вводить ключ на екран виводить його ASCII-код

```
// Приклад простого асоціативного масиву.
#include <iostream>
#include <map>
#include "windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	map<char, int>m;
	int i;

	// Записуємо пари в асоціативний масив.
	for (i = 0; i<26; i++) {
		m.insert(pair<char, int>(' A' + i, 65 + i));
	}

	char ch;
	cout << "  Ввудіть ключ: ";
	cin >> ch;

	map<char, int>::iterator p;

	// Знайти значення по заданому ключу.
	p = m.find(ch);
	if (p != m.end())
		cout << "  ASCII-код ключа рівний " << p->second<<endl;
	else
		cout << "Ключ не знайдений.\n";
	system("pause");
		return 0;
    }
```





Зверніть увагу на те, що для створення пари ключ-значення застосовується шаблонний клас **pair**. Тип даних, визначений цим класом, повинен відповідати типу асоціативного масиву, в який вставляються пари ключ-значення.

Після ініціалізації асоціативного масиву будь-яке значення можна знайти за його ключу. Для цього слід викликати функцію **find(),** яка повертає ітератор, встановлений на необхідний

елемент або на кінець масиву, якщо потрібного елементу в масиві не виявилося. Значення, пов'язане з знайденим ключем, міститься в полі **second** класу **pair**.

У попередньому прикладі пара ключ-значення створювалася явним чином з допомогою конструктора класу **pair <char, int>.** Поряд з цим можна застосовувати функцію **make_pair(),** яка створює пари ключ-значення на основі інформації про тип параметрів. Наприклад, в попередній програмі оператор

```
//m.insert(make_pair((char)('A' + i), 65 + i);
```

також вставляв би пари ключ-значення в об'єкт **m**. Для того щоб запобігти автоматичне перетворення результату складання i+'A' в тип **int** необхідно виконати приведення до типу char. В іншому випадку тип визначався б автоматично.

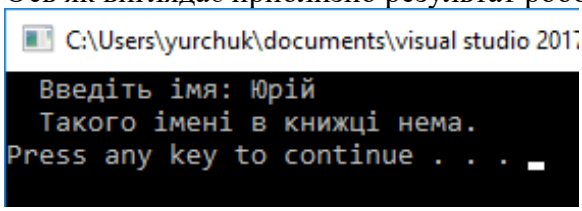
Асоціативний масив, що містить об'єкти

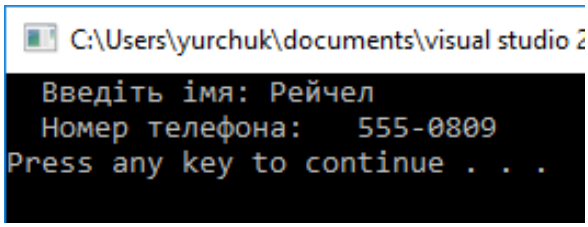
Як і будь-який інший контейнер, асоціативний масив можна використовувати для зберігання об'єктів класів, визначених програмістом. Наприклад, наступна програма створює просту телефонну книжку. Інакше кажучи, вона створює ассоціативний масив імен, пов'язаних з номерами. Для цього використовуються два класи з іменами **name** і **number**. Оскільки асоціативний масив зберігає впорядкований список ключів, в програмі визначається оператор "<" для порівняння об'єктів класу **name**. Як правило, оператор "<" необхідно визначати в будь-якому класі, об'єкти якого використовуються в якості ключів. (Деякі компілятори вимагають, щоб в таких класах були визначені додаткові оператори порівняння.)

```
// Приклад асоціативного масиву.
// для створення телефонної книжки.
#include <iostream>
#include <map>
#include <cstring>
#include "windows.h"
using namespace std;

class name {
	char str[40];
public:
	name() { strcpy(str, "  "); }
	name(char *s) { strcpy(str, s); }
	char *get() { return str; }
};
// Визначаєм оператор < для обєктів класа name.
bool operator<(name a, name b)
{
	return strcmp(a.get(), b.get())<0;
}
class phoneNum {
	char str[80];
public:
	phoneNum() { strcmp(str, "  "); }
	phoneNum(char *s) { strcpy(str, s); }
	char *get() { return str; }
};

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	map<name, phoneNum> directory;

	// Вносимо імена і номера в асоціативний масив.
	directory.insert(pair<name, phoneNum>(name("Том"), phoneNum("  555-4533")));
	directory.insert(pair<name, phoneNum>(name("  Крис"),phoneNum("  555-9678")));
	directory.insert(pair<name, phoneNum>(name("Джон"), phoneNum("  555-8195")));
	directory.insert(pair<name, phoneNum>(name("Рейчел"), phoneNum("  555-0809")));
```

```
    // Знаходим номер телефона по вказаному імені.
    char str[80];
    cout << "  Введіть імя: ";
    cin >> str;
    map<name, phoneNum>::iterator p;

    p = directory.find(name(str));
    if (p != directory.end())
           cout << "  Номер телефона: " << p->second.get()<<endl;
    else
           cout << "  Такого імені в книжці нема. \n";
    system("pause");
    return 0;
}
```

Ось як виглядає приблизно результат роботи програми





В даному випадку кожен елемент асоціативного масиву представляє собою символьний масив, в якому зберігається рядок, що завершується нульовим байтом. Пізніше ми знайдемо більш простий спосіб вирішити описану задачу за допомогою стандартного типу s**tring**.