Лекція 4. Агрегація об’єктів

1. *Контейнерний клас. Модель пам’яті для зберігання послідовності.*
2. *Конструктори контейнера. Явний конструктор. Деструктор.*
3. *Проблема глибокого копіювання та її вирішення.*
4. *Оператор індексування. Винятки.*
5. *Перебір елементів з конкретною метою, розсилання повідомлень.*
6. *Узагальнений перебір. Функціональні типи.*

Контейнером називають екземпляр класу, призначений для зберігання колекції об’єктів. Він *агрегує* декілька однотипних об’єктів у одну сутність і надає новий інтерфейс: з колекцією взаємодіють не так, як з окремим об’єктом. Розглянемо процес його побудови на прикладі класу, що зберігає колекцію об’ємних фігур.

> *Завдання.* Оголосіть клас-контейнер, що зберігає послідовність змінної довжини об’ємних фігур. Клас повинен за потреби перерозподіляти пам’ять – збільшувати її обсяг, визначати весь набір конструкторів, деструктор, оператор присвоєння. Функціональність контейнера: додавання елементів у кінець, вставляння всередину, вилучення за номером, вилучення всіх, доступ за номером, перебір елементів: друк кожного, зберігання кожного, виконання довільної дії з кожним, пошук першого, що задовольняє певний критерій.

Контейнер – власник своїх елементів, керує їхнім життєвим циклом: знищує перед своїм знищенням, копіює під час свого копіювання чи присвоєння. Поліморфний контейнер містить сукупність вказівників на базовий клас. Пригадуємо, що у C++ вказівник на базовий клас сумісний з вказівником на будь-який підклас, тому з його допомогою і створюють колекції різнотипних екземплярів.

Для зберігання послідовності значень можна використовувати векторну або зв’язну пам’ять. З обома ми вже знайомі з попереднього семестру. Векторна пам’ять виділяється неперервною ділянкою, має фіксований розмір та забезпечує прямий доступ до своїх елементів за номером (індексом). Вона ефективна для читання-запису за індексом, перебору. У мові C++ векторну пам’ять використовують статичні та динамічні масиви. Зв’язна пам’ять займає окремі ділянки фізичної пам’яті, пов’язані взаємними посиланнями. Вона ефективна при додаванні та вилученні ланок у довільних місцях послідовності, проте надає повільніший послідовний доступ до елементів, геть незручна для індексування. Зв’язні структури мовою C++ моделюють за допомогою структур чи класів, що містять вказівники на сусідні ланки, як, наприклад, у лінійних одно- та двозв’язних списках.

Про влаштування списків ми вже говорили, а от про влаштування «масиву змінного розміру» – ще ні. Цим і займемося. Як основу для проектованого контейнера використаємо динамічний масив вказівників на об’єкти. Масив не може змінювати свого розміру, то як же контейнер зможе додавати нові елементи? Зазвичай при створенні контейнера пам’ять виділяють з невеликим запасом, а тоді поміщають додані елементи на вільні місця. Коли запас вичерпається, виділяють нову ділянку пам’яті більшого розміру, копіюють в неї наявні значення, звільняють стару пам’ять і продовжують роботу з новою, більшою, як схематично зображено на рис. 1. Тут пунктиром зображено попередню ділянку пам’яті контейнера та вказівники на його елементами, синіми стрілками – копіювання вказівників у нову пам’ять; зайняту пам’ять заштриховано.

Функцію розширення пам’яті можна застосовувати до будь-якого динамічного масиву, але її алгоритм не прикрасить основну програму. Знайдеться мало охочих програмувати таке щоразу в своїх проектах. Але цього і не потрібно, адже одне із завдань класу – приховати від користувача складність влаштування і надати зручний інтерфейс для використання. Отож, методи обслуговування пам’яті контейнера ми оголосимо в закритій частині класу.

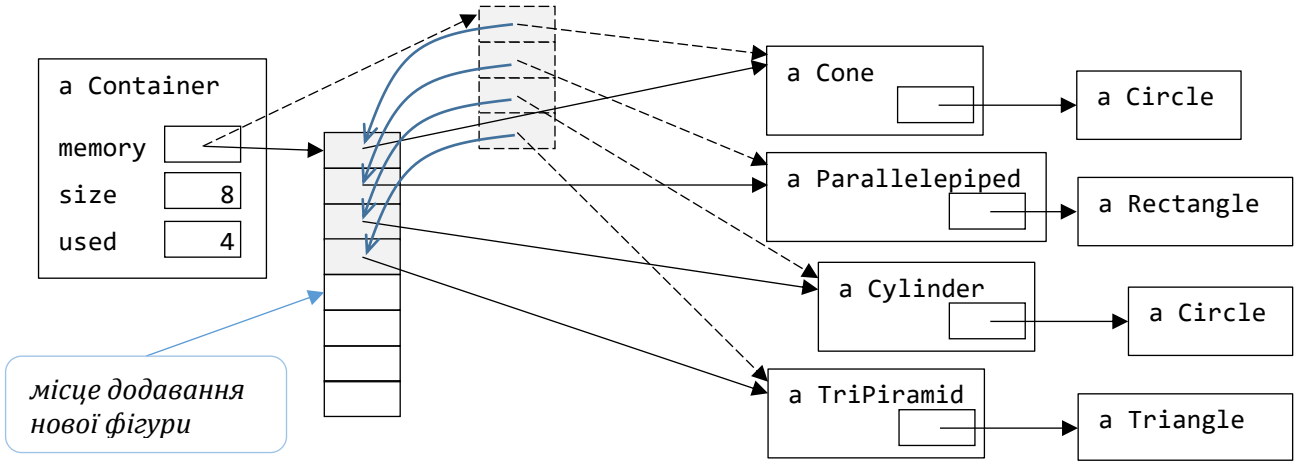

Рис. 1. Процес збільшення пам’яті контейнера.

Схема демонструє непросте внутрішнє влаштування контейнера. Це означає, що в нього мав би бути «змістовний» деструктор, досить складний конструктор копіювання і оператор присвоєння. Складність копіювання у тому, що контейнер не знатиме, кого копіювати за поліморфним вказівником. Відповідальність за розпізнавання типу покладемо на базовий клас збереженого об’єкта.

Щоб забезпечити правильність роботи контейнера (створення фігур лише правильних типів, доступ за правильним індексом) – виконуватимемо необхідні перевірки, про проблеми сигналізуватимемо об’єктами-винятками.

Як наділити контейнер універсальною здатністю виконувати довільну дію з кожним елементом? Контейнер «знає» своє внутрішнє влаштування й «уміє» перебирати елементи, але нічого не може знати про бажання майбутніх користувачів: які саме дії їм будуть потрібні. Вихід один: зробити таку дію параметром методу, що перебирає елементи і застосовує дію до кожного з них. Подібно можна задавати критерій пошуку.

Файл ShapesArray.h

```
// тип функції, яку можна застосувати до кожного елемента контейнера
typedef void Action(VolShape*);

// тип критерію для перевірки(пошуку) елементів контейнера
typedef bool Predicate(VolShape*);

class ShArray
{
private:
	int size, used;			// розміри виділеної та зайнятої пам'яті
	VolShape** mem;			// вказівник на саму пам'ять
	void checkMem();		// за потреби перерозподіляє пам'ять
	bool chekIndex(int i) const { if (i < 0 || i >= used) throw BadIndex(i); }
public:
	class BadIndex			// позначатиме помилку - неіснуючий індекс
	{
	public:
		int index;
		BadIndex(int i): index(i) {}
	};
// найменший контейнер може містити одну фігуру
	ShArray(): used(0), size(1), mem(new VolShape*[1]) {}

// можна наперед замовити довільний розмір виділеної пам'яті, щоб запобігти перерозподілам
// explicit захистить від випадкового перетворення цілого числа у контейнер
	explicit ShArray(int n): used(0), size(n), mem(new VolShape*[n]) {}
```

```
// обов'язково для класу, що використовує динамічну пам'ять
      ShArray(const ShArray &);
      ~ShArray();
      ShArray& operator=(const ShArray &);
// доступ до елементів
      ShArray& put(VolShape*);
      ShArray& insert(VolShape*, int); //throw(BadIndex&);
      ShArray& remove(int);            //throw(BadIndex&);
      ShArray& clear();
      VolShape*& operator[](int);      //throw(BadIndex&);
      const VolShape* operator[](int) const; //throw(BadIndex&);
// сервісні методи
      int high() const { return used - 1; }
      void loadFrom(std::ifstream&);   //throw(VolShape::BadClassname&);
// перебір елементів
      void printOn(ostream&) const;
      void doEach(Action) const;
      int detectFirst(Predicate) const;
};
```

Ми проектуємо достатньо «інтелектуальний» клас, тому і заголовковий файл вийшов чималенький. Спробуємо крок за кроком пояснити призначення та влаштування методів.

Користувач класу *ShArray* не повинен турбуватися по розмір масиву, клас повинен самостійно підлаштовуватися під потреби користувача. Один з поширених прийомів для вирішення такого роду завдань – використання динамічного масиву. Зазвичай пам'ять виділяють з невеликим запасом, щоб можна було швидко збільшувати розмір контейнера (в межах запасу). Далі відслідковують розмір виділеної (*size*) та зайнятої (*used*) пам'яті, за потреби виділяють нову, більшу ділянку пам'яті, куди переміщують наявні елементи – у нашому контейнері достатньо скопіювати вказівники. Для визначання нового розміру застосовують різні тактики: або збільшення на постійну величину, або подвоєння розміру, або комбінація цих двох підходів.

Перерозподіл пам'яті в класі *ShArray* виконує метод *checkMem()*:

```
// виділення більшої пам'яті для масиву, що зростає
void ShArray::checkMem()
{     if (used == size)
      {     size *= 2;
            VolShape** newMem = new VolShape*[size];
            // копіювання вказівників на наявні об'єкти
            for (int i=0; i<used; ++i) newMem[i] = mem[i];
            delete [] mem;
            mem = newMem;
      }
}
```

Метод оголошено в приватній частині класу, оскільки він є способом реалізації динамічного перерозподілу пам'яті і не стосується інтерфейсу класу. Зверніть увагу на те, що самі об'єкти, збережені в контейнері, залишаються на місці: ми копіюємо лише вказівники на них. При бажанні для контейнера можна використати не векторну, зв'язну пам'ять – не масив, а лінійний список. Така зміна реалізації не вплине на інтерфейс.

Одна з найпоширеніших помилок при роботі з масивами – вказання неправильного індекса, номера, що лежить поза межами масиву. Користувач контейнера міг би робити додаткові перевірки, щоб не ставалося таких помилок. Але існує сотня причин, щоб не робити цього: забув, ліньки, зайві галуження затуманюють основний алгоритм тощо.

Обов'язок перевіряти правильність індекса доречно покласти на розробника контейнера, бо йому це найлегше, і він може охопити всі випадки. Отримуємо таку ситуацію: розробник вміє виявляти помилки, але не знає, як їх виправляти; користувач міг би виправити свій індекс, але не знає, коли стається помилка. Щоб ці двоє порозумілися,

використовують спеціальний сигнальний механізм – запуск винятка. Ідея проста: програмний код, що виявив помилку, перериває звичайне виконання і запускає спеціальний об'єкт-виняток (у C++ винятком може бути будь-яка сутність, навіть, ціле число). Користувач програмного коду, зважаючи на можливість виникнення помилок, запускає його в спеціальному, захищеному режимі, що дає йому змогу перехопити виняток і відреагувати на помилку належним чином. Помилку розпізнають за типом об'єкта-винятка.

Зазвичай для позначення помилок оголошують нові класи. У нашому прикладі такий клас – *BadIndex*. У нього дуже просте влаштування, бо його головне призначення – повідомити тип. Транспортування неправильного номера – додаткова можливість. У нашому прикладі винятки можуть виникати в методах доступу до елемента контейнера: кожен з них перевірятиме правильність індекса єдиним методом *checkIndex()*, котрий і запустить виняток. Метод *checkIndex()* також оголошено приватним.

Один з конструкторів класу *ShArray* має єдиний параметр типу *int*. Відомо, що такі конструктори діють, як перетворювачі типу. Якщо в деякому контексті, де мав би бути об'єкт-контейнер, раптом виявляється ціле число, то «добрий» компілятор автоматично викликає конструктора і перетворює це число на порожній контейнер з відповідним обсягом зарезервованої пам'яті. Нечасто таке перетворення буває саме тим, чого хотів програміст. Аби запобігти автоматичному виклику конструктора, його позначають модифікатором *explicit*, який дозволяє виключно явний виклик.

Влаштування деструктора треба обговорити. Очевидно, що він мусить звільняти свою внутрішню пам'ять. Ми могли б оголосити це так:

```
ShArray::~ShArray()
{
	delete [] mem;
}
```

Тепер спробуємо використати контейнер фігур:

```
	ShArray Train; Cylinder C1; Cylinder* C2 = new Cylinder();
	Train.put(&C1); Train.put(C2);
	Train.put(new Parallelepiped());
```

Що станеться, коли контейнер *Train* перестане існувати? Циліндри *C1* та **C2* можна буде використовувати й надалі, а от паралелепіпед буде втрачено, причому сам об'єкт залишиться в динамічній пам'яті, а вказівник на неї зникне разом з контейнером – відбудеться втрата динамічної пам'яті.

На початку лекції ми вирішили, що контейнер є власником своїх елементів, то нехай він звільняє пам'ять від вмісту масиву та від нього самого:

```
ShArray::~ShArray()
{
	for (int i=0; i<used; ++i) delete mem[i];
	delete [] mem;
}
```

Тепер у нашому прикладі з'являться інші проблеми. Зі звільненням паралелепіпеда все гаразд, але, коли контейнер *Train* перестане існувати, *C2* стане «висячим» вказівником, міститиме адресу об'єкта, якого вже нема. А спроба звільнити пам'ять від циліндра *C1* призведе до аварійного завершення програми, бо він не займає динамічну пам'ять.

Обидва варіанти деструктора мають право на існування і потребують докладного опису в документації. Ми доведемо «ідею власності» до кінця і використаємо другий. При цьому зазначимо, що до контейнера можна додавати лише динамічні об'єкти, тобто, додавання *Train.put(&C1);* – заборонено. Виконувати *Train.put(C2);* можна, але одразу після цього потрібно перервати зв'язок вказівника з об'єктом: *C2 = nullptr;*.

Намагання оголосити конструктор копіювання (і оператор присвоєння) натикається на неочікувані труднощі. Конструктор мав би виконати глибоке копіювання – відтворити

структуру пам'яті контейнера та скопіювати кожен об'єкт, елемент контейнера. Але наш масив містить поліморфні вказівники, які можуть вказувати на довільну об'ємну фігуру. Нам же для копіювання об'єктів потрібно знати їхній тип.

Ми знаємо, що в класі *VolumeShape* є віртуальні методи, а для таких класів підтримується структура RTTI. З її допомогою можна довідатися конкретний тип об'єкта за вказівником. Це – хороша новина. Погана новина така: явне з'ясування типу за допомогою розгалужених перевірок є ознакою поганого стилю програмування або просто помилкового підходу до побудови об'єктно-орієнтованої програми. Ми доведемо до кінця ідею з використанням RTTI, але хоча б заховаємо подалі від людських очей галуження з перевірками.

```
ShArray::ShArray(const ShArray & A): used(A.used), size(A.used), mem(new VolShape*[A.used])
// відповідальним за розпізнавання типу є статичний метод класу VolumeShape
{
      for (int i = 0; i < used; ++i) mem[i] = VolumeShape::CopyInstance(A.mem[i]);
}

// «Цитата» з оголошення класу VolumeShape: статичний метод для виготовлення
//   копій екземплярів відомих підкласів VolumeShape
VolumeShape* VolumeShape::CopyInstance(VolumeShape* v)
{
      if ( typeid(*v) == typeid(Cylinder) )
            return new Cylinder(*dynamic_cast<Cylinder*>(v));
      if ( typeid(*v) == typeid(Parallelepiped) )
            return new Parallelepiped(*dynamic_cast<Parallelepiped*>(v));
      if ( typeid(*v) == typeid(TriPrism) )
            return new TriPrism(*dynamic_cast<TriPrism*>(v));
      if ( typeid(*v) == typeid(Cone) ) return new Cone(*dynamic_cast<Cone*>(v));
      if ( typeid(*v) == typeid(RectPiramid) )
            return new RectPiramid(*dynamic_cast<RectPiramid*>(v));
      if ( typeid(*v) == typeid(TriPiramid) )
            return new TriPiramid(*dynamic_cast<TriPiramid*>(v));
      throw BadClassname("Unknown type of Volume Shape encountered");
}
```

Це кострубато, але воно працює. Як тільки в ієрархії *VolumeShape* з'явиться новий підклас, доведеться модифікувати цей метод. Запуск винятка передбачено на той випадок, коли користувач спробує копіювати об'єкт, тип якого не передбачено в методі *CopyInstance*. Локалізація такого роду перевірок в методі класу – найменше зло. Принаймні, все розташовано в одному місці, поруч з джерелами походження об'єктів. Різноманітні користувачі ієрархії можуть безпечно використовувати *CopyInstance* для своїх потреб, не здогадуючись про його влаштування.

Проблема копіювання об'єктів невідомого типу має інше, набагато красивіше вирішення. Його можна знайти в літературі, присвяченій патернам проектування (патерн – типове рішення поширеної проблеми, золотий фонд програмістської майстерності). Потрібний нам патерн називається «Прототип». Хто знає тип об'єкта? – Він сам. То нехай сам себе і копіює!

Щоб реалізувати патерн, доведеться вдосконалити ієрархію *VolumeShape*: додамо до базового класу віртуальний метод *clone()* та перевизначимо його в конкретних підкласах.

```
class VolShape
{protected:
      double h;
      Shape* base;
public:   . . . . .
      virtual VolShape* clone() = 0;
      . . . . .
}
class Cylinder : public DirectShape
{public:   . . . . .
      virtual Cylinder* clone() { return new Cylinder(*this); }
};
```

```
class Parallelepiped : public DirectShape
{public:   . . . . .
      virtual Parallelepiped* clone() { return new Parallelepiped(*this); }
};
. . .
```

Коротко і ясно! Тепер конструктор копіювання контейнера матиме вигляд:

```
ShArray::ShArray(const ShArray & A): used(A.used), size(A.used), mem(new VolShape*[A.used])
{
      for (int i = 0; i < used; ++i) mem[i] = A.mem[i]->clone();
}
```

Оператор присвоєння напишіть самі з використанням описаного підходу.

Зверніть увагу на те, як перевантажено метод *clone()* у підкласах. Він повертає вказівник на конкретний тип, а не на абстрактний, як визначено в базовому класі. Тут все правильно. Завдяки висхідній сумісності типів (вказівників на підкласи з вказівником на базовий клас) застосоване перевантаження працює правильно.

Оператор індексування *operator*[*int*] – звичайний засіб доступу до елементів послідовного контейнера, як наш масив. Зазвичай визначають пару таких операторів: звичайний і константний. Перший з них ще називають селектор-модифікатор, а другий – селектор, селектор для читання. Константний оператор індексування дає змогу індексувати колекцію незмінних об'єктів. Зазвичай обидва оператори індексування мають однаковий тип результату – посилання на тип елемента, оскільки повертають не копію елемента, а сам елемент контейнера. Всередині такі оператори влаштовані однаково. Проте такий підхід виявляється хибним при роботі з масивами вказівників. Компілятор вважає помилковим оголошення *const VolShape*& operator[](int) const* – але, чому? Тип *const VolShape*&* означає «посилання на вказівник на незмінну фігуру», тобто, фігуру змінювати не можна, а вказівник – так, оскільки посилання дає повну владу над ним. Константний метод не може такого дозволити, тому оголошення не компілюється. Можливі два способи виправлення помилки: *const VolShape*& const operator[](int) const* – посилання також незмінне, тому вказівник захищено від змін, або *const VolShape* operator[](int) const* – повертається копія вказівника, а сам оригінал залишається недоторканим.

```
// оператор надає доступ для "читання-запису" до наявних елементів контейнера
VolShape*& ShArray::operator[](int i)
{
      chekIndex(i);
      return this->mem[i];
}
```

Зверніть увагу на те, що такий оператор не захищає контейнер від неправильних дій користувача. Наприклад, вказані нижче дії спричинятимуть проблеми

```
      ShArray Train; Cylinder C1; Cylinder* C2 = new Cylinder();
      Train.put(new Cylinder(C1)); Train.put(C2); C2 = nullptr;
      Train.put(new Parallelepiped());
      cout << Train[0] << Train[1] << Train[2]; // до цього місця все гаразд
      Train[0] = new Parallelepiped(); // погано – циліндр втрачено
      Train[1] = nullptr; // ще гірше, бо і циліндр втрачено, і mem[1]-> не діє
      delete Train[2];    // погано, бо mem[2] не визначено
```

«Мудрий» контейнер міг би завантажувати свої елементи з файла. У класі *ShArray* – це метод *loadFrom()*. Такий метод повинен розрізняти певний формат зберігання інформації про об'єкти, впізнавати тип, зчитувати відповідні дані та створювати екземпляри. Такий формат обумовлює розробник класу. Формат повинен бути зручним і добре документованим. Наприклад, таким: перший рядок файла – ціле число, що задає кількість об'єктів, кожен наступний рядок описує один об'єкт як ідентифікатор типу і числові значення полів даних об'єкта цього типу. Зразок файла:

```
4
Cylinder 3.5 1.5
Parallelepiped 3.0 3.0 3.0
TriPrism 4.1 2.0 2.0 60
Cone 4.2 1.5
```

«Почесний обов'язок» розпізнавати ідентифікатори і створювати екземпляри відповідних класів, як і раніше, покладемо на метод базового класу *VolumeShape*. Розпізнавання – річ ризикована. Хтось може вказати у файлі неправильний ідентифікатор. Тоді станеться помилка, про яку метод повідомить винятком. Клас винятку потрібно оголосити всередині *VolumeShape*.

```
// Знову «цитата» з оголошення класу VolumeShape: статичний метод для завантаження
//   екземплярів відомих типів об'ємних фігур з файла
VolumeShape* VolumeShape::MakeInstance(std::ifstream& fin) throw(BadClassname&)
{
	int y; double h, r, a, b;
	char name[40] = {0};
	fin >> name;
	if (strcmp(name, "Cylinder")==0)
	{	fin >> h >> r; return new Cylinder(h,r);
	}
	if (strcmp(name, "Parallelepiped")==0)
	{	fin >> h >> a >> b; return new Parallelepiped(h,a,b);
	}
	if (strcmp(name, "TriPrism")==0)
	{	fin >> h >> a >> b >> y; return new TriPrism(h,a,b,y);
	}
	if (strcmp(name, "Cone")==0)
	{	fin >> h >> r; return new Cone(h,r);
	}
	if (strcmp(name, "RectPiramid")==0)
	{	fin >> h >> a >> b; return new RectPiramid(h,a,b);
	}
	if (strcmp(name, "TriPiramid")==0)
	{	fin >> h >> a >> b >> y; return new TriPiramid(h,a,b,y);
	}
	throw BadClassname(name);
}

// метод завантажує з файла інформацію про об'ємні фігури
// формат даних має бути правильним
void ShArray::loadFrom(std::ifstream& fin) throw(VolShape::BadClassname&)
{
	this->clear();
	int k, i; fin >> k;
	// якщо ім'я класу у файлі задано неправильно, станеться виняток
	try
	{	for (i = 0; i < k; ++i) this->put(VolShape::MakeInstance(fin));
	}
	// можемо частково опрацювати цей виняток: правильно задати розмір контейнера
	// далі - повторний запуск, щоб користувач вирішував, що робити ще
	catch (VolShape::BadClassname& bcn)
	{
		used = i; throw;
	}
}
```

Клас контейнера перехоплює виняток, що сигналізує про помилку задання типу фігури, і намагається зберегти цілісність стану контейнера: фіксує, скільки фігур було завантажено без помилок. Проте, це ще не все опрацювання. Користувача також треба повідомити про те, що під час завантаження фігур щось пішло не так. Тому перехоплений виняток запускають повторно.

Передбачено два методи для додавання до контейнера нових елементів: *put*(*VolShape**) дописує нову фігуру в кінець контейнера, а *insert*(*VolShape**, *int*) – вставляє на задане місце. Кожен з них збільшує обсяг зайнятої пам'яті, тому попередньо викликає допоміжний *checkMem*(). Метод вставляння влаштовано складніше, оскільки він мусить звільнити місце в масиві для нового елемента і посунути збережені вказівники праворуч на одну позицію.

Розглянемо ще методи перебору елементів контейнера. Реалізувати виведення в потік просто:

```
void ShArray::printOn(ostream& os) const
{
	if (used == 0) os << "Empty Array of VolumeShapes\n";
	else
	{	os << "Array of " << used << " VolumeShapes:\n";
		for (int i = 0; i < used; ++i) os << *mem[i] << '\n';
		os << "---\n";
	}
}
```

Складніше виконати деяку дію з кожним елементом контейнера, якщо ми наперед не знаємо, чого може захотіти користувач. Надамо йому деякий загальний метод, що приймає вказівник на функцію. Користувач зможе налаштувати його на довільну дію, описавши її у функції з відповідним прототипом. Така ж ідея лежить і в основі пошуку елемента за деякою загальною ознакою. Метод налаштовують зовнішньою функцією, що повертає *bool*. Метод поверне номер першого знайденого елемента контейнера або значення, більше за кількість елементів у ньому.

```
// тип функції, яку можна застосувати до кожного елемента контейнера
typedef void Action(VolShape*);

void ShArray::doEach(Action f) const
// Застосовує дію f до кожного елемента контейнера
{
	for (int i = 0; i < used; ++i) f(mem[i]);
}

// тип критерію для перевірки(пошуку) елементів контейнера
typedef bool Predicate(VolShape*);

int ShArray::detectFirst(Predicate condition) const
// Знаходить номер першого елемента контейнера, для якого виконано condition
// Повертає знайдений номер, або розмір контейнера, якщо пошук завершився невдачею
{
	int i = 0;
	for ( ; i < used; ++i)
		if (condition(mem[i])) break;
	return i;
}
```

Тепер надрукувати об'єм кожного елемента контейнера можна так:

```
void printVol(VolShape* s)
{
	std::cout << *s << "\n     has volume V = " << s->volume() << '\n';
}
int main()
{
	ShArray A(5);
	...
	A.doEach(printVol);
}
```

Або ще простіше за допомогою лямбда-виразу:

```
A.doEach([](VolShape* v){
      std::cout << *v << "\n     has volume V = " << v->volume() << '\n'; });
```

Пошук першого елемента з достатньо великою площею основи:

```
k = A.detectFirst([](VolShape* S){ return S->baseSquare() > 20.1; });
if (k > A.high()) std::cout << "Nothing detected\n";
else std::cout << "Shape detected " << *A[k] << '\n';
```

Далі потрібно ознайомитися з кодом програмного проекту та проекспериментувати з ним. Ви могли б наділити контейнер здатністю індексувати свої елементи, починаючи з 1, а не з 0, або з довільного заданого числа. Корисно було б додати методи відшукання найбільшого та найменшого елементів, обчислення суми площ, об’ємів, або замість усіх цих методів запропонувати, як всі дії можна виконати за допомогою універсального методу *doEach*.