Лекція 14. Асоціативні контейнери STL

1. *Трішки міркувань про реалізацію множини.*
2. *Список слів без повторень. Частота вживання сів у тексті.*
3. *Журнал спостережень за тваринами.*
4. *Операції з множинами.*
5. *Як працюють асоціативні контейнери – експеримент.*

Як програмно реалізувати структуру даних множина? В математиці множина – первісне поняття, що не підлягає визначенню, аксіома. За словами творця теорії множин Георга Кантора, множина – це збірка певних і різних об'єктів нашої інтуїції чи інтелекту, яку розглядають як ціле. Найсуттєвішим у математичному понятті множини є об'єднання *різних* об'єктів в *одне ціле*. Порядок елементів множини не має значення, тому їхнє взаємне розташування не фіксують. Невідомо, який елемент множини перший, а який – останній.

Програми більш приземлені, ніж математика. Вони оперують з реальними сутностями в оперативній пам'яті, а не з «об'єктами інтелекту», тому реалізація множини безумовно відрізнятиметься від математичного поняття. У програмних рішеннях перш за все беруть до уваги ефективність і надійність реалізації, зручність використання.

Якщо перелік можливих елементів множини відомий наперед, то зображенням множини може бути сукупність ознак того, чи присутній кожен елемент у множині чи ні. При цьому встановлюють взаємно однозначну відповідність між елементами і ознаками. Поговоримо, наприклад, про реалізацію *множини літер*. Всі можливі літери, значення типу *char*, відомі. Тип містить 256 різних значень. Присутність літери в множині можна позначити одним бітом: 1, якщо літера є, і 0 у протилежному випадку. Для зображення множини потрібно 256 біт, що складає 32 байти. Можна використати масив з восьми цілих (по чотири байти кожен). За такого підходу операції з множинами – це побітові операції зі значеннями типу *int*. У програмі одні з найефективніших. Всі множини мають однаковий розмір: порожня – це 256 бітів зі значеннями 0, повна – стільки ж одиниць. Відомо повну множину, тому можна визначити доповнення заданої множини. Саме так реалізовано тип *set of T* в мові програмування Object Pascal. Обмеження на тип *T*: він не може містити більше як 256 значень.

Проте підхід має суттєві недоліки: не для всіх типів можна вказати вичерпний перелік можливих значень; якщо такий перелік великий, то реалізація може потребувати невиправдано великих масивів прапорців; щоб перебрати елементи множини, доводиться перебирати значення типу або реконструювати їх за місцем прапорця в зображенні множини.

Розробники бібліотеки STL використали інший підхід: вони вирішили зберігати в множині самі об'єкти, а не ознаки їхньої присутності. Зрештою, множина – це об'єднання різних об'єктів в одне ціле, тобто, в контейнер. Як гарантувати, що об'єкти контейнера різні? При кожному додаванні елемента до множини доведеться перевіряти, чи там такий уже є. Пошук у невпорядкованій послідовності матиме складність $O(length)$, що може виявитися завеликим для множин значного обсягу. Пошук зі складністю $O(\log_2 length)$ можна реалізувати у впорядкованій послідовності, наприклад, бінарний пошук у відсортованому масиві. Це дуже ефективний пошук, але неможливо реалізувати швидку вставку в масив. Таким чином приходимо до ідеї використання збалансованого дерева пошуку для зберігання елементів множини: швидкий пошук, легка вставка, не на багато складніше вилучення.

Завдяки реалізації, множина STL отримала впорядкованість елементів. Математиці до впорядкованості байдуже, а для використання в програмах це – додаткова зручність, адже можна використовувати ітератори. Контейнер *set* оголошено як шаблон класу, який можна конкретизувати різними типами даних. Таким чином реалізація є універсальною. Для пошуку в дереві множина використовує оператор <. Якщо тип даних не має такого оператора, другим параметром шаблону множини можна вказати бінарний предикат, який виконуватиме порівняння елементів множини.

Реалізація на основі дерева пошуку виявилася такою продуктивною, що її використано ще для трьох інших контейнерів: «мультимножини» *multiset*, відображення *map* і «мультивідображення» *multimap*. Невідомо, кого спроектували першим: множину чи відображення, – але назву групі контейнерів (*асоціативні контейнери*) дало *відображення*. Як уже було сказано в лекції 9, асоціацією в програмуванні називають пару (*ключ*; *значення*), а колекцію асоціацій – словником, наприклад, у Smalltalk, чи Python. Словник у C++ це – *map*. Вершина дерева пошуку відображення містить ключ (для порівнянь) і значення. Розробники трактують множину як відображення, у якого ключ і значення збігаються.

Дуже часто відображення помилково називають асоціативним масивом, але чи пасує називати масивом дерево тільки тому, що для відображення визначено оператор [ ]? Питання швидше риторичне. Далі розглянемо можливості і приклади використання асоціативних контейнерів.

Синтаксис оголошення

```
set<T>, multiset<T> //#include <set>

template <class Key, class Traits=less<Key>,
      class Allocator=allocator<Key>>
class set;

template <class Key, class Compare =less <Key>,
      class Allocator =allocator <Key>>
class multiset;

map<K,T>, multimap<K,T> //#include <map>

template <class Key, class Type, class Traits = less<Key>,
      class Allocator=allocator<pair <const Key, Type>>>
class map;

template <class Key, class Type, class Traits=less <Key>,
      class Allocator=allocator <pair <const Key, Type>>>
class multimap;
```

Шаблони множини і мультимножини однакові. Другим параметром можна задати власний компаратор, якщо з якихось причин оператор < не влаштовує програміста, або не доступний для типу *T*. Шаблони відображення та мультвідображення також однакові. Тут два вільних типи даних: *Key* – тип ключа, *Type* – тип значення. Третій параметр шаблона задає компаратор ключів. Зверніть увагу на те, що пам'ять у відображенні виділяється для пари ключ-значення.

Спільні риси

	Визначення типів
key_type	A comparable type of the key
value_type	== key_type || == pair<const key_type, mapped_type>
mapped_type	A type that represents the data type stored in a map. *(map & multimap)*
	Ітератори
begin	Returns an iterator that addresses the first element in the set.
end	Returns an iterator that addresses the location succeeding the last element in a set.
rbegin	Returns an iterator addressing the first element in a reversed set.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed set.
	Константні ітератори
cbegin	Returns a const iterator that addresses the first element in the set.

cend	Returns a const iterator that addresses the location succeeding the last element in a set.
crbegin	Returns a const iterator addressing the first element in a reversed set.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed set.
	Загальні (як для контейнера)
get_allocator	Returns a copy of the allocator object used to construct the set.
swap	Exchanges the elements of two sets.
max_size	Returns the maximum length of the set.
	Спеціальні
	вставляння
insert	Inserts an element or a range of elements into a set.
emplace	Inserts an element constructed in place into a set.
emplace_hint	Inserts an element constructed in place into a set, with a placement hint.
	пошук
count	Returns the number of elements in a set whose key matches a parameter-specified key.
find	Returns an iterator addressing the location of an element in a set that has a key equivalent to a specified key.
equal_range	Returns a pair of iterators respectively to the first element in a set with a key that is greater than a specified key and to the first element in the set with a key that is equal to or greater than the key.
lower_bound	Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.
upper_bound	Returns an iterator to the first element in a set with a key that is greater than a specified key.
	вилучення
erase	Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.
clear	Erases all the elements of a set.
	перевірки
empty	Tests if a set is empty.
size	Returns the number of elements in the set.
	специфічні
key_comp	Retrieves a copy of the comparison object used to order keys in a set.
value_comp	Retrieves a copy of the comparison object used to order element values in a set.
	Доступ до елемента відображення
at	Finds an element with a specified key value. (*map only*)
operator[]	Inserts an element into a map with a specified key value. (*map only*)

Список слів

Задача. Створити список слів без повторень, що входять до текстового файла. Словом вважати послідовність відмінних від пропуску літер. Слова відокремлено одним або кількома пропусками чи кінцем рядка.

Перше, що спадає на гадку – використати список. Дописування до кінця списку відбувається за константний час. Він має досить ефективні методи впорядкування, та вилучення повторних входжень. Можна спробувати.

```
// файлові потоки для заданого файла і результатів
ifstream fin("OldmanAndSea.txt"); // заданий
ofstream fout("result.txt");      // отриманий
// доступ до фалів буде через ітератори
istream_iterator <string> begin(fin), end;
```

```
ostream_iterator <string> out(fout,"\n");

// список для накопичення слів
list <string> L;
// всю роботу виконає алгоритм, ітератор потоку читання, і оператор введення з потоку
// саме оператор введення "вміє" виокремлювати групи літер, відокремлених пропусками
// нові слова дописуємо в кінець списку
copy(begin, end, back_inserter(L));
// вилучення повторних входжень слів - двоетапне
L.sort();
L.unique();
// тепер можна зберегти отриманий результат
*out++ = "----------------------------- list after sort & unique";
// всю роботу знову виконує алгоритм
copy(L.begin(),L.end(),out);
// загальна кількість знайдених слів
cout << "There are " << L.size() << " words\n";
```

А чи не розумніше було б для вилучення повторних входжень використати множину? Це доволі поширений прийом у програмуванні. Спробуємо й ми. Для взаємодії з потоками використаємо вже оголошені об'єкти.

```
// повернемо потік до початкового стану
fin.clear(); fin.seekg(0);
// множина для накопичення слів
set <string> S;
// вставляння в множину завжди починають з початку
copy(begin, end, inserter(S, S.begin()));
// все готово - можна зберігати результат
*out++ = "----------------------------- set after copy";
copy(S.begin(), S.end(), out);
// загальна кількість знайдених слів
cout << "There are " << S.size() << " words\n";
```

Програма з множиною виявилася компактнішою: один *copy* замість *copy, sort, unique* для списку. Пропонуємо читачам самостійно перевірити, який фрагмент коду працює швидше. Різниця в часі буде тим помітнішою, чим більший розмір заданого файла.

Загальна кількість знайдених слів обома способами однакова, і перелік слів – теж. Але з'ясовується, що оператор читання рядків не дуже добре дає собі раду з розпізнаванням слів: розділові знаки він сприймає як частину слова, і «слово» й «слово,» програма вважає різними, а це не так. Щоб виправити цю помилку, доведеться замість копіювання послідовності слів з файла загальним алгоритмом *copy* використати щось більш інтелектуальне. До речі, матимемо нагоду доречно використати алгоритм *transform* і потік читання з рядка.

```
// повертаємо файл і множину до початкового стану
S.clear();
fin.clear(); fin.seekg(0);
// знаки, що підлягають вилученню помістимо в множину
set<char> punct = { '.', ',', '!', '?', ':', ';', '-', '(', ')', '\"', '[', ']' };
string line;
// читатимемо з файла по одному рядку
while (getline(fin, line))
{
	// вилучення розділових знаків (нелітерних символів)
	transform(line.begin(), line.end(), line.begin(),
		[&punct](char c){ return punct.find(c) == punct.end() ? c : ' '; });
	// виокремлення слів і додавання до множини
	istringstream is(line);
	string word;
	while (is >> word) S.insert(word);
}
// все готово - можна зберігати результат
*out++ = "----------------------------- set after input & wiping";
```

```
        copy(S.begin(), S.end(), out);
        // загальна кількість знайдених слів
        cout << "There are " << S.size() << " words\n";
        // тепер файли можна закрити
        fin.close();
        fout.close();
```

Тут алгоритм *transform* витирає з прочитаного рядка всі символи, що належать множині *punct* (заміняє їх пропусками). Таке перетворення не змінює довжину рядка, тому все працює без помилок. Лямбда-вираз, переданий алгоритмові, захоплює посилання на множину *punct*. Вона потрібна, щоб розпізнати «зайві» символи, а посилання використано, щоб не створювати її копію. Тепер оператор читання (з рядка) коректно розпізнає слова, після чого додаємо їх до множини. Загальна кількість слів очікувано зменшилася.

[Повний текст програм у проекті ListVSset]

Чи зауважили ви, які можливості множин ми вже використали:

- *set <string> S* – створення порожньої множини;
- *set<char> punct = { '.', ',', '!', '?', ':', ';', '-', '(', ')', '\"', '[', ']' }* – початкове значення створеної множини задано списком-ініціалізатором;
- *S.clear()* – вилучення всіх елементів множини;
- *S.begin(), S.end()* – використання ітераторів;
- *punct.find(c) != punct.end()* – перевірка, чи належить елемент множині;
- *S.insert(word)* – додавання елемента до множини;
- *S.size()* – визначення кількості елементів множини

У наступних прикладах скористаємося також і іншими можливостями.

Підрахунок слів

Задача. Обчислити, скільки разів зустрічається в тексті програми кожне слово.

Схоже, ця програма є ускладненим варіантом попередньої, адже нам тут теж потрібно мати перелік слів без повторень і, додатково, кількості кожного з них. Розв'язок задачі допоможе проілюструвати використання мультимножини і відображення.

Пригадаємо, що таке «мультимножина». Судячи з назви, це – сукупність об'єктів, що може містити повтори (однакові об'єкти). Як уже було сказано, *multiset* реалізовано на основі дерева пошуку, тому контейнер впорядковує свої елементи. Завдяки наявності ітераторів користувач може трактувати *multiset* як впорядковану послідовність, але без оператора індексування. У впорядкованій послідовності однакові слова стоятимуть поруч. Щоб довідатися, скільки разів слово зустрічається в тексті, можемо визначити довжину проміжку однакових слів.

```
char replaceNotAlpha(char c)
{
        return c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' || c == '_' ? c : ' ';
}
/* Обчислення кількості слів у файлі за допомогою мультимножини
   Зверніть увагу на те, як обчислюється кількість однакових елементів*/
void MultisetWord()
{
        cout << "\n*** Count words in the program with help of multiset\n\n";
        ifstream in ("MultiSetWord.cpp");
        multiset<string> wordmset;
        string line;
        while (getline(in, line)) // завантажуємо по одному рядку
        {                         // залишаємо в рядках лише латинські літери
                transform(line.begin(), line.end(), line.begin(), replaceNotAlpha);
                istringstream is(line);
                string word;   // читаємо слова і додаємо їх до контейнера
                while (is >> word) wordmset.insert(word);
```

```
	}
	// отриману мультимножину переберемо за допомогою ітераторів
	typedef multiset<string>::iterator MSIter;
	MSIter it = wordmset.begin();
	while (it != wordmset.end())
	{
		// метод equal_range повертає пару ітераторів [початок; кінець)
		pair<MSIter, MSIter> p = wordmset.equal_range(*it);
		// шаблонна функція для обчислення довжини інтервалу
		int count = distance(p.first, p.second);
		cout << *it << "  :  " << count << '\n';
		it = p.second; // перехід до наступного слова
	}
}
```

Ту саму роботу може виконати відображення. Ключами відображення будуть знайдені слова, а значеннями – кількості входжень до тексту.

```
// оператор виведення пари потрібно включити до простору std – до того простору,
// де визначено шаблон pair
namespace std
{
	template<typename F, typename S>
	ostream& operator<<(ostream& os, const pair<F, S>& p)
	{
		return os << p.first << " : " << p.second;
	}
}

void WordCount()
{
	cout << "\n*** Count words in the program\n\n";
	ifstream in("MultiSetWord.cpp");
	// відображення спочатку порожня, не містить жодного ключа
	map<string, int> wordmap;
	string line;
	while (getline(in, line))
	{
		transform(line.begin(), line.end(), line.begin(), replaceNotAlpha);
		istringstream is(line);
		string word;
		// оператор [] знаходить значення за ключем або
		// автоматично створює пару (ключ, значення за замовчуванням)
		while (is >> word) ++wordmap[word];
	}
	// словник можна вивести стандартним алгоритмом, якщо
	// перевантажено оператор виведення для pair
	copy(wordmap.begin(), wordmap.end(), ostream_iterator<pair<string, int>>(cout, "\n"));
	cout << '\n';
	system("pause");
	return;
}
```

Треба звернути увагу на декілька важливих моментів. Додавання нових значень до *map* можна записати дуже компактно, як в рядку програми, виділеному товстим. Оператор [ ] викликає метод *insert* відображення, тому нові ключі автоматично додаються до контейнера. Для відповідного значення встановлюється значення за замовчуванням. У нашій програмі лічильник отримає значення 0. Після чого відразу буде збільшений оператором ++. Якщо відображення вже містить шуканий ключ, то оператор [ ] поверне посилання на знайдений лічильник, і він, знову ж таки, збільшиться.

Наступне: ітератор відображення вказує на пару (*ключ*; *значення*). Зверніть увагу на тип, вказаний в конструкторі ітератора потоку виведення. До речі, щоб алгоритм *copy* зміг вивести вміст відображення в потік, потрібно, щоб в просторі *std* було визначено оператор

виведення для *pair*. На жаль, його там нема. На щастя, простори імен можна розширювати, що ми і зробили на початку наведеного фрагмента програми.

[Повний текст програм у проекті WordsCounts]

Журнал спостережень

Задача. Натураліст веде спостереження за тваринами в дикій природі і фіксує в журналі вид тварини, яку він зустрів, місце і час спостереження. Запропонуйте структуру даних, що моделюватиме такий журнал і наведіть приклади його використання.

Очевидно, що вид тварини і місце спостереження пов'язані між собою. Що обрати в якості первинного значення: вид, чи місце? Відповідь залежить від потреб дослідника: чи його цікавить міграція тварин певного виду, чи, навпаки, які саме тварини з'являються в певному місці. Дослідник спостерігає за обмеженим числом видів, а часові рамки і місце спостережень практично не обмежені, тому первинним вважатимемо вид. За таких умов одному видові відповідатиме декілька місць спостережень.

Для фіксації спостережень можна використати відображення *map*, але тоді одному ключу має відповідати колекція місць. Якщо обрати відображення *multimap*, то контейнер зможе містити декілька записів з однаковими ключами (видами тварин) і різними значеннями (місцями спостережень). Для зручності місце і час спостереження опишемо окремим класом. Щоб продемонструвати роботу такого журналу зімітуємо роботу дослідника: використаємо генератор спостережень для наповнення журналу записами.

```
// Опис спостереження поєднує координати і час
class DataPoint
{      int x, y;               // координати
       time_t time;            // час
public:
       DataPoint() : x(0), y(0), time(0) {}
       DataPoint(int xx, int yy, time_t tm) : x(xx), y(yy), time(tm) {}
       int getX() const { return x; }
       int getY() const { return y; }
       const time_t* getTime() const { return &time; }
};
// джерело назв видів тварин
vector<string> animals{ "chipmunk", "beaver", "marmot", "weasel", "squirrel", "bear",
       "ptarmigan", "eagle", "hawk", "vole", "deer", "otter", "hummingbird", "elk" };
    /* "бурундук", "бобер", "бабак", "ласка", "білка", "ведмідь",
       "рябчик", "орел", "яструб", "полівка", "олень", "видра", "колібрі", "лось"*/

// запис в журналі
typedef pair<string, DataPoint> Sighting;

ostream& operator<<(ostream& os, const Sighting& s)
{
       char time[30]; ctime_s(time, sizeof time, s.second.getTime());
       return os << s.first << " sighted at x=" << s.second.getX() << ", y="
              << s.second.getY() << ", time=" << time;
}
// генератор для автоматичного наповнення журналу записами
class SightingGen
{
       vector<string>& animals;
       enum { D = 100 };
public:
       SightingGen(vector<string>& an): animals(an) {}
       Sighting operator()()
       {
              Sighting result;
              // вид тварини вибираємо з переліку випадково
              int select = rand() % animals.size();
              result.first = animals[select];
              // координати випадкові, час поточний
```

```
            result.second = DataPoint(rand() % D, rand() % D, time(0));
            return result;
        }
};
// текстове меню для вибору виду тварини
int menu()
{       cout << "\nSELECT an animal or '-1' to quit:\n";
        for (size_t i = 0; i < animals.size(); i++)
        {       cout << '[' << i << ']' << animals[i] << "     ";
                if ((i + 1) % 5 == 0) cout << '\n';
        }
        cout << endl;
        int i; cin >> i;
        if (i > 0) i %= animals.size();
        return i;
}
// тут усе відбувається
int main()
{       typedef multimap<string, DataPoint> DataMap;
        typedef DataMap::iterator DMIter;
        DataMap sightings; // журнал спостережень
        srand(time(0));
        // записуємо в журнал 50 спостережень
        generate_n(inserter(sightings, sightings.begin()), 50, SightingGen(animals));
        // весь журнал - на екран
        cout << "\n\nSEE AT ALL SIGHTINGS HERE: \n";
        copy(sightings.begin(), sightings.end(), ostream_iterator<Sighting>(cout, ""));
        cout << "----------------------------\n";
        // отримання інформації про вибрану тварину
        for (int count = 0; count < 10; ++count)
        {       int i = menu();
                if (i < 0) break;
                pair<DMIter, DMIter> range = sightings.equal_range(animals[i]);
                copy(range.first, range.second, ostream_iterator<Sighting>(cout, ""));
        }
        return 0;
}
```

Екземпляр генератора містить посилання на вектор з назвами видів тварин. Саме звідти він випадково вибирає назву для чергового спостереження. Результатом роботи оператора () є *pair<string, DataPoint>*. Саме ця пара потрапляє аргументом у метод *insert* відображення *sightings*. При виведенні журналу спостережень на екран, як і в попередній програмі, працює оператор виведення пари. Як і раніше, метод *equal_range* повертає пару ітераторів: на діапазон спостережень за твариною одного виду.

[Повний текст програм у проекті WildLifeMonitor]

Операції з множинами

Чи належить елемент *x* множині *S* можна перевірити двома способами: *if (S.count(x) > 0)* або *if (S.find(x) != S.end())*. Тут працюють методи контейнера. А от звичні математикові операції об'єднання, перетину, різниці множин виконують за допомогою узагальнених алгоритмів.

```
int main()
{
        int arr[] = { 2, 0, 9, 8, 3, 5, 6, 1, 4, 7 };

        // Множини спеціально збудовано з частин масиву
        set<int> A(arr, arr + 7);
        set<int> B(arr + 4, arr + 10);
        ostream_iterator<int> out(cout, " ");

        // Вигляд вихідних множин
        cout << " A = { ";
        copy(A.begin(), A.end(), out); cout << "}\n";
```

```
	cout << " B = { ";
	copy(B.begin(), B.end(), out); cout << "}\n";

	// Чи належить arr[0] множинам?
	cout << arr[0] << " occures in A " << A.count(arr[0]) << " times.\n";
	if (B.find(arr[0]) != B.end()) cout << arr[0] << " belongs to B.\n";
	else cout << arr[0] << " not belongs to B.\n";

	// Виконання дій з множинами за допомогою алгоритмів
	set<int> C;   // перетин
	set_intersection(A.begin(), A.end(), B.begin(), B.end(), inserter(C, C.begin()));
	cout << "\n A * B = { ";
	copy(C.begin(), C.end(), out); cout << "}\n";

	set<int> D;   // об'єднання
	set_union(A.begin(), A.end(), B.begin(), B.end(), inserter(D, D.begin()));
	cout << " A + B = { ";
	copy(D.begin(), D.end(), out); cout << "}\n";

	set<int> E;   // різниця
	set_difference(A.begin(), A.end(), B.begin(), B.end(), inserter(E, E.begin()));
	cout << " A - B = { ";
	copy(E.begin(), E.end(), out); cout << "}\n";

	return 0;
}
```

Не надто зручно. Хто програмував на Object Pascal, той сумуватиме за операторами, визначеними для множин: + – об’єднання, * – перетин тощо. Проте такий підхід надзвичайно гнучкий. Можна знаходити перетин і об’єднання довільних контейнерів, що мають властивості множини – унікальність і впорядкованість елементів.

```
	// операції з множинами можна виконувати не тільки з множинами
	multiset<int> M(arr, arr + 7); // це множина, бо в масиві всі елементи різні
	list<int> L(arr + 4, arr + 10);
	L.sort(); L.unique();          // тепер це також множина

	cout << "\nInersection\n";
	set_intersection(M.begin(), M.end(), L.begin(), L.end(), out);

	cout << "\nUnion\n";
	set_union(M.begin(), M.end(), L.begin(), L.end(), out);

	cout << "\nDifference\n";
	set_difference(M.begin(), M.end(), L.begin(), L.end(), out);
	cout << '\n';
```

[Повний текст програм у проекті SetOperations]

Використання другого параметра шаблона функції продемонструємо за допомогою старих знайомих – ієрархії плоских геометричних фігур, що оголошує класи прямокутник, квадрат, трикутник, круг.

```
// об'єкт-функція для порівняння екземплярів-фігур
struct shapeComparer
{
	bool operator()(const Shape* a, const Shape* b) const
	{
		return a->square() < b->square();
	}
};
```

```
int main()
{
	typedef set<Shape*>::iterator Iterator;

	// Джерело даних для побудови-випробування множини
	Shape* shapes[] = { new Circle(), new Square(), new Rectangle(), new Triangle(),
		new Rectangle(2., 1.), new Circle(), new Triangle(4., 3., 90),
		new Triangle(3., 3., 60) };
	const int n = sizeof shapes / sizeof *shapes;


	// Нетривіальне оголошення множини
	set<Shape*, shapeComparer> shapeSet;

	for (int i = 0; i < n; ++i)
	{
		// метод вставляння повертає пару: місце вставки і ознаку успіху
		// якщо вставка була успішною, то res.first вказує на вставлений об'єкт
		pair<Iterator, bool> res = shapeSet.insert(shapes[i]);
		cout << *shapes[i]
			<< (res.second ? " was inserted successfully\n" : " was NOT inserted\n");
	}
	cout << "\n  ***  The set contains:\n";

	// Традиційний перебір контейнера
	Iterator it = shapeSet.begin();
	while (it != shapeSet.end())
	{	cout << **it << '\n';
		++it;
	}
	return 0;
}
```

У такій множині однаковими вважаються рівновеликі фігури.

[Повний текст програм у проекті SetOfShapes]

Як працюють асоціативні контейнери

Ми вже досліджували функціонування послідовних контейнерів за допомогою екземплярів класу *Noisy*. Використаємо їх і цього разу, щоб побачити виклики конструкторів і деструкторів при додаванні елементів до множини і до відображення.

```
int main()
{
	const int size = 7;
	cout << "*** The array creation\n";
	Noisy na[size];

	// заповнення множини
	cout << "*** The set creation\n";
	set<Noisy> ns(na, na + size);
	cout << endl;
	Noisy n;

	// додавання нового елемента множини
	cout << "\n*** Insertion\n";
	ns.insert(n);
	ns.insert(Noisy());
	cout << endl;
	cout << "\n*** Looking for\n";
	cout << "ns.count(" << n << ")=" << ns.count(n) << endl;

	// перевірка наявності в множині
	if (ns.find(n) != ns.end()) cout << "n(" << n << ") found in ns\n";
```

```
        // конструювання на місці
        cout << "\n*** Emplacing\n";
        ns.emplace(22);
        ns.emplace(7, 5);

        // вилучення
        cout << "\n*** Erasing\n";
        ns.erase(22);
        ns.erase(ns.find(n));

        // виведення
        cout << "\n *** The set\n";
        copy(ns.begin(), ns.end(), ostream_iterator<Noisy>(cout, "\n"));
        cout << "\n\n--------------------------------\n";

        // автоматичне заповнення відображення
        cout << "\n*** Fun with a map<int,Noisy>\n";
        map<int, Noisy> nm;
        cout << "\n-- Automatic filling --------------------\n";
        for (int i = 0; i < 10; ++i) nm[i];
        cout << "\n-- Output the map ---------------------\n";
        for (size_t j = 0; j < nm.size(); ++j) cout << "nm[" << j << "] = " << nm[j] << '\n';
        cout << "\n-- Insertion by [] ----------------------\n";

        // додавання нової пари до відображення
        nm[10] = n;
        cout << "\n-- Insertion ----------------------------\n";

        // вставляння нової пари
        nm.insert(make_pair(47, n));
        cout << "\n-- Emplacing -------------------------\n";
        nm.emplace(15, Noisy());
        nm.emplace(20, 25);
        nm.emplace(21, Noisy(3, 3));
        cout << "\n-- Replacing -------------------------\n";
        nm[47] = Noisy();
        nm[5] = nm[0];
        cout << "\n-- Looking for a key ------------------\n";

        // пошук за ключем
        cout << "\n nm.count(10) = " << nm.count(10)
             << "\n nm.count(11) = " << nm.count(11) << '\n';
        map<int, Noisy>::iterator it = nm.find(6);
        if (it != nm.end())
             cout << "value:" << (*it).second << " found in nm at location 6\n";

        // ітератор відображення перебирають пари ключ-значення
        cout << "\n-- Output the map ---------------------\n";
        for (it = nm.begin(); it != nm.end(); ++it)
             cout << (*it).first << ':' << (*it).second << ".\n";
        cout << "\n--------------------------------\n";

        // Як працює оператор []
        map<Noisy, Noisy> mnn;
        Noisy n1, n2;
        cout << "\n================================\n";
        mnn[n1] = n2;
        cout << "\n................................\n";
        mnn[n2] = n1;
        cout << "\n--------------------------------\n";
        cout << mnn[n1] << '\t' << mnn[n2] << endl;
        cout << "\n-------- Clean up -------------\n";
        system("pause");
        return 0;
}
```

[Повний текст програм у проекті NoisyExperiment]

Цю програму варто пов'язати з її виведенням. Постарайтеся розтлумачити собі кожен рядок виведення.

```
*** The array creation
default: [ 0 ]
default: [ 1 ]
default: [ 2 ]
default: [ 3 ]
default: [ 4 ]
default: [ 5 ]
default: [ 6 ]
*** The set creation
   copy: [ 0 ]
   copy: [ 1 ]
   copy: [ 2 ]
   copy: [ 3 ]
   copy: [ 4 ]
   copy: [ 5 ]
   copy: [ 6 ]

default: [ 7 ]

*** Insertion
   copy: [ 7 ]
default: [ 8 ]
   copy: [ 8 ]
destruct~[ 8 ]


*** Looking for
ns.count(Noisy_7)=1
n(Noisy_7) found in ns

*** Emplacing
one par: [ 22 ]
two par: [ 705 ]

*** Erasing
one par: [ 22 ]
destruct~[ 22 ]
destruct~[ 22 ]
destruct~[ 7 ]

 *** The set
Noisy_0
Noisy_1
Noisy_2
Noisy_3
Noisy_4
Noisy_5
Noisy_6
Noisy_8
Noisy_705


--------------------------------

*** Fun with a map<int,Noisy>

-- Automatic filling --------------------
default: [ 12 ]
default: [ 13 ]
default: [ 14 ]
default: [ 15 ]
default: [ 16 ]
default: [ 17 ]
default: [ 18 ]
default: [ 19 ]
default: [ 20 ]
default: [ 21 ]

-- Output the map ---------------------
nm[0] = Noisy_12
nm[1] = Noisy_13
nm[2] = Noisy_14
nm[3] = Noisy_15
nm[4] = Noisy_16
nm[5] = Noisy_17
nm[6] = Noisy_18
nm[7] = Noisy_19
nm[8] = Noisy_20
nm[9] = Noisy_21

-- Insertion by [] ----------------------
default: [ 22 ]
 assign: ( 22 ) = [ 7 ]

-- Insertion ----------------------------
   copy: [ 7 ]
   copy: [ 7 ]
destruct~[ 7 ]

-- Emplacing ----------------------------
default: [ 23 ]
   copy: [ 23 ]
destruct~[ 23 ]
one par: [ 25 ]
two par: [ 303 ]
   copy: [ 303 ]
destruct~[ 303 ]

-- Replacing ----------------------------
default: [ 26 ]
 assign: ( 7 ) = [ 26 ]
destruct~[ 26 ]
 assign: ( 17 ) = [ 12 ]

-- Looking for a key ------------------

 nm.count(10) = 1
 nm.count(11) = 0
value:Noisy_18 found in nm at location 6

-- Output the map ---------------------
0:Noisy_12.
1:Noisy_13.
2:Noisy_14.
3:Noisy_15.
4:Noisy_16.
5:Noisy_12.
6:Noisy_18.
```

```
7:Noisy_19.
8:Noisy_20.
9:Noisy_21.
10:Noisy_7.
15:Noisy_23.
20:Noisy_25.
21:Noisy_303.
47:Noisy_26.

--------------------------------
default: [ 27 ]
default: [ 28 ]

================================
   copy: [ 27 ]
default: [ 29 ]
 assign: ( 29 ) = [ 28 ]

................................
   copy: [ 28 ]
default: [ 30 ]
 assign: ( 30 ) = [ 27 ]

--------------------------------
Noisy_28        Noisy_27

-------- Clean up --------------
Для продолжения нажмите любую клавишу . . .
destruct~[ 28 ]
destruct~[ 27 ]
destruct~[ 27 ]
destruct~[ 28 ]
destruct~[ 28 ]
destruct~[ 27 ]
destruct~[ 26 ]
destruct~[ 303 ]
```

```
destruct~[ 25 ]
destruct~[ 23 ]
destruct~[ 7 ]
destruct~[ 21 ]
destruct~[ 20 ]
destruct~[ 19 ]
destruct~[ 18 ]
destruct~[ 12 ]
destruct~[ 16 ]
destruct~[ 15 ]
destruct~[ 14 ]
destruct~[ 13 ]
destruct~[ 12 ]
destruct~[ 7 ]
destruct~[ 705 ]
destruct~[ 8 ]
destruct~[ 6 ]
destruct~[ 5 ]
destruct~[ 4 ]
destruct~[ 3 ]
destruct~[ 2 ]
destruct~[ 1 ]
destruct~[ 0 ]
destruct~[ 6 ]
destruct~[ 5 ]
destruct~[ 4 ]
destruct~[ 3 ]
destruct~[ 2 ]
destruct~[ 1 ]
destruct~[ 0 ]

-----------------------------------------
  Noisy creations: 31
Copy-Constructions: 15
Assigniments: 5
Destructions: 46
```