Списки

Клас **list** створює двонаправлений лінійний список. На відміну від вектора, який надає довільний доступ до своїх елементів, доступ до елементів списка може бути лише послідовним. Оскільки список є двонаправленим, можна перемішатися від початку до кінця і від кінця до початку списку.

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

template <class T, class Allocetor = allocator<T> > class list

Тут шаблонний параметр T задає тип даних, що зберігаються в списку. Розділювач пам'яті задається класом **Allocator**, причому за замовчуванням використовується стандартний механізм розподілу пам'яті. Клас **list** має наступні конструктори

explicit list (const Allocator &a = Allocator ( )) ;
explicit list (size_type num, const T &val = T( ) ,
const Allocator &a = Allocator ( )) ;
list (const list<T,Allocator> &ab) ;
template <class InIter>list (InIter *start* , InIter *end* ,
const Allocator &a = Allocator ( )) ;

Перша версія конструктора створює порожній список. Друга - список, що складається з *num* елементів, що мають значення *val*, причому це значення можна задавати за замовчуванням. Третій варіант конструктора створює список, що містить елемент об'єкта *ob*. Четверта версія конструктора формує список, що складається з елементів, що лежать в діапазоні, заданому ітераторами *start* і *end*.

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

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

Найбільш важливі функції-члени, визначені в класі **list**, перераховані в табл. 24.3. Як і в класі **vector**, функція **push_back()** записує значення в кінець списку. Для вставки нового елемента в початок списку призначена функція **push_front().** За допомогою функції **insert()** значення можна записати в середину списку. Два списки можна об'єднати, викликавши функцію **splice()**. Крім того, за допомогою функції **merge()** один список можна впровадити в інший.

Таблиця 24.3. Функції, визначені в класі list

Функція-член	Опис
reference_back ( ) ; const_reference back( ) const;	Повертає посилання на останній елемент списку
iterator begin ( ) ; const_iterator begin( ) const;	Повертає ітератор, установлений на перший елемент списку
void clear ( ) ;	Видаляє зі списку всі елементи
bool empty ( ) const;	Повертає значення true, якщо список порожній, в протилежному випадку повертає значення fa1se
iterator end ( ) ; const_iterator end ( ) const;	Повертає ітератор, установлений на перший елемент списку
iterator erase (iterator i) ;	Видаляє елемента, на який посилається ітератор *i*. Повертає ітератор, установлений на елемент, следующий за видаленим
iterator erase(iterator *start*, iterator *end*);	Видаляє елементи з діапазону, заданого ітераторами *start* і *end*. Повертає ітератор, установлений на елемент, наступний за останнім віддаленим
reference front ( ) ; const_reference front ( ) const;	Повертає посилання на перший елемент списку
iterator incert (iterator i, const T &val) ;	Вставляє елемент *val* перед елементом, на який посилається ітератор *i*. Повертає ітератор, установлений на елемент *val*
void insert (iterator i,	

size_type num, const T &val) ; template <class inIter> void insert (iterator i, InIter start, InIter end) ;	Вставляє *num* копій елементу *val* перед елементом, на який посилається ітератор *i* Вставляє перед елементом, на який посилається ітератор i, послідовність елементів, задану ітераторами *start* і *end*/
void merge (list<T,Allocator> &ob); template <class Comp> void merge(list<T,Allocator> &ob, Comp cmpfn);	Впроваджує впорядкований список, що міститься до об'єкті ob, в заданий список. В результаті виникає впорядкований-ний список. Після впровадження список, що міститься в об'єкті *ob*, стає порожнім. Друга версія функції тегде отримує як параметр функцію, що дозволяє порівнювати елементи списку між собою
void pop_back ( ) ;	Видаляє останній елемента списку
void pop_front ( ) ;	Видаляє перший елемент списку
void pop_back (const T &val) ;	Додає елемент val в кінець списку
void pop_front (const T &val) ;	Додає елемент val в початок списку
void remove (const T &val) ;	Видаляє зі списку елементи, значення яких дорівнює val
void reverse ( ) ;	Міняє порядок слідування елементів списку на протилежний
size_type size ( ) const;	Повертає поточну кількість елементів списку
void sort( ) ; template <class Comp> void sort (Comp cmpfn) ;	Впорядковує список. Друга версія впорядковує список, використовуючи для порівняння елементів функцію cmpfn
void splice (iterator I, list<T,Allocator> &ob) ;	Вставляє в позицію, вказану літератором *i*, вміст об'єкта ob. Після виконання операції список ob становиться порожнім
void splice (iterator I, list<T,Allocator> &ob, iterator el) ;	Елемент, на який посилається ітератор I, видаляється зі списку ob і вставляється в викликає список в позицію, задану ітератором el
void splice (iterator I, list<T,Allocator> &ob, iterator start, iterator end) ;	Елементи списку зі, розташовані в діапазоні, заданому ітераторами 8 (аг1і ЕПБ, видаляються зі списку зі і вставляют¬ся в викликає список в позицію, задану ітератором /

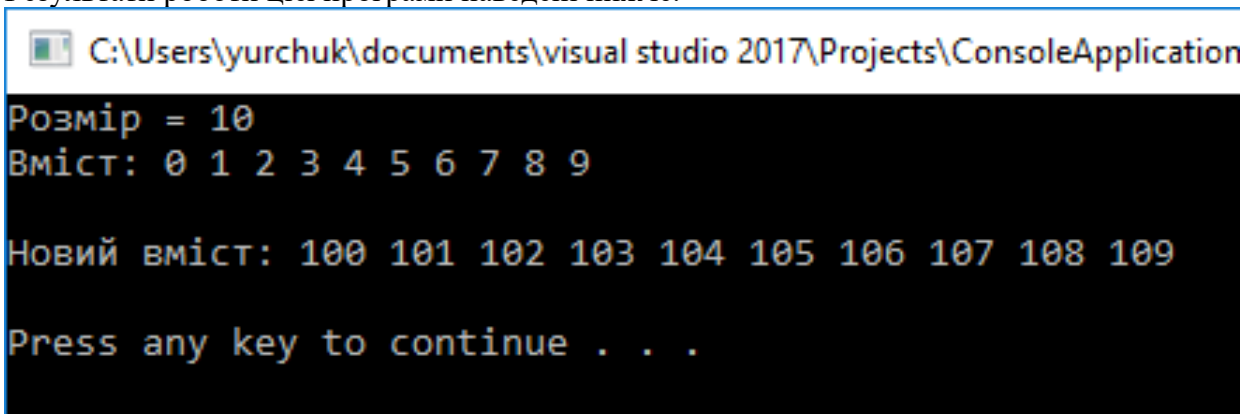
Для досягнення гнучкості і машинонезалежності будь-який об'єкт, що поміщається в список, повинен мати конструктор за замовчуванням. Крім того, в ньому повинен бути визначений оператор "<" і, можливо, інші оператори порівняння. Точні вимогги, які ставляться до об'єктів, залежать від конкретного компілятора.

Розглянемо приклад, який ілюструє основні операції над списками.

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

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	list<int> lst; // Створюємо порожній список.
	int i;
	for (i = 0; i<10; i++) lst.push_back(i);

	cout << "Розмір = " << lst.size() << endl;
```

```
        cout << "Вміст: ";
        list<int>::iterator p = lst.begin();
        while (p != lst.end()) {
               cout << *p << " ";
               p++;
        }
        cout << "\n\n";

        // Змінюємо вміст списку.
        p = lst.begin();
        while (p != lst.end()) {
               *p = *p + 100;
               p++;
        }

        cout << "Новий вміст: ";
p = lst.begin();
        while (p != lst.end()) {
               cout << *p << " ";
               p++;
        }
        cout << "\n\n";
        system("pause");
        return 0;
    }
```

Результати роботи цієї програми наведені нижче.



Ця програма створює список цілих чисел. Спочатку формується порожній список. Потім за допомогою функції **push_back()** в нього записуються 10 цілих чисел, причому кожне наступне число записується в кінець існуючого списку. Після цього на екран виводяться розмір і вміст самого списку. Для виведення вмісту списку на екран використовується ітератор.

```
list<int>::iterator p = lst.begin();
while (p != lst.end () )  {
  cout << *p << " ";
  p++;
}
```

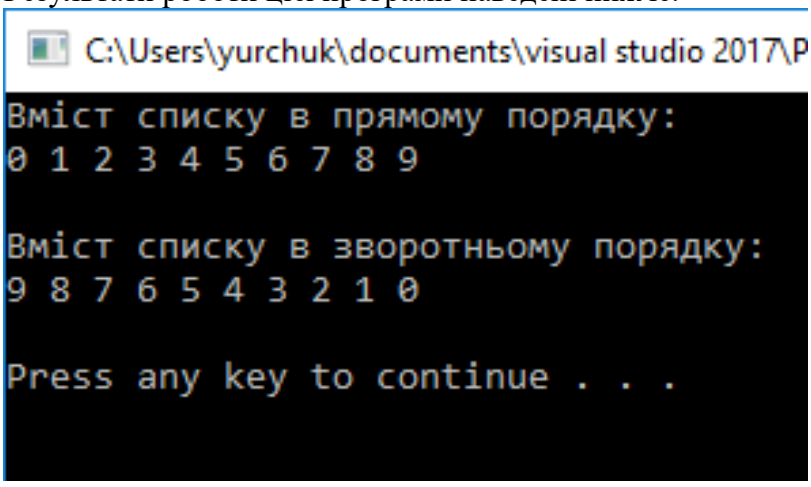
В цьому фрагменті итератор p спочатку встановлюється на першу позицію списку. За¬тем при кожному проході циклу итератор p збільшується на одиницю і переміщається на наступний елемент. Виконання циклу завершується, коли итератор p вказує на кінець списку. По суті, цей цикл не відрізняється від циклу, застосованого вище для переміщення по вектору. Такий цикл широко використовується в бібліотеці SLT. Той факт, що одну й ту ж конструкцію можна застосовувати до різних типів контейнеров, свідчить про мощі бібліотеки SLT.

Функція end()

Настав час описати одну досить несподівану властивість функції **end().** Вона повертає вказівник не на останній елемент контейнера, а на наступний за ним елемент. Таким чином, останньому елементу контейнера відповідає значення **end()-1.** Це дозволяє створювати дуже ефективні алгоритми обходу всіх елементів контейнера, включаючи останній елемент. Якщо значення ітератора стає рівним значенню функції **end(),** значить, всі елементи контейнера пройдені. Це властивість завжди слід мати на увазі, оскільки воно досить неприродно. Розглянемо наступну програму, яка виводить на екран елементи списку в прямому і зворотному порядку.

```
// Функція end ().
#include <iostream>
#include <list>
#include "windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	list<int> lst; // Створюєм порожній список.
	int i;

	for (i = 0; i<10; i++) lst.push_back(i);

	cout << "Вміст списку в прямому порядку:\n";
	list<int>::iterator p = lst.begin();
	while (p != lst.end()) {
		cout << *p << " ";
		p++;
	}
	cout << "\n\n";
	cout << "Вміст списку в зворотньому порядку:\n";
	p = lst.end();
	while (p != lst.begin()) {
		p--;  // Зменшує вказівник.
		cout << *p << " ";
	}
	cout << "\n\n";
	system("pause");
	return 0;
}
```

Результати роботи цієї програми наведені нижче.



Фрагмент коду, призначений для виводу вмісту списку на екран, повністю збігається з попередніми прикладами. Однак слід звернути особливу увагу на фрагмент коду, який виводить вміст списку у зворотному порядку. Спочатку ітератор p за допомогою функції встановлюється на кінець списку. Оскільки функція **end()** повертає ітератор, установлений на об'єкт, розташований за останнім об'єктом у списку, перед використанням ітератор p слід зменшити на одиницю. Саме з цієї причини ітератор p в циклі виводу зменшується перед

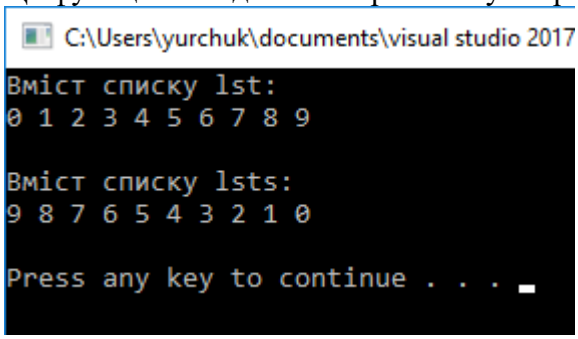
виконанням оператора **cout**, а не після. Ніколи не слід забувати, що функція **end()** повертається курсор не на останній елемент контейнера, а на наступний за ним елемент.

Порівняння функцій push_front() і push_back()

Список можна створювати, додаючи елементи або в його кінець, або в початок. До цих пір ми додавали елементи тільки в кінець списку і застосовували для цього функцію **push_back()**. Для того, щоб додати елементи в початок списку, слід викликати функцію **push_front()**. Розглянемо приклад.

```
/* Демонстрація відмінностей між функціями
push_back() и push_front().  */
#include <iostream>
#include <list>
#include "windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	list<int> lst, lsts;

	int i;

	for (i = 0; i<10; i++) lst.push_back(i);

	for (i = 0; i<10; i++) lsts.push_front(i);

	list<int>::iterator p;

	cout << "Вміст списку lst:\n";
	p = lst.begin();
	while (p != lst.end()) {
		cout << *p << " ";
		p++;
	}
	cout << "\n\n";

	cout << "Вміст списку lsts:\n";
	p = lsts.begin();
	while (p != lsts.end()) {
		cout << *p << " ";
		p++;
	}
	cout << "\n\n";
	system("pause");
	return 0;
}
```

Ця функція виводить на екран наступні результати



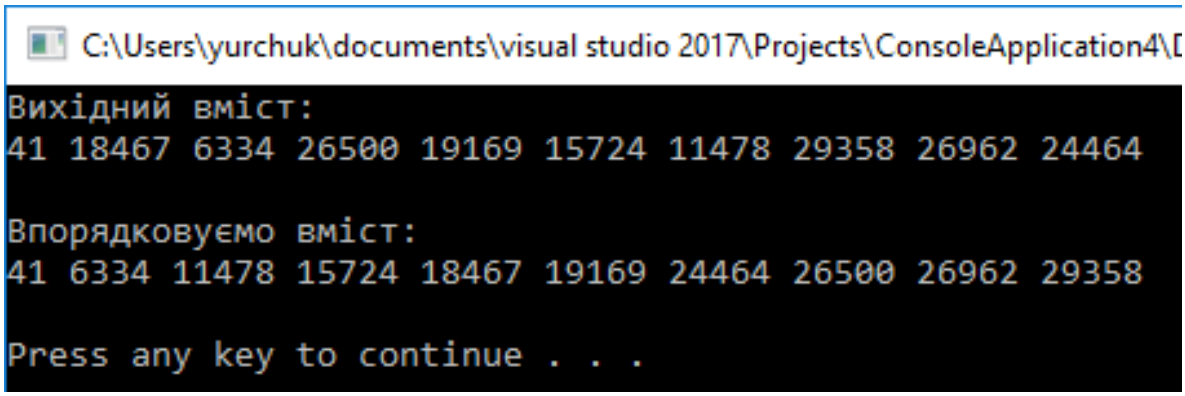
Оскільки при створенні списку **lst2** елементи додавалися в початок, порядок їх слідування протилежний порядку проходження елементів у списку **lst1**, при створенні якого вони додавалися в кінець.

Сортування списку

Список можна впорядкувати, викликавши функцію-член **sort().** Наступна програма створює список, що складається з випадкових цілих чисел, а потім сортує їх в порядку зростання.

```
// Сортування списку.
#include <iostream>
#include <list>
#include "windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	list<int> lst;
	int i;

	// Створюєм список, який складається з випадкових цілих чисел.
	for (i = 0; i<10; i++)
		lst.push_back(rand());

	cout << "Вихідний вміст:\n";
	list<int>::iterator p = lst.begin();
	while (p != lst.end()) {
		cout << *p << " ";
		p++;
	}
	cout << endl << endl;

	// Впорядковуєм список.
	lst.sort();

	cout << "Впорядковуємо вміст:\n";
	p = lst.begin();
	while (p != lst.end()) {
		cout << *p << " ";
		p++;
	}
	cout << "\n\n";
	system("pause");
	return 0;
}
```

Ось як виглядає приблизний висновок цієї програми.

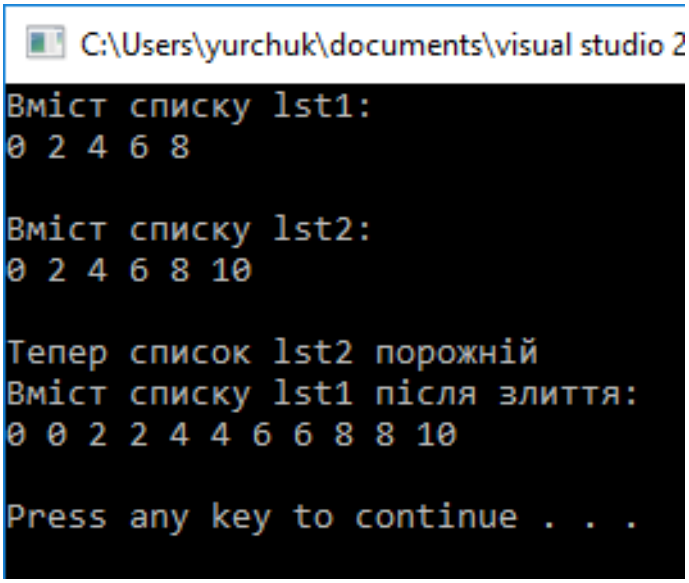


Вставка одного списку в інший

Упорядкований список можна вставити в інший список. Результатом цієї операції є упорядкований список, що складається з елементів вихідних списків. Новий список зберігається в поточному об'єкті, а другий список стає порожнім. Ця операція ілюструється таким прикладом. Перший список складається з парних чисел від 0 до 9. Другий список складається з непарних чисел. Після злиття цих спі¬сков утворюється послідовність 0 1 2 3 4 5 6 7 8 9.

```
// Злиття двох списків.
#include <iostream>
#include <list>
#include "windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	list<int> lst, lsts;
	int i;


	for (i = 0; i<10; i += 2) lst.push_back(i);

	for (i = 0; i<11; i += 2) lsts.push_back(i);

	cout << "Вміст списку lst1:\n";
	list<int>::iterator p = lst.begin();
	while (p != lst.end()) {
		cout << *p << " ";
		p++;
	}
	cout << endl << endl;

	cout << "Вміст списку lst2:\n";
	p = lsts.begin();
	while (p != lsts.end()) {
		cout << *p << " ";
		p++;
	}
	cout << endl << endl;
	// Об'єднуємо два списки.
	lst.merge(lsts);
	if (lsts.empty())
		cout << "Тепер список lst2 порожній\n";

	cout << "Вміст списку lst1 після злиття:\n";
	p = lst.begin();
	while (p != lst.end()) {
		cout << *p << " ";
		p++;
	}
	cout << "\n\n";
	system("pause");
	return 0;
}
```

Розглянемо результати роботи цієї програми.



Цей приклад має ше одну особливість, пов'язану із застосуванням функції **empty()**. Якщо викликаний контейнер порожній, вона повертає значення **true**. Зверніть увагу на те, що функція **marge()** видаляє всі елементи зі списку, які підходять вставці. Результати роботи програми це підтверджують.

УВАГА!

Метод **marge()** застосовується до відсортованих списків, і при об'єднанні списків, також відбувається сортування.

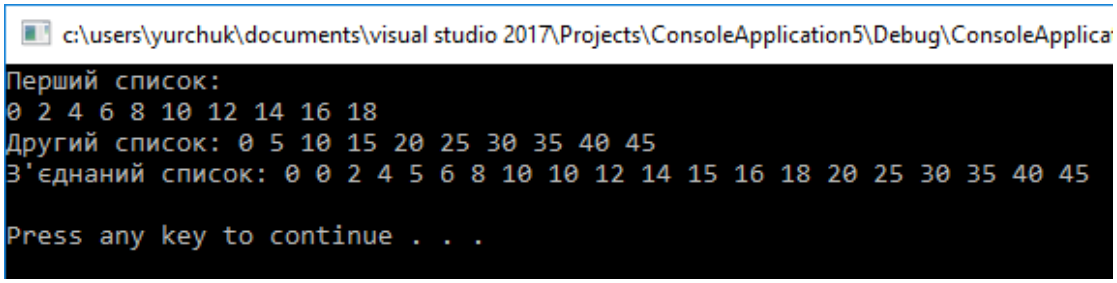
Список, що містить об'єкти класу

Розглянемо приклад, в якому використовується список об'єктів класу **myclass**. Зверніть увагу на те, що в цьому класі перевантажуються оператори "<", ">", "| =" і "==". (При роботі з деякими компіляторами це робити не обов'язково, в той же час інші компілятори змушують перевантажувати ще й додаткові оператори.) У бібліотеці STL ці операторні функції використовуються для порівняння об'єктів, що зберігаються в контейнері. Навіть якщо список не впорядкований, іноді доводиться порівнювати елементи при пошуку, сортуванні або злиття.

```
//#include "windows.h"
//system("pause");
/*SetConsoleCP(1251);
SetConsoleOutputCP(1251);*/

// Список, який містить об'єкти.
#include <iostream>
#include <list>
#include <cstring>
#include "windows.h"
using namespace std;

class myclass {
	int a, b;
	int sum;
public:
	myclass() { a = b = 0; }
	myclass(int i, int j) {
		a = i;
		b = j;
		sum = a + b;
	}
	int getsum() { return sum; }

	friend bool operator<(const myclass &o1,
		const myclass &o2);
```

```
        friend bool operator>(const myclass &o1,
                const myclass &o2);
        friend bool operator==(const myclass &o1,
                const myclass &o2);
        friend bool operator!=(const myclass &o1,
                const myclass &o2);
};
bool operator<(const myclass &o1, const myclass &o2)
{
        return o1.sum < o2.sum;
}
bool operator>(const myclass &o1, const myclass &o2)
{
        return o1.sum > o2.sum;
}
bool operator==(const myclass &o1, const myclass &o2)
{
        return o1.sum == o2.sum;
}
bool operator!=(const myclass &o1, const myclass &o2)
{
        return o1.sum != o2.sum;
}
int main()
{
        int i;

        // Створюємо перший список.
        list<myclass> lst;
        for (i = 0; i<10; i++) lst.push_back(myclass(i, i));

        cout << "Перший список: \n";
        list<myclass>::iterator p = lst.begin();
        while (p != lst.end()) {
                cout << p->getsum() << " ";
                p++;
        }
        cout << endl;

        // Створюємо другий список.
        list<myclass> lsts;
        for (i = 0; i<10; i++) lsts.push_back(myclass(i * 2, i * 3));

        cout << "Другий список: ";
        p = lsts.begin();
        while (p != lsts.end()) {
                cout << p->getsum() << " ";
                p++;
        }
        cout << endl;

        // Виконуємо злиття списків lst1 і lst2.
        lst.merge(lsts);

        // Виводимо на екран з'єднаний список.
        cout << "З'єднаний список: ";
        p = lst.begin();
        while (p != lst.end()) {
                cout << p->getsum() << " ";
                p++;
        }
        cout << "\n\n";
        system("pause");
        return 0;
}
```

Програма створює два списки, що містять об'єкти класу **myclass**, і виводить на екран вміст кожного списку. Потім вона виконує злиття списків і виводить результат, наведений нижче.



🔷 Конструктори

```
list();                          // Порожній список
list(size_type n);               // n елементів зі значенням за замовчуванням
list(size_type n, const T& val);  // n елементів із значенням val
list(iterator first, iterator last); // Копія діапазону
list(const list& other);         // Копіює інший список
list(list&& other);              // Переміщує інший список
```

🔷 Основні методи доступу

```
front();     // Повертає посилання на перший елемент
back();      // Повертає посилання на останній елемент
empty();     // Чи порожній список?
size();      // Кількість елементів
max_size();  // Максимальна можлива кількість елементів
```

🔷 Операції вставки

```
push_front(const T& val);      // Додати на початок
push_back(const T& val);       // Додати в кінець
emplace_front(args...);        // Сконструювати об'єкт на початку
emplace_back(args...);         // Сконструювати об'єкт в кінці
insert(iterator pos, const T& val); // Вставити перед pos
emplace(iterator pos, args...);     // Сконструювати перед pos
```

🔷 Операції видалення

```
pop_front();                 // Видалити перший
pop_back();                  // Видалити останній
erase(iterator pos);         // Видалити елемент за ітератором
erase(iterator first, iterator last); // Видалити діапазон
clear();                     // Очистити список
remove(const T& val);        // Видалити всі val
remove_if(pred);             // Видалити за умовою
unique();                    // Видалити послідовні дублікати
```

🔷 Операції зі списками

```
merge(list& other);          // Злиття відсортованих списків
merge(list& other, comp);    // Злиття з компаратором
splice(pos, other);          // Перемістити весь інший список у pos
```

```
splice(pos, other, it);      // Один елемент з іншого списку
splice(pos, other, first, last); // Діапазон із іншого списку
reverse();                   // Змінити порядок на зворотний
sort();                      // Сортувати
sort(comp);                  // Сортувати з компаратором
```

🔷 Оператори

```
operator=();                 // Присвоєння (копія або переміщення)
operator==, !=, <, >, <=, >=; // Порівняння списків
```

🔷 Ітератори

```
begin();          // Ітератор на перший елемент
end();            // Ітератор за останнім елементом
rbegin();         // Зворотний ітератор (з кінця)
rend();
cbegin();         // const-версії
cend();
crbegin();
crend();
```

Клас string

Як відомо, мова С ++ не передбачає вбудованого типу для рядків. Для роботи з ними існує дві можливості. По-перше, можна використовувати традиційний масив символів, що завершується нульовим байтом, з яким ми вже добре знайомі. Іноді цей масив називають С-рядком. По-друге, можна створити об'єкт класу **string**. Розглянемо цей спосіб докладніше.

Фактично клас **string** є спеціалізацією більш загального шаблонного класу **basic_string**. Насправді клас **basic_string**.має дві спеціалізації: клас **warting**, підтримуючий 8-бітові рядки, і клас **string**, призначений для роботи з рядками розширених символів. Оскільки 8- бітові рядки досі широко використовуються в програмуванні, ми розглянемо лише клас **string**.

Перш ніж перейти до вивчення класу **string**, слід розібратися, чому його включили в бібліотеку мови С ++. Стандартні класи звичайно не є частиною мови С ++. Кожне нововведення мови супроводжується тривалими і напруженими дебатами. Оскільки мова С++ вже містила засоби для роботи з масивами символів, що завершуються нульовим байтом, клас **string**, на перший погляд, виглядає винятком із цього правила. Однак це неправильно, оскільки до рядків, що завершується нульовим байтом, не можна застосувати жоден стандартний оператор мови С ++. Такі рядки не можуть бути частиною звичайних виразів. Розглянемо, наприклад, наступний фрагмент.

```
char s1 [80], s2 [80], s3 [80];

s1 = "Альфа"; //Неможна!
s2 = "Бета"; //Неможна!
s3 = s1 + s2; //Помилка, і так теж неможна!
```

Як випливає з коментарів, в мові С ++ неможливо ні привласнити масиву символів нове значення, використовуючи оператор присвоювання (за винятком ініціалі¬зації), ні застосувати оператор "+" для конкатенації двох рядків.

Ці операції слід записати, використовуючи бібліотечні функції.

```
strcpy (s1, "Альфа");
strcpy (s2, "Бета");
strcpy (s3, s1);
strcpy (s3, s2);
```

Оскільки з формальної точки зору масив символів, що завершується нульовим байтом, не є типом, до нього не можна застосовувати оператори мови С ++. Це ускладнює навіть найпростіші операції з рядками. Саме це обмеження послужило стимулом для розробки стандартного класу **string**. Слід памятати, що, визначаючи клас в мові С ++, ви формуєте новий тип даних, котрий повністю інтегрується в середу мови. Зрозуміло, це означає, що в новому класі звичайні оператори можна перевантажити. Отже, стандартний клас **string** дозволяє працювати з рядками, як з будь-яким іншим типом даних, а саме: застосовуючи до них оператори.

Однак існує й інша причина, що обгрунтовує існування стандартного класу **string**, - безпека. В руках недосвідченого або безтурботного програміста останній елемент масиву, що містить нульовий байт, абсолютно беззахисний. Розглянемо, наприклад, стандартну функцію **strcpy()**, призначену для копіювання рядків. Ця функція не передбачає ніякої перевірки кордонів результуючого масиву. Якщо вихідний масив довший результуючого, може виникнути непоправна помилка, що приводить до краху операційної системи.

Отже, існують три причини, з яких стандартний клас **string** був включений в мову С ++: логічність (тепер рядок є типом даних), зручність (до рядків можна застосовувати стандартні оператори мови С ++) і безпека (неможливо вийти за межі масиву) . Слід зауважити, що немає ніяких причин відмовлятися від традиційних масивів символів, що завершуються нульовим байтом. Вони як і раніше є найбільш ефективним способом реалізації рядків. Однак, якщо швидкодія програми не є принципово важ-ним фактором, новий клас string надає безпечний і надзвичайно зручний спосіб роботи з рядками.

Хоча традиційно клас **string** відносять до стандартної бібліотеці мови С ++, а не до бібліотеці STL, насправді він є різновидом контейнерних класів. Це значить, що він підтримує

всі алгоритми, описані в попередньому розділі. Крім цього, клас **string** володіє додатковими можливостями. Для роботи з класом **string** необхідно включити в програму заголовок <**string**>.

Клас **string** дуже великий. Він містить велику кількість конструкторів і функцій-членів. Крім того, багато функції-члени мають перевантажені версії. З цієї причини клас **string** неможливо описати в одній главі. Замість цього ми обмежимося найбільш важливими і часто вживаними властивостями. Загальне уявлення про принципи роботи класу string дозволить читачам легко розібратися з усім іншим.

Клас **string** містить декілька конструкторів. Прототипи трьох найбільш важливих конструкторів приведені нижче.

string();
string(const char *str);
string(const string &str);

Перший конструктор створює порожній об'єкт класу **string**. Другий конструктор створює об'єкт класу **string** з рядка, що завершується нульовим байтом, на яку посилається вказівник *str*. Ця версія конструктора перетворює С- рядок в об'єкт класу **string**. Третій конструктор створює об'єкт класу **string** на основі іншого об'єкта цього класу.

Для об'єктів класу **string** визначено велику кількість операторів.

Оператор	Значення
=	присвоєння
+	Конкатенация
+=	Присвоєння і конкатенація
==	рівність
!	нерівність
<	менше
<=	Менше або дорівнює
>	більше
>=	Більше або дорівнює
[]	індексація
<<	вивід
>>	Ввід

Ці оператори дозволяють використовувати об'єкти класу **string** в звичайних виразах і уникнути застосування функцій на зразок **strcpy ()** або **strcat ()**. Як правило, в одному і тому ж вираженні об'єкти класу **string** можна змішувати зі звичайними С-рядками. Наприклад, об'єкту класу **string** можна привласнити рядок, що завершується нульовим байтом.

Для конкатенації двох об'єктів класу **string**, а також об'єкта класу **string** і звичайної С-рядки можна застосовувати оператор "+". Інакше кажучи, допускаються наступні варіанти.

string+string
string+С-рядок
С-рядок+string

Крім того, за допомогою оператора "+" до кінця рядка можна приписати ще один символ.

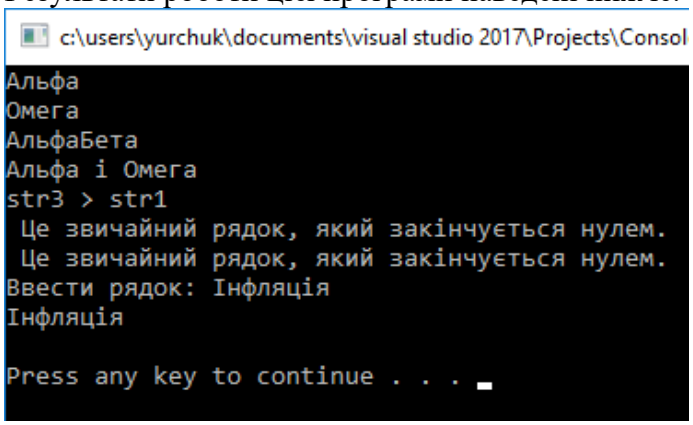
У класі **string** визначена константа **npos**, рівна -1. Ця константа задає максимально можливу довжину рядка.

Клас **string** дозволяє надзвичайно легко обробляти рядки. Наприклад, об'єктам класу **string** можна привласнювати рядка, укладені в подвійні лапки, а також порівнювати об'єкти між собою, користуючись звичайними операторами порівняння. Ці операції ілюструються наступною програмою.

```
//Короткий приклад, демонструючий клас string.
#include <iostream>
#include <string>
#include "windows.h"
using namespace std;
```

```
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	string str1("Альфа");
	string str2("Бета");
	string str3("Омега");
	string str4;
	string str6;

	// Присвоюємо один рядок іншому.
	str4 = str1;
	cout << str1 << "\n" << str3 << "\n";

	// З'єднуємо рядки.
	str4 = str1 + str2;
	cout << str4 << "\n";

	// З'єднуємо об'єкт класу string із С-рядком.
	str4 = str1 + " і " + str3;
	cout << str4 << "\n";

	//Порівнюємо рядки.
	if (str3 > str1) cout << "str3 > str1\n";
	if (str3 == str1 + str2)
		cout << "str3 == str1+str2\n";

	/* Об'єкту класу string можна присвоїти звичайний рядок. */
	str1 = " Це звичайний рядок, який закінчується нулем. \n";
	cout << str1;

	//Створює об'єкт класу string, використовуючи
	//інший об'єкт цього класу.
	string str5(str1);
	cout << str5;

	//Введення рядка.
	cout << "Ввести рядок: ";
	cin>>str6;
	cout<<str6;
	cout << "\n\n";
	system("pause");
	return 0;
}
```

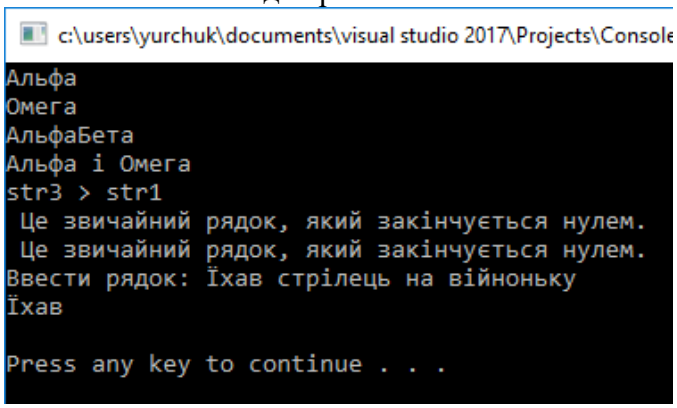
Результати роботи цієї програми наведені нижче.



Звернемо увагу на частину коду

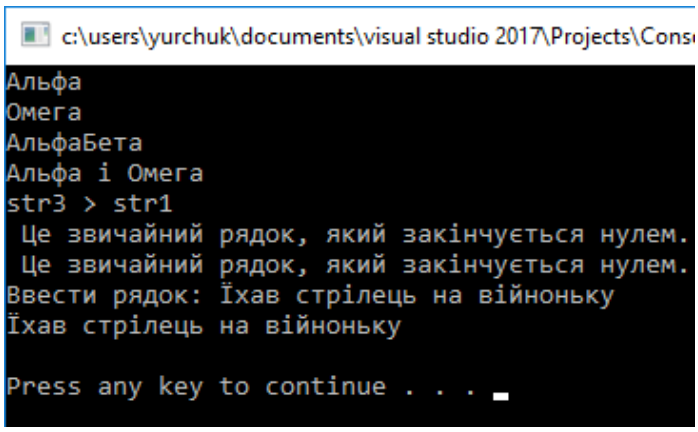
```
//Введення рядка.
	cout << "Ввести рядок: ";
	cin>>str6;
	cout<<str6;
```

Такий запис не введе пробіли



Для повноцінного ведення інформації з пробілами слід використовувати функцію getline(), яка витягує з вхідного потоку рядки:

```
//Введення рядка.
	cout << "Ввести рядок: ";
	getline(cin,str6);
	cout<<str6;
```



Зверніть увагу на те, як легко тепер працювати з рядками. Наприклад, за допомогою оператора "+" їх можна конкатеніровать, а оператор ">" дозволяє порівнювати рядки між собою. Для того щоб виконати ці операції в стилі мови С, довелося б викликати функції **strcat()** і **strcxnp().** Оскільки об'єкти класу **string** і звичайні рядки можна змішувати в одному і тому ж вираженні, програми стають набагато простіше.

Слід зазначити ще одну особливість попередньої програми: розмір рядка ніде не вказувався явно. Об'єкти класу string автоматично обчислюють свій розмір, необхідний для зберігання відповідного рядка. Таким чином, при привласненні або конкатенації рядків результуюча рядок автоматично збільшиться на необхідну величину. Це динамічна властивість об'єктів класу **string** є їх безперечною перевагою в порівнянні зі звичайними рядками, в котрих можливий вихід за межі допустимого діапазону.

Деякі функції - члени класу string

Хоча основні операції над рядками можна виконати за допомогою основних операторів, більш складні дії над рядками здійснюються функціями - членами класу **string**. Оскільки клас **string** містить занадто велику кількіть функцій-членів, ми розглянемо лише деякі з них.

Основні маніпулятори

Щоб привласнити один рядок другому, необхідно застосовувати функцію **assign()**. Вона має два варіанти.

string &assign (const string &strob, size_type start, size_type num);

string &assign (const string *str, size_type num);

Перша функція присвоює об'єкту *num* символи з рядка *strob* починаючи з індексу, заданого параметром *start*. Друга функція присвоює викликаному об'єкту *num* символів з С-рядка *str*, починаючи з індексу, заданого параметром **start**. У кожному разі повертається посилання на викликаючий об'єкт. Зрозуміло, для присвоєння повних рядків було б набагато легше застосовувати оператор "=". Функцію **assign** слід використовувати лише для присвоєння частини рядка.

За допомогою функції-члена **append ()** можна додати одному рядку частина іншого рядка. Ця функція має дві форми.

string &append (const string &strob, size_type start, size_type num);

string &append (const string *str, size_type num);

Перша функція додає до викликаного об'єкту *num* символів з рядка *strob*, починаючи з позиції, заданої параметром *start*. Друга функція приписує до викликаного об'єкту *num* символів з С-рядка str; починаючи з індексу, заданного параметром *start*. У кожному разі повертається посилання на зухвалий об'єкт. Зрозуміло, для конкатенації двох повних рядків було б набагато легше застосувати оператор "+". Функцію **append** слід використовувати лише для прі¬пісиванія часткової рядка.

Функції **insert()** і **replace()** призначені для вставки і заміни символів. Їх прототипи перераховані нижче.

string &insert (size_type start, const string &strob);

string &insert (size_type start, const string &strob,
size_type intStart, size_type num);

string &replace (size_type start, const string &strob);

string &replace (size_type start, const string &strob,
size_type replaceStart, size_type replaceNum);

Перша форма функції **insert ()** вставляє рядок *strob* в викликає об'єкт класу **string**, починаючи з індексу, заданого параметром *start*. Другий варіант функції **insert ()** вставляє *num* символів рядка *strob*, починаючи з позиції, заданої параметром *insStart*, в викликає об'єкт класу *string*, починаючи з індексу, заданого параметром *start*.

Перша форма функції **replace()** замінює *num* символів викликає об'єкта класу **string**, починаючи з позиції, заданої параметром starts рядком *strob*. Другий варіант функції **replace ()** замінює *orgNum* символів об'єкта класу, починаюси з позиції, заданої параметром starts *replaceNum* символами рядка strob, починаючи з індексу, заданого параметром replaceStart. В обох випадках повертається посилання на зухвалий об'єкт.

За допомогою функції **erase()** можна видалити символи з рядка. Один з її виглядає наступним чином.

string &erase (size_tyoe *start* = 0, *num* = npos);

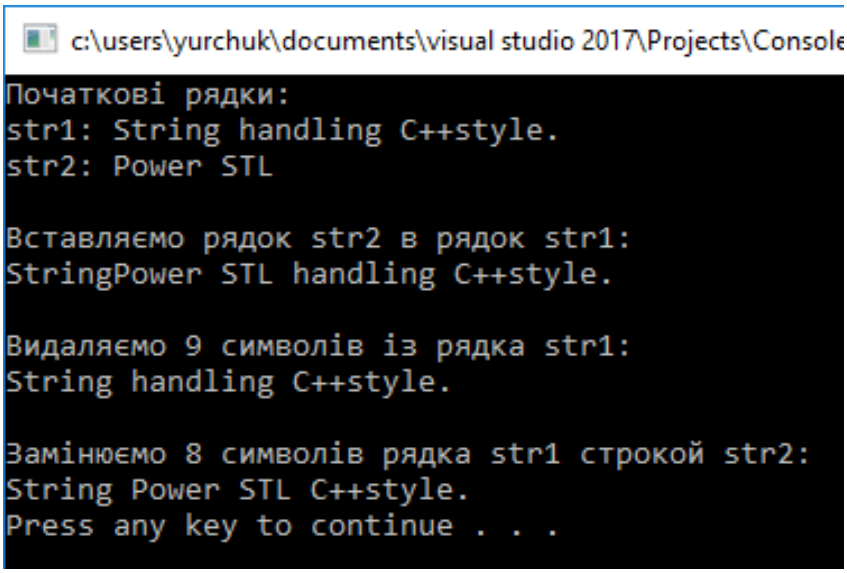
Ця функція видаляє *num* символів з викликає об'єкта, починаючи з позиції заданої параметром *start*.

Функції **insert (), erase ()** і **replace ()** ілюструються наступною програмою.

```
//Демонстрація функцій insert(), erase() і replace().
#include <iostream>
#include <string>
#include "windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	string str1("String handling C++style.");
	string str2("Power STL");

	cout<<"Початкові рядки:\n";
```

```
        cout << "str1: " << str1 << endl;
        cout << "str2: " << str2 << "\n\n";

        //Демонстрація функції insert().
        cout << "Вставляємо рядок str2 в рядок str1:\n";
        str1.insert(6, str2);
        cout << str1 << "\n\n";

        //Демонстрація функції erase().
        cout << "Видаляємо 9 символів із рядка str1:\n";
        str1.erase(6, 9);
        cout << str1 << "\n\n";

        //Демонстрація функції replace.


cout << "Замінюємо 8 символів рядка str1 строкой str2:\n";
str1.replace(7, 8, str2);
cout << str1 << endl;

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

Результати роботи програми наведемо нижче



Пошук символу в рядку

Клас **string** містить декілька функцій-членів, призначених для пошуку подстрок, зокрема функції **find ()** і **rfind()**. Їх прототипи приведені нижче.

size_type find(const string &strob, size_type start=0) const;

size_type rfind(const string &strob, size_type start=npos) const;

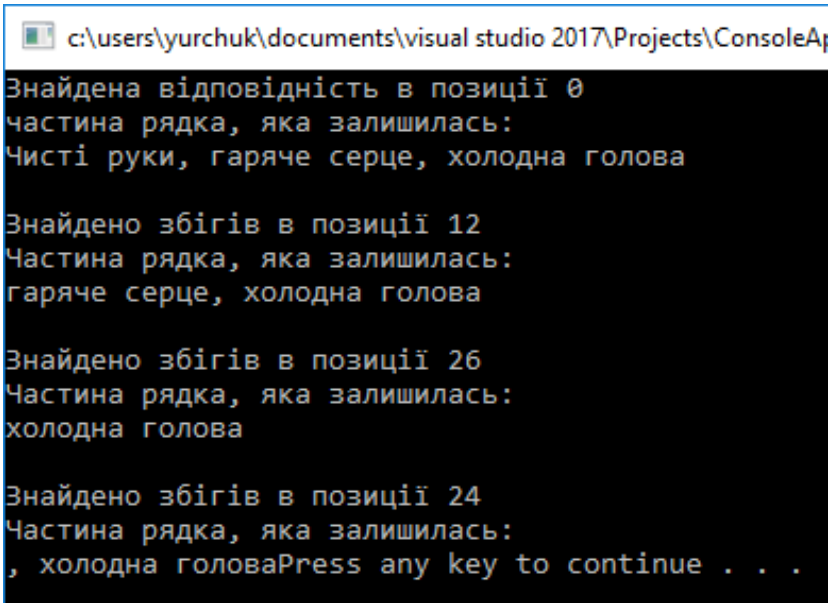
Починаючи з позиції, заданої параметром *start*, функція **find()** виконує пошук першого входження рядка, що міститься в об'єкті *strob*. Якщо підрядок знайдено, функція **find ()** повертає індекс її першого символу в шуканому рядку. Якщо підрядок не знайдено, повертається константа npos. Функція **rfind()** є протилежністю функції **find ().** Починаючи з позиції, заданої параметром *start,* функція **rfind()** виконує пошук останнього вхождення рядка, що міститься в об'єкті strob. Якщо підрядок знайдена, функція **rfind()** повертає індекс її останнього входження в досліджуваний об'єкт. Якщо підрядок не знайдено, повертається константа npos.

Розглянемо короткий приклад, в якому використовуються функції **find ()** і **rfind().**

```
#include <iostream>
#include "windows.h"
```

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

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	int i;
	string s1 =
		"Чисті руки, гаряче серце, холодна голова";
	string s2;

	i = s1.find("Чисті");
	if (i != string::npos) {
		cout << "Знайдена відповідність в позиції " << i << endl;
		cout << "частина рядка, яка залишилась: \n";
		s2.assign(s1, i, s1.size());
		cout << s2;
	}
	cout << "\n\n";

	i = s1.find("гаряче");
	if (i != string::npos) {
		cout << "Знайдено збігів в позиції " << i << endl;
		cout << "Частина рядка, яка залишилась:\n";
		s2.assign(s1, i, s1.size());
		cout << s2;
	}
	cout << "\n\n";

	i = s1.find("холодна");
	if (i != string::npos) {
		cout << "Знайдено збігів в позиції " << i << endl;
		cout << "Частина рядка, яка залишилась:\n";
		s2.assign(s1, i, s1.size());
		cout << s2;
	}
	cout << "\n\n";

	//Знаходимо останню кому.
	i = s1.rfind(",");
	if (i != string::npos) {
		cout << "Знайдено збігів в позиції " << i << endl;
		cout << "Частина рядка, яка залишилась:\n";
		s2.assign(s1, i, s1.size());
		cout << s2;
	}

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

Результат роботи програми показаний нижче.



Порівняння рядків

Для порівняння повних рядків зазвичай застосовуються перевантажені оператори порівняння, описані вище. Однак, якщо виникає необхідність порівняти частина одного рядка з іншого рядком, необхідно використовувати функцію-член compare ().

int compare(size_type start, size_type num, const string &strob) const;

Ця функція порівнює *num* символів, які належать об'єкту *strob*, з викликає рядком. Якщо викликає рядок коротше рядка *strob*, функція *compare ()* повертає від'ємне число. Якщо викликає рядок довший рядка strob, функція *compare ()* повертає позитивне число. Якщо довжини рядків дорівнюють, повертається нуль.

Створення С-рядка

Хоча об'єкти класу **string** корисні самі по собі, іноді виникає необходідність перетворити їх в С-рядка. Наприклад, об'єкт класу **string** можна затосувати для створення імені файлу. Однак при відкритті файлу необхідно задати вказівник на звичайний С-рядок. Для вирішення цієї проблеми призначена функція **c_str ().** Її прототип наведено нижче.

const char *c_str() const;

Ця функція повертає покажчик на С-рядок, що міститься в визваному об'єкті класу **string**. С-рядок не повинна змінюватися. Крім того, немає гарантії, що вона залишиться коректної після виконання операцій над об'єктом.

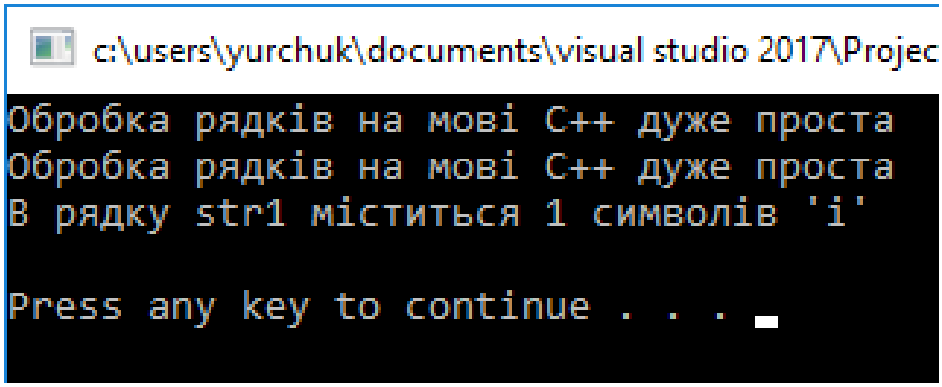
Рядки як контейнери

Клас **string** відповідає багатьом умовам, які пред'являються контейнерам. Це значить, що він підтримує стандартні контейнерні функції, такі як **begin (), end ()** і **size ()**. Отже, до об'єктів класу **string** можна застосовувати алгоритми з бібліотеки STL.

```
// Рядки як контейнери.
#include <iostream>
#include <string>
#include <algorithm>
#include "windows.h"
using namespace std;

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

```
	SetConsoleOutputCP(1251);
	string str1("Обробка рядків на мові C++ дуже проста");
	string::iterator p;
	int i;

	// Застосування функції size().
	for (i = 0; i<str1.size(); i++)
		cout << str1[i];
	cout << endl;

	// Застосування ітератора.
	p = str1.begin();
	while (p != str1.end())
		cout << *p++;
	cout << endl;
	//Застосування алгоритму count ().
	i = count(str1.begin(), str1.end(), 'i');
	cout << "В рядку str1 міститься " << i << " символів 'i'\n";


	cout << endl;

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

Результат роботи цієї програми показаний нижче.

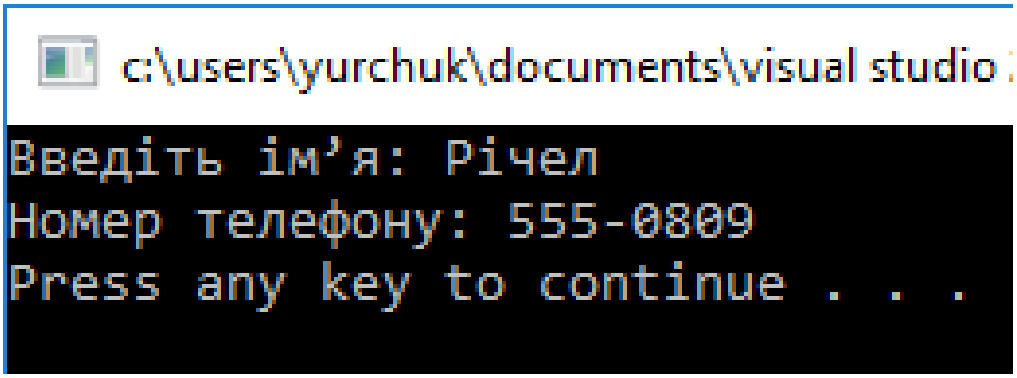


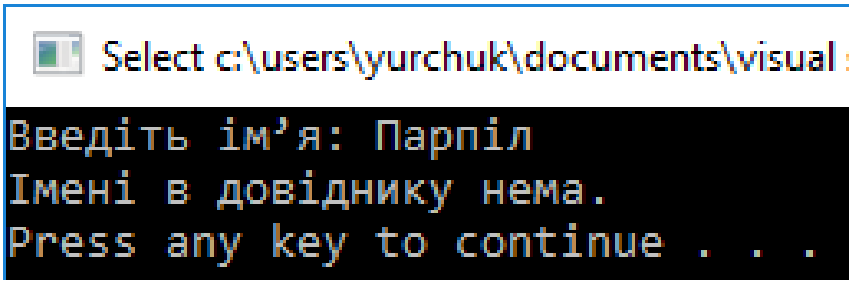
Запис рядка в інший контейнер

Хоча клас **string** є контейнерним, зазвичай рядка зберігаються в інших стандартних контейнерах, наприклад, в асоціативних масивах або списках. Так, програму, що імітує телефонну книжку, можна переписати інакше. Для зберігання імен та телефонних номерів вона використовує асоціативний масив, що містить об'єкти класу **string**, а не С-рядка.

```
#include <iostream>
#include <map>
#include "windows.h"
#include <string>
using namespace std;

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

	directory.insert(pair<string, string>("Том", "555-4533"));
	directory.insert(pair<string, string>("Крис", "555-9678"));
	directory.insert(pair<string, string>("Джон", "555-8195"));
	directory.insert(pair<string, string>("Річел", "555-0809"));
```

```
        string s;
        cout << "Введіть ім’я: ";
        cin >> s;

        map<string, string>::iterator p;

        p = directory.find(s);
        if (p != directory.end())
                cout << "Номер телефону: " << p->second<<endl;
        else

cout << "Імені в довіднику нема.\n";
        system("pause");
return 0;
}
```





🔷 `std::string` — основне

`std::string` — це клас з заголовку `<string>`, що реалізує динамічний масив символів з багатим набором методів.

```
#include <string>

std::string s = "Привіт, світе!";
```

🔷 Основні можливості

Категорія	Приклади методів
Конструктори	`string(),string("abc"),string(n, 'a')`
Доступ	`s[i],at(i),front(),back()`
Зміна	`append(),insert(),erase(),replace(),clear(),push_back(),pop_back()`
Пошук	`find(),rfind(),find_first_of()`
Перетворення	`substr(),c_str(),data()`
Інше	`size(),length(),empty(),resize(),capacity(),shrink_to_fit()`

🔷 Основні методи з прикладами

🔶 Створення рядка

```
std::string s1;                    // порожній рядок
std::string s2 = "hello";          // з літерала
std::string s3(5, 'A');            // "AAAAA"
```

🔶 Доступ до символів

```
s2[1] = 'a';        // змінюємо символ
char ch = s2.at(2); // з перевіркою меж
char f = s2.front();
char b = s2.back();
```

🔶 Додавання

```
s2 += " world";               // додавання
s2.append("!!!");             // ще один спосіб
s2.push_back('!');            // один символ
```

🔶 Видалення

```
s2.pop_back();                // видаляє останній символ
s2.erase(5, 3);               // з 5 позиції видалити 3 символи
s2.clear();                   // очистити повністю
```

🔶 Вставка / Заміна

```
std::string s = "привіт";
s.insert(6, " світ");          // → "привіт світ"
s.replace(0, 6, "добрий день"); // → "добрий день світ"
```

🔶 Пошук

```
std::string s = "де знайти символ?";
size_t pos = s.find("символ"); // індекс або string::npos
```

🔶 Витяг частини

```
std::string part = s.substr(3, 5); // 5 символів з індексу 3
```

🔹 Інші методи

```
s.length();          // довжина
s.size();            // теж саме
s.empty();           // чи порожній?
s.resize(100);       // змінити розмір (додасть нулі)
s.capacity();        // розмір виділеної пам'яті
s.shrink_to_fit();   // стиснути до реального розміру
```

🔹 Перетворення на `const char*`:

```
const char* cstr = s.c_str(); // для функцій типу printf
```

🔹 Операції порівняння

```
if (s1 == s2) ...
if (s1 < s2)  ...
```

🔹 Переваги `std::string`:

- Автоматично керує пам’яттю
- Багато корисних методів
- Легке об’єднання, вставка, пошук

🔹 Недоліки:

- Менш ефективний при дуже частих модифікаціях у середині рядка (у таких випадках — краще `std::ostringstream`, `std::vector<char>`)