7. Ієрархічне програмування

Як писав Едсгер Дейкстра в 1975 році [25], програмування належить до найскладніших галузей прикладної математики. Відтоді воно хоч і вийшло за межі прикладної математики, але поєднало в собі елементи інженерних і фундаментальних наук, а отже, стало ще складнішим. Цю думку поділяють не лише студенти-інформатики, готуючись до іспитів із програмування, а й самі розробники програмного забезпечення.

Говорячи про складність програмування, ми не заперечуємо, що є й прості програми, зокрема подібні до розглянутих у перших розділах. Проблеми ж, про які йтиметься тепер, стосуватимуться програмних систем, що за складністю наближаються до так званих промислових програмних продуктів. Їхні властивості – тривалий час життя, багато користувачів і великий колектив розробників.

Проблема складності програмного забезпечення постійно перебуває в центрі уваги вчених і розробників. Фредерік Брукс [26], обґрунтовуючи неможливість створення «срібної кулі» – універсального інструмента, який на порядок підвищив би продуктивність праці програмістів, – твердить, що складність програмного забезпечення криється в природі самих програмних систем. Інший авторитет у галузі програмної інженерії Греді Буч [27] зазначав, що складність програмування спричинена внутрішньою складністю самих програм, яку неможливо усунути, але якою можна керувати, уносячи порядок до хаосу, що складається з мільйонів команд. Неорганізована складність призводить до того, що будь-яка команда в програмі може вплинути на поведінку дуже віддаленої її частини.

Ефективність застосування об'єктної декомпозиції програм можна пояснити обмеженням неконтрольованих впливів усередині програми. Зв'язки всередині об'єкта сильніші, ніж зв'язки поза ним. Цей постулат об'єктного програмування, який підтримує прийнята нами концепція закритих атрибутів, допомагає приховати значну частину складності класу всередині об'єкта, виносячи назовні лише зв'язки з іншими об'єктами.

На фізичному рівні окремого класу складністю керують, поділяючи клас на визначення та реалізацію. Визначення зазвичай розміщують у заголовному файлі, а реалізацію переносять до окремого файлу. Це дає змогу не обтяжувати визначення деталями реалізації там, де вони несуттєві. Особливе місце у визначенні класу належить його відкритій частині, яка містить *інтерфейс* класу, оскільки саме від нього залежить, якими способами клієнти класу використовуватимуть його.

Наступний крок у керуванні складністю переводить нас на рівень взаємодії об'єктів. Потрібно відрізняти динамічну взаємодію, яка може ви-

никати між конкретними об'єктами в певні періоди їхнього життєвого циклу, від сталої взаємодії, властивої всім об'єктам певного класу впродовж усього життєвого циклу. Називатимемо її взаємодією класів. Вона має статичний характер у тому розумінні, що виникає до виконання програми та підтримується впродовж усього часу виконання. Взаємодію як окремих об'єктів, так і цілих класів доцільно організувати ієрархічно – у вигляді ієрархій об'єктів та ієрархій класів. Це дає змогу керувати складністю програмних систем.

7.1. Ієрархія об'єктів

Перший крок у керуванні складністю ми вже, власне, зробили, застосувавши вкладення об'єктів – агрегацію та композицію (див. підрозділ 5.3). Цей тип ієрархії називають *ієрархією об'єктів*.

Об'єктно-орієнтоване програмування за своєю суттю ієрархічне. Уже самі поняття класу й об'єкта можна вважати проявами програмної ієрархії, яка задає підпорядкованість нестатичних членів-даних класу (атрибутів) – своєму об'єкту, а функцій-членів класу (методів), як і статичних членів-даних, – самому класу. Якщо атрибути самі виявляються об'єктами якихось класів, то підпорядкованість атрибутів своєму об'єкту зумовлює ієрархію об'єктів. Саме її ми вивчатимемо тепер, а підпорядкованістю методів класам займемося пізніше.

Отже, розглянемо детальніше ієрархію об'єктів, а саме деякі її властивості, суттєві з погляду вивчення інших типів ієрархій.

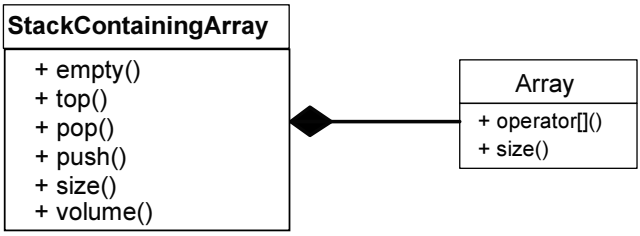

Рис. 7.1. Діаграма класу стеків із вкладеним масивом

Візьмемо пару об'єктів із попереднього розділу, а саме стек і вкладений у нього масив. Із діаграми об'єктів (рис. 7.1) видно, що екземпляр `_stackArray` масиву `Array` – прихована частина об'єкта класу `StackContainingArray`, оскільки відповідно до загальної домовленості щодо прав доступу атрибут міститься в закритій частині класу.

Через це клієнти класу `StackContainingArray` не мають доступу до масиву `_stackArray`. Тому їм недоступна навіть відкрита частина масиву,

наприклад операція індексування. Така організація прав доступу, про що вже йшлося раніше, захищає стек від несанкціонованого втручання в його вміст поза дозволеними операціями доступу.

Наведений далі код класу стеків значною мірою повторює визначення стека з лістингу 6.39. Обробку аварійних ситуацій пропустимо, пропонуючи читачеві самостійно внести відповідні доповнення.

Лістинг 7.1

```
// Стек із вкладеним до нього масивом
template <class Elem>
class StackContainingArray {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem> _stackArray; // Масив для розміщення стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackContainingArray(const size_t size):
      _stackArray(size), _top(_bos) {}
   ~StackContainingArray() {} // Деструктор
   bool empty() const {return _top==_bos;} // Чи порожній стек?
// Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
   size_t size() const {return _top+1;} // Видати поточний розмір стека
// Видати ємність контейнера
   size_t volume() const {return _stackArray.size();}
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {return _stackArray[_top];}
   void pop() {_top--;}  // Виштовхнути верхівку стека
   void push(Elem value) {_stackArray[++_top]=value;} // Помістити в стек
};
template <class Elem> const size_t StackContainingArray<Elem>::_bos=-1;
```

Отже, масив `_stackArray` – член стека з тими самими правами, що й інші члени цього класу, зокрема маркер верхівки стека `_top`. Важливо розуміти, що права доступу до об'єкта і його частин залежать від того, де міститься цей об'єкт. Позаяк масив `_stackArray` розміщено в закритій частині класу `StackContainingArray`, то доступ до нього ззовні стека закрито. Нагадаємо визначення класу масивів у трохи скороченому вигляді, знову ж пропускаючи деталі обробки особливих ситуацій.

Лістинг 7.2

```
// Простий масив елементів
template <class ArrayElem>
```

```
class Array {
public:
// Конструктор, закритий для конвертування типів
   explicit Array (const size_t sz): _size(sz),
      _array(new ArrayElem[sz]) {}
// Деструктор
   ~Array() {delete [] _array;}
// Селектор елемента масиву: операція сталого індексування
   const ArrayElem& operator[] (const size_t index) const
      {return _array[index];}
// Селектор-модифікатор елемента масиву: операція індексування
   ArrayElem& operator[] (const size_t index) {return _array[index];}
// Видати розмірність масиву (селектор)
   size_t size() const {return _size;}
private:
   size_t _size;   // Розмірність масиву
   ArrayElem* _array;   // Вміст масиву
// Операції та функції, заборонені для використання в масивах
   Array (const Array&); Array& operator=(const Array& a);
};
```

Зробимо перший висновок щодо ієрархії об'єктів. Композит (у розглядуваному випадку – стек) містить у собі як члена класу об'єкт – екземпляр компонента (тут – масив). Говорячи про відношення між компонентом і композитом, розрізнятимемо право власності та право доступу. Компонент перебуває у власності композита. Утім володіння компонентом не дає композиту виняткових прав доступу до нього. Композит поводиться як звичайний клієнт свого компонента. Права доступу композита стосуються лише відкритої частини компонента. Так, стек має доступ до конструктора й деструктора масиву, операторів індексування та функції обчислення розміру масиву.

Разом із цим композит має права виняткової власності на компонент, приховуючи його від інших об'єктів. Клієнти композита не мають доступу ні до самого компонента, ні навіть до його відкритої частини. Водночас композит може відкрити доступ до свого компонента повністю або частково за допомогою вже відомого засобу делегування. Зокрема, стек делегує право на виконання своєї відкритої функції-члена `volume()` масиву, доручивши це функції-члену масиву `size()`.

Вправа 7.1

Складіть тестову програму та виконайте тестування класу `StackContainingArray`.

Вправа 7.2

Доповніть класи механізмами обробки аварійних ситуацій, зумовлених спробами прочитати верхівку чи виштовхнути елемент порожнього стека, додати елемент до переповненого стека.

Вправа 7.3

За зразком класу стеків із лістингу 7.1 розробіть і реалізуйте параметризований клас `StackAggregatingArray` обмежених стеків, атрибут якого `_pstackArray` був би приєднаним за допомогою указника

```
Array<Elem>* _pstackArray;
```

Порівняйте класи `StackAggregatingArray` і `StackContainingArray` з точки зору можливості розширення ємності стека.

Вправа 7.4

Сформуйте параметризований клас `UnboundedStackOnArray` необмежених стеків на базі масиву.

Вправа 7.5

Розробіть параметризований клас `UnboundedStackOnList` необмежених стеків на базі списку.

Як відомо, вимоги до програм уточнюються та змінюються в процесі їх використання, тому для підтримки працездатності програм доводиться вносити до них зміни та навіть удаватися до перепрограмування. Цю діяльність зазвичай називають *супроводом* (maintenance) програми. Особливо цінують легкі в супроводі програми, легко налаштовувані на *нові застосування* (reuse) без суттєвого перепрограмування. (На думку автора, узвичаєний у літературі переклад терміна «reuse» російською мовою як «повторное использование» не зовсім відповідає суті поняття.)

Припустімо, що в процесі експлуатації виникла потреба доповнити стек засобом підглядання, тобто можливістю прочитати вміст його внутрішньої частини. Цього можна було б досягти за допомогою утиліти `peekback()`, застосувавши допоміжний стек:

Лістинг 7.3

```
// Утиліта підглядання за вмістом стека
template <typename T>
bool peekback(StackContainingArray<T>& s, size_t n, T& res) {
   StackContainingArray<T> ss(n); // Допоміжний стек
   bool found=false;
   for(size_t i=0; i<n; ++i) {   // Цикл підходу до шуканого елемента
```

```
      T x=s.top();
      if(res==x) {
         found=true; // Шуканий елемент знайдено
         break;
      }
   // Перекладаємо проміжний елемент до допоміжного стека
      s.pop();
      ss.push(x);
   }
   while(!ss.empty()) {   // Цикл відновлення вмісту стека
      T x=ss.top();
      ss.pop();
      s.push(x);
   }
   return found;
}
```

Аналізуючи утиліту peekback(), візьмемо до уваги одну очевидну недоладність: стек, до якого ми підглядаємо, реалізовано на базі масиву – структури даних із прямим доступом, а утиліта використовує послідовний доступ. Прямий доступ під час підглядання реалізуємо в новому класі, який назвемо стеком із підгляданням PeekBackStackContainingArray. Значна його частина повторює звичайний стек.

Лістинг 7.4

```
// Стек із підгляданням на базі масиву
template <class Elem>
class PeekBackStackContainingArray {
// Спільна зі стеком частина
private:
   static const size_t _bos;  // Дно стека
   size_t _top;  // Верхівка стека
   Array<Elem> _stackArray;   // Масив для розміщення стека
public:
// Конструктор, закритий від конвертування типів
   explicit PeekBackStackContainingArray (const size_t size):
      _stackArray(size), _top(_bos) {}
   ~PeekBackStackContainingArray() {}  // Деструктор
   bool empty() const {return _top==_bos;}  // Чи не порожній стек?
   // Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
   size_t size() const {return _top+1;} // Видати поточний розмір стека
// Видати ємність контейнера
   size_t volume() const {return _stackArray.size();}
```

```
// Керування стеком:
// Видати верхівку стека
   const Elem& top() const {return _stackArray[_top];}
   void pop() {_top--;}  // Виштовхнути верхівку стека
// Помістити в стек
   void push(Elem value) {_stackArray[++_top]=value;}
// Додано функцію підглядання
   bool peekback(const size_t index, Elem& elem) const {
      if (index>_top) return false;
      elem=_stackArray[index];
      return true;
   }
};
```

Вправа 7.6

Складіть тестову програму та протестуйте клас `PeekBackStackContainingArray`.

Проаналізуємо зроблене. На рис. 7.2 показано склад класів і відношення між ними.

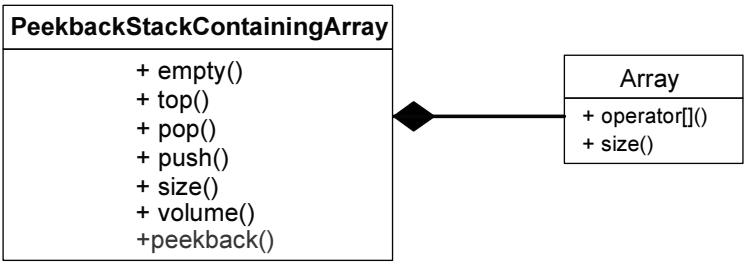

Рис. 7.2. Діаграма класу стеків з підгляданням

Досягненням можна вважати те, що готовий програмний текст класу стеків можна використовувати як значну частину нового класу: власне, до нього дописано одну нову функцію підглядання. Вада ж полягає в дублюванні коду цих двох класів. Зазвичай його вважають небажаним, оскільки дублювання породжує потребу постійно відстежувати відповідність кодів упродовж усього життєвого циклу програмної системи. Дублювання спричиняє ускладнення супроводу, тому що доводиться повторювати кожну правку, унесену до класу `StackContainingArray`, у тій частині `PeekBackStackContainingArray`, яка його повторює. Вихід полягає у використанні об'єктного, а не текстового коду базового класу `StackCo-`

ntainingArray. Це породжує принципово відмінний тип ієрархії – ієрархію класів, до розгляду якої ми перейдемо в наступному підрозділі. Але спочатку зробимо два зауваження.

У такий само спосіб, як і для стека з вбудованим масивом, можна задати стек із підгляданням, створений на агрегованому масиві.

Вправа 7.7

Створіть клас PeekBackStackAggregatingArray стеків із підгляданням, базуючись на класі StackAggregatingArray.

Для стека на базі списків підглядання реалізувати складніше. Звісно, можна скористатися запропонованою вище утилітою. Інший спосіб – включити до стека додаткову структуру даних для реалізації прямого доступу.

Вправа 7.8

Створіть клас PeekBackStackOnList стеків із підгляданням на базі класу UnboundedStackOnList.

7.2. Ієрархія класів: відкрите успадкування

Перейдемо тепер до вивчення найважливішої концепції об'єктно-орієнтованого програмування, яка, власне, і зумовлює виникнення ієрархії класів. Вона одержала назву *успадкування* (inheritance), а точніше – *відкритого успадкування* (public inheritance).

Варто почати з простого прикладу. Візьмемо клас прямокутників Rectangle:

Лістинг 7.5

```
typedef double length_t;    // Тип довжини сторони
// Клас прямокутників
class Rectangle {
private:
   length_t _height; // Висота
   length_t _width; // Ширина
public:
// Конструктор
   Rectangle (const length_t height, const length_t width);
   ~Rectangle(); // Деструктор
   Rectangle (const Rectangle&); // Копіювальний конструктор
   Rectangle& operator=(const Rectangle&); // Присвоєння
// Селектори сторін
   length_t getA() const; length_t getB() const;
```

```
// Модифікатори сторін
   Rectangle& setA(const length_t); Rectangle& setB(const length_t);
// Функції обчислення характеристик:
   length_t area() const; // площі
   length_t diagonal() const; // діагоналі
};
// Операція виведення прямокутника
ostream& operator<<(ostream&, const Rectangle&);
```

Клас `Rectangle` має звичайний набір членів: закриті атрибути довжини та ширини, зрозумілі конструктори, деструктор, оператор присвоєння, селектори та модифікатори, а також функції обчислення площі та діагоналі прямокутника. Наявність серед членів класу всіх функцій, окрім двох останніх, зрозуміла. Функція обчислення площі могла б бути утилітою – наприклад, із сигнатурою

```
length_t area(const Rectangle&);
```

як і функція обчислення діагоналі:

```
length_t diagonal(const Rectangle&);
```

Трохи пізніше ми проаналізуємо аргументи на користь того чи іншого рішення. Поки що залишимо обидві функції – утиліту й метод, реалізувавши одну через іншу:

Лістинг 7.6

```
// Реалізація утиліт через метод:
// обчислення площі
length_t area(const Rectangle& rec) {return rec.area();}
// обчислення діагоналі
length_t diagonal(const Rectangle& rec) {return rec.diagonal();}
```

Обмежившись лише утилітою, для обчислення площі чи діагоналі довелося б скористатися селекторами довжини та ширини, тоді як методи можуть звертатися до атрибутів напряму (що, можливо, не завжди добре):

Лістинг 7.7

```
// Реалізація функції обчислення площі
length_t Rectangle::area() const {return _height*_width;}
#include <cmath>
// Реалізація функції обчислення діагоналі
length_t Rectangle::diagonal() const {
   return sqrt(_height*_height+_width*_width);
}
```

Реалізація класу очевидна, але наведемо її повністю, щоб мати цілісну картину ієрархії, до якої ми рухаємося:

Лістинг 7.8

```
// Реалізація конструктора
Rectangle::Rectangle(const length_t height, const length_t width):
      _height (height), _width (width) {}
// Реалізація деструктора
Rectangle::~Rectangle() {}
// Реалізація копіювального конструктора
Rectangle::Rectangle (const Rectangle& rec):
      _height (rec._height), _width (rec._width) {}
// Реалізація присвоєння
Rectangle& Rectangle::operator=(const Rectangle& source) {
   _height=source._height; _width=source._width;
   return *this;
}
// Реалізація селекторів
length_t Rectangle::getA() const {return _height;}
length_t Rectangle::getB() const {return _width;}
Rectangle& Rectangle::setA(const length_t a) {_height=a; return *this;}
// Реалізація модифікаторів
Rectangle& Rectangle::setB(const length_t b) {_width=b; return *this;}
// Реалізація операції виведення прямокутника
ostream& operator<<(ostream& os, const Rectangle& rec) {
   os<<rec.getA()<<'x'<<rec.getB();
   return os;
}
```

Тепер виділимо важливий різновид прямокутника – квадрат. Зрозуміло, що кожен квадрат можна розглядати як прямокутник, але не навпаки. Узявши довільний прямокутник `source`, завжди можна перевірити, чи він не квадрат, порівнявши довжини сторін: `source.getA()==source.getB()`. Проте ця перевірка відбуватиметься вже на етапі виконання програми, тоді як ми прагнемо якнайбільше перевірок (передусім щодо типів об'єктів) зробити ще на етапі компіляції.

Властивості, які об'єкти зберігають упродовж усього життєвого циклу, називатимемо *статичними*, а інші, тобто ті, що можуть змінюватися під час виконання програми, – *динамічними*. «Бути прямокутником» – статична властивість, «мати певну довжину сторони» – зазвичай динамічна, оскільки довжину можна змінити, застосувавши відповідний модифікатор. Якщо квадрат розуміти як прямокутник із рівними сторонами, то властивість «бути квадратом» теж стає динамічною. Це добре чи погано?

Проведемо аналогію з числами. Візьмемо, наприклад, цілий і дійсний типи. Властивість належати до них статична. У цілого числа не може

з'явитися дробової частини, тоді як дуже легко одержати дійсне число, дробова частина якого – нуль. Однак від цього воно не змінює свого типу в тому розумінні, що воно зберігається в форматі дійсного числа і до нього й далі застосовуватимуться операції та функції з дійсними параметрами. Їх узагалі-то можна застосувати й до цілого числа, але попередньо перетворивши тип.

Подібно до того, як із дійсних чисел було виділено тип цілих чисел, хотілося б виділити клас квадратів, задавши його як окремий клас, наприклад

Лістинг 7.9

```
// Незалежний клас квадратів
class Square {
private:
   length_t _side; // Сторона
public:
   Square (const length_t); // Конструктор
   ~Square(); // Деструктор
   Square (const Square&); // Копіювальний конструктор
   Square& operator=(const Square&); // Присвоєння
   length_t getA() const; // Селектор сторони
   Square& setA(const length_t); // Модифікатор сторони
// Функції обчислення характеристик:
   length_t area() const; // площі
   length_t diagonal() const; // діагоналі
};
// Операція виведення квадрата
ostream& operator<<(ostream&, const Square&);
```

Однак із такого визначення не видно зв'язку між квадратами та прямокутниками. Щоб він виник, приєднаємо один клас до іншого, застосувавши механізм *відкритого успадкування* (public inheritance), яке полягає у визначенні нового класу квадратів `DerivedSquare` не окремо, а на основі базового класу `Rectangle` як похідного від нього:

Лістинг 7.10

```
// Клас квадратів, похідний від класу прямокутників
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {} // Деструктор
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
};
```

Що це означає? Передусім те, що об'єкт класу квадратів – водночас повноцінний об'єкт класу прямокутників. Заголовок

```
class DerivedSquare: public Rectangle
```

читають так: кожен об'єкт класу `DerivedSquare` є похідним від об'єкта базового класу `Rectangle`. Об'єкт «квадрат» створено з об'єкта «прямокутник», тому квадрат має всі властивості базового класу. При цьому зникає потреба дублювати функції, задані для прямокутників, у класі квадратів. Клієнти класу квадратів одержать доступ не лише до його відкритої частини, але й до відкритої частини базового класу прямокутників. Цей зв'язок однобічний: базовий клас (тут – прямокутник), не має й не може мати інформації про утворені від нього похідні класи.

Говорять, що похідний об'єкт – *уточнення*, чи *спеціалізація*, базового, тоді як базовий об'єкт називають *узагальненням* похідного.

Створення об'єкта будь-якого похідного класу складається з двох кроків. Спочатку працює конструктор базового класу, який створює базовий об'єкт. Після цього виконується тіло конструктора похідного класу. Природно, що для створення квадрата потрібно передбачити коректну ініціалізацію атрибутів прямокутника. Типова помилка може полягати в передаванні конструкторові базового класу неправильних параметрів.

Порівняємо два варіанти конструктора:

Лістинг 7.11

```
// Коректний конструктор квадрата: у списку
// ініціалізації конструктору прямокутника передано довжину сторони
DerivedSquare::DerivedSquare(const length_t side):
                                    Rectangle(side, side) {}
```

і

Лістинг 7.12

```
// Некоректний конструктор квадрата: список
// ініціалізації пропущено; буде створено прямокутник
// із заданими за замовчуванням сторонами
DerivedSquare::DerivedSquare (const length_t side) {}
```

Під час кожного створення об'єкта похідного класу конструктор базового класу виконується *незалежно* від того, чи є базовий клас у списку ініціалізації конструктора. Від його наявності залежить спосіб, яким йому буде передано параметри. Якщо його немає, то система програмування підставить виклик конструктора без параметрів або зі значеннями заданих за замовчуванням параметрів. Оскільки такого конструктора в класі прямокутників немає, спроба компілювати функцію з лістингу 7.12 призведе до синтаксичної помилки. Можна вважати, що нам пощастило: якби в прямокутника були задані за замовчуванням параметри, наприклад

```
// Конструктор
Rectangle (const length_t heigh=10, const length_t width=20);
```

то непомітно для користувача система програмування створила б замість квадрата із заданою під час виклику стороною прямокутник зі сторонами 10 і 20. Ось такий вийшов би «квадрат»!

Аналогічно можна реалізувати копіювальний конструктор квадрата і його оператор присвоєння:

Лістинг 7.13

```
// Реалізація копіювального конструктора квадрата
DerivedSquare::DerivedSquare (const DerivedSquare& source):
   Rectangle(source.getA(), source.getB()) {}
// Реалізація присвоєння квадратів
DerivedSquare& DerivedSquare:: operator=(const) DerivedSquare& source {
   return setA(source.getA());
}
```

У списку його ініціалізації знову бачимо прямокутник, сторони якого взято з оригіналу для копіювання.

До речі, копіювальний конструктор прямокутника можна застосувати до квадрата. Оскільки кожен квадрат – повноцінний прямокутник, його можна використовувати всюди, де передбачено використання прямокутника, зокрема передати як параметр у вигляді відсилки чи указника:

Лістинг 7.14

```
DerivedSquare square(5);
Rectangle copy(square); cout<<"Copied rec: "<<copy<<endl;
```

Результатом виконання стане прямокутник `copy` – точна копія квадрата `square`, який, утім, перестане бути екземпляром класу квадратів. Порівняйте з аналогічною ініціалізацією числових типів:

```
double d(1);
```

Поки що обмежимося цим зауваженням про використання похідних класів замість базових, відклавши детальний розгляд до ознайомлення з динамічним поліморфізмом.

Вправа 7.9

Складіть тестову програму та виконайте тестування класів `Rectangle` і `DerivedSquare`.

Вправа 7.10

a) Доповніть клас квадратів копіювальним конструктором, який перетворюватиме прямокутник з рівними сторонами на відповідний йому квадрат.

b) Реалізуйте метод присвоєння для класу квадратів.

Розглянемо детальніше будову класу квадратів. Як видно з його визначення в лістингу 7.10, власними елементами квадрата є лише конструктори, деструктор і присвоєння. Усе інше квадрат успадковує від базового класу прямокутників. Це означає, що квадрат має дві (але рівні) сторони, виражені атрибутами `_height` і `_width`, селектори та модифікатори цих атрибутів, функції обчислення площі та діагоналі, а також утиліту виведення.

Звісно, успадкування атрибутів і методів звільняє від потреби дублювати код, яке довелося зробити в лістингу 7.9. Проте наслідки успадкування не цілком відповідають нашим уявленням про коректну поведінку квадрата. Справді, подивимося, що станеться в разі спроби застосувати до квадрата модифікатори прямокутника. Виявляється, що при цьому квадрат може бути спотворено:

Лістинг 7.15

```
DerivedSquare square(10); // Спотворення квадрата
cout<<square<<endl; // Показуємо квадрат зі стороною 10
square.setB(20); // Модифікація лише однієї зі сторін
cout<<square<<endl; // Показуємо «дивний квадрат»: одна сторона – 10,
                    // друга – 20
```

Частковий вихід із ситуації некоректної поведінки методів базового класу стосовно об'єктів похідного полягає в тому, що слід відмовитися від певної частини методів базового класу та замінити їх на власні методи похідного класу. Для цього є дві можливості. Перша – перевизначити модифікатори сторін у класі квадратів:

Лістинг 7.16

```
// Квадрат, доповнений власними модифікаторами
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {} // Деструктор
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
// Власні модифікатори сторони квадрата
   DerivedSquare& setA(const length_t);
   DerivedSquare& setB(const length_t);
};
```

реалізувавши кожен із методів так, щоб зі зміною довжини однієї зі сторін базового прямокутника водночас відповідно змінювалася довжина другої сторони, залишаючи його квадратом:

Лістинг 7.17

```
// Одночасна модифікація обох сторін прямокутника
DerivedSquare& DerivedSquare::setA(const length_t a) {
// Потрібно замінити довжини обох сторін базового прямокутника
   Rectangle::setB(a); Rectangle::setA(a);
   return *this;
}
DerivedSquare& DerivedSquare::setB(const length_t b) {
   Rectangle::setB(b); Rectangle::setA(b);
   return *this;
}
```

Зверніть увагу на те, що в тілі методу `setA()` класу квадратів буде викликано модифікатори `setA()` та `setB()` класу прямокутників. Про це свідчить застосування операторів визначення області дії – `Rectangle::setA(a)` та `Rectangle::setB(a)`.

Проте це рішення певною мірою нейтралізує переваги успадкування, призводячи до часткового дублювання коду. Тому привабливішим може видатись інший спосіб – перевизначити лише один із модифікаторів, закривши доступ до іншого:

Лістинг 7.18

```
// Квадрат, доповнений власним модифікатором сторони A
// із закритим доступом до модифікатора сторони B
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {} // Деструктор
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
// Власний модифікатор однієї сторони квадрата
   DerivedSquare& setA(const length_t);
private:
   Rectangle::setB; // Закритий доступ до модифікатора сторони B
};
```

Реалізація залишається такою самою, як її подано в лістингу 7.17. Проте жодне із запропонованих рішень не розв'язує проблеми повністю. У разі як заміни обох модифікаторів (лістинг 7.17), так і закритого доступу до одного з них (лістинг 7.18), у квадрата залишається можливість доступу до модифікатора прямокутника за допомогою застосування оператора визначення області дії (перевірте).

Лістинг 7.19

```
// Спотворення квадрата через доступ до
// модифікатора базового прямокутника
DerivedSquare square(10); cout<<square<<endl;
```

```
// Знову модифікація лише однієї зі сторін
square.Rectangle::setB(20);
// Виводимо «дивний квадрат»: одна сторона – 10, друга – 20
cout<<square<<endl;
```

Виклик модифікатора `setB()`, як і раніше, спотворить квадрат, і його варто було б уважати синтаксично некоректним, адже доступ до методу було закрито. Однак завдяки гнучкості, властивій мові C++, похідний клас може закрити лише безпосередній доступ до відкритого методу базового класу за допомогою власного інтерфейсу, але не здатен заборонити кваліфікований (із застосовуванням кваліфікатора) доступ за допомогою оператора визначення області дії.

Для повного закривання методів базового класу використовують інші прийоми, наприклад закрите успадкування, яке буде розглянуто пізніше. Є також методи побудови ієрархії за допомогою абстрактних інтерфейсних класів, і вони позбавлені проблем, спричинених перевизначенням методів. Про це йтиметься в підрозділі 7.9.

Ми ж продовжимо налаштовувати клас квадратів на якнайкоректніше співіснування з базовим класом.

Якщо модифікатор сторони квадрата потребував скасування відповідного модифікатора прямокутника, то інші методи, наприклад, обчислення площі чи діагоналі, можуть бути коректно використані як базовим прямокутником, так і похідним від нього квадратом. Те саме стосується утиліт.

Лістинг 7.20

```
// Виклик квадратом утиліти й методу прямокутника
DerivedSquare square; cout<<square<<endl;
cout<<square<<" area "<<area(square)<<'('<<square.area()<<')'<<endl;
```

Похідні класи здатні модифікувати код базового класу шляхом заміщення його методів. Це стосується будь-якого методу базового класу. Наприклад, для методів обчислення площі `area()` та діагоналі `diagonal()` у класі квадратів розробник міг би прийняти рішення про ефективнішу реалізацію обчислень характеристик квадрата. Тоді це матиме такий вигляд:

Лістинг 7.21

```
// Квадрат, доповнений методами обчислення
// площі та діагоналі, власним модифікатором сторони A
// й із закритим доступом до модифікатора сторони B
class DerivedSquare: public Rectangle {
public:
   DerivedSquare (const length_t side=10); // Конструктор
   ~DerivedSquare() {}  // Деструктор
```

```
   DerivedSquare (const DerivedSquare&); // Копіювальний конструктор
   DerivedSquare& operator=(const DerivedSquare&); // Присвоєння
// Власний модифікатор однієї сторони квадрата
   DerivedSquare& setA(const length_t);
// Власні функції обчислення характеристик
   length_t area() const; length_t diagonal() const;
private:
   Rectangle::setB; // Закритий доступ до модифікатора сторони B
};
```

Як тільки у визначенні похідного класу з'явиться заміщена функція, безпосереднє використання базової функції буде скасовано. Тому заміщенню потрібно надати власну реалізацію:

Лістинг 7.22

```
// Реалізація функції обчислення площі квадрата
length_t DerivedSquare::area() const {return getA()*getA();}
// Реалізація функції обчислення діагоналі квадрата
const double sqrt2=sqrt(2.); //1.4142135623730950;
length_t DerivedSquare::diagonal() const {return getA()*sqrt2;}
```

Фрагмент програми з лістингу 7.20 тепер виконуватиметься інакше – з використанням утиліти прямокутника, але методу квадрата:

Лістинг 7.23

```
// Виклик квадратом утиліти прямокутників і методу квадратів
DerivedSquare square(10); cout<<square<<endl;
cout<<square<<" area "<<area(square)<<'('<<square.area()<<')'<<endl;
```

Можна замістити й утиліти, задавши їх окремо з параметрами похідного класу:

Лістинг 7.24

```
// Реалізація утиліти обчислення площі квадрата
length_t area(const DerivedSquare& square) {
   return square.getA()*square.getA();
}
// Реалізація утиліти обчислення діагоналі квадрата
length_t diagonal(const DerivedSquare& square) {
   return square.getA()*sqrt2;
}
```

Іще раз наведемо попередній фрагмент коду з лістингу 7.20. Тепер як метод, так утиліта вибиратимуться з класу квадратів:

Лістинг 7.25

```
// Виклик квадратом власних утиліти й методу
DerivedSquare square(10); cout<<square<<endl;
cout<<square<<" area "<<area(square)<<'('<<square.area()<<')'<<endl;
```

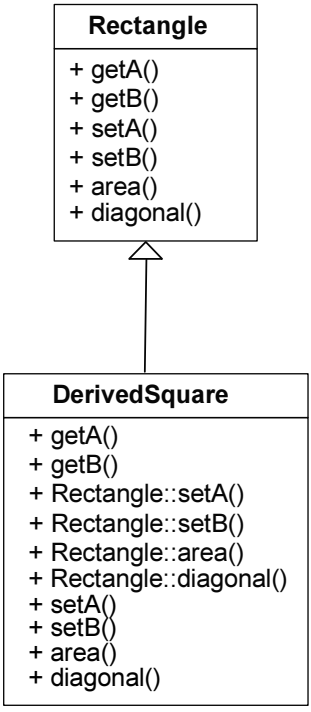

Рис. 7.3. Діаграма ієрархії класів прямокутників і квадратів

Заміщення методу базового класу власною реалізацією в похідному класі не зумовлює повної відмови від методу базового класу. Він залишається доступним для похідного класу, утім із застосуванням операції визначення області дії. Як ви бачили, визначення області дії дає змогу похідному класу не лише використовувати заміщений метод, але й відкрити доступ до методу, відкритого в базовому класі, але закритого в похідному.

Відношення двох розглянутих класів проілюстровано на рис. 7.3.

Із нього видно, що до об'єктів похідного класу можна безпосередньо застосовувати методи базового класу, якщо вони не конфліктують за іменами. У розгляданому випадку це селектори `getA()` і `getB()`. У разі наявності однойменних методів виклики методів похідного класу записуються звичним способом, а виклики методів базового класу потребують додаткового кваліфікатора.

Отже, відкрите успадкування дає похідному класу змогу повністю успадковувати поведінку базового, розширити її додатковими власними функціональними можливостями, у разі потреби уточнюючи або навіть скасовуючи поведінку (тобто методи) базового класу. Після детального вивчення можливостей успадкування ми дамо рекомендації щодо їх застосування.

Вправа 7.11

Задайте клас паралелограмів. Побудуйте на його основі ієрархію класів паралелограмів і прямокутників.

7.3. Ієрархія класів: типізація об'єктів

Коли не було ієрархії класів, поняття класу й типу перебували в однозначній відповідності: кожному класу відповідав рівно один тип, визначений цим класом. Якщо не брати до уваги наявності так званих *простих*

(старих) типів (plane old type), то кожен програмований тип задано якимось класом. Ієрархія класів (точніше – відкрите успадкування) зумовлює порушення цієї взаємної однозначності типів і класів. Тепер кожному похідному класу можна приписати принаймні два типи: його власний і визначений базовим класом. Така неоднозначність потребуватиме додаткових уточнень під час виклику функцій, які допускають заміщення в похідних класах.

Розглянемо детальніше деякі аспекти заміщення та визначимо його межі. Важливо розрізняти заміщення функцій і заміщення об'єктів, схоже на перетворення типів. Почнемо з останнього.

Як відомо з підрозділу 4.9, перетворення типів виконує конвертор. Ним може бути конструктор чи оператор перетворення типу. Зазвичай ці перетворення неявні, але вони завжди зумовлюють створення нового об'єкта потрібного типу, використовуючи початковий об'єкт як параметр. Якихось особливих обмежень на перетворення типів немає. Важливо лише, щоб було виконано одну з умов: або цільовий клас містить конструктор із параметром початкового класу, або початковий клас містить оператор перетворення свого об'єкта в об'єкт цільового класу.

Перетворення, які ми розглядали досі, не стосувались ієрархії. Так ми перетворювали дійсне число на комплексне чи об'єкт, що виражав довжину в британських мірах, в об'єкт у давньоруських. Наявність ієрархії класів надає поняттю перетворення типів нового змісту. Як щойно було продемонстровано, кожен квадрат водночас можна вважати прямокутником, тобто скрізь, де маємо прямокутник, дозволено використовувати квадрат, але не навпаки. Так само прямокутник або квадрат можна використати замість паралелограма, про що йшлось у вправі 7.11. Отже, використання успадкувань надає додатковий вимір для перетворень типів угору за ієрархією успадкувань, від похідного типу до базового. Тому кожен об'єкт похідного класу має багато типів, оскільки належить кожному зі своїх базових класів, яких може бути кілька.

У зв'язку з цим виникають дві можливості: перетворити об'єкт похідного типу до базового типу чи використовувати похідний об'єкт як базовий. Останнє завжди можливо, адже кожен похідний об'єкт містить у собі екземпляр базового, доповнений властивостями похідного. Застосування тієї чи іншої можливості залежить від способу доступу до об'єкта: одержуючи його за допомогою імені, указника чи відсилки, ми надаємо об'єкту тип відповідного іменування.

Розглянемо це детальніше. У разі визначення звичайного імені екземпляру класу, яке передбачає виклик конструктора, цим конструктором

буде створено об'єкт саме того типу, який приписано імені, що визначається. Конструктор, який це робить, однозначно визначається типом імені. Так, визначення

```
Rectangle rec(10, 20);
```

зумовлює створення прямокутника, а

```
DerivedSquare square(30);
```

створює квадрат.

Для створення нового об'єкта можна використати копіювальний конструктор. Так, визначення імені

```
Rectangle copyrec(rec);
```

створює копію `copyrec` прямокутника `rec`. Водночас визначення

```
Rectangle copysquare(square);
```

спричинить створення нового об'єкта перетворенням квадрата `square` на прямокутник `copysquare`.

Визначення ж відсилки чи указника не приводять до створення нових об'єктів. Відсилку буде пов'язано з якимось наявним об'єктом. Наприклад, визначення

```
Rectangle& refrec(rec);
```

зумовить появу альтернативного імені `refrec` для вже наявного прямокутника `rec`. Так само після виконання команди

```
Rectangle& refrecInSquare(square);
```

відсилка `refrecInSquare` стане альтернативним ім'ям для прямокутника, що є базовою частиною квадрата `square`.

Те саме стосується указників. Указником на базовий клас можна ідентифікувати як окремий об'єкт базового класу, так і базову частину об'єкта похідного класу:

Лістинг 7.26

```
Rectangle* ptrrec(&rec);
Rectangle* ptrrecInSquare(&square);
```

Головна відмінність опосередкованого іменування за допомогою відсилок або указників порівняно з прямим іменуванням полягає в тому, що один і той самий об'єкт без будь-яких перетворень можна іменувати іменами кількох підхожих типів. Крім власного типу об'єкта підхожим для

нього буде будь-який базовий тип в ієрархії успадкувань. Це називають доступом до одного й того самого об'єкта за допомогою різних інтерфейсів, кожен з яких задано відповідним класом. Можна сказати, наприклад, що відсилка `refrecInSquare`, як і указник `ptrrecInSquare`, використовують для доступу до квадрата `square` інтерфейс прямокутника, оскільки їх було задано як відсилку й указник на прямокутник.

Розглянемо простий ілюстративний приклад. Візьмемо клас банківських рахунків зі скромним набором атрибутів: номер рахунка, його власник і залишок на рахунку:

Лістинг 7.27

```
// Банківський рахунок
class Account {
private:
   static unsigned int _freeID; // Поточний вільний номер рахунка
   const unsigned int _number; // Власний номер рахунка
   int _balance; // Залишок коштів на рахунку
   string _owner; // Власник рахунка
public:
// Конструктор рахунка
   Account(const string owner, const unsigned int balance):
      _owner(owner), _balance(balance), _number(++_freeID) {}
   ~Account() {} // Деструктор рахунка
   string getName() const {return _owner;} // Селектор власника рахунка
   int getBalance() const {return _balance;} // Селектор балансу
   unsigned int getNumber() const {return _number;} // Селектор номера
                                                    // рахунка
   void setBalance(int balance) {_balance=balance;} // Модифікатор
                                                    // балансу
   int state() {return getBalance();} // Обчислення стану рахунка
};
```

Найважливішою для цього прикладу стане функція стану рахунка `state()`, яка в найпростішому випадку просто показує залишок коштів на рахунку, але в принципі може бути складнішою, нараховуючи преміальні різного типу.

Далі визначимо похідний клас ощадних рахунків. Він характеризуватиметься значенням запроваджених у банку ощадних відсотків `_savingRate` і розміром прибутку за відсотками `_interest`.

Лістинг 7.28

```
// Ощадний рахунок: спеціалізація банківського рахунка
class SavingAccount: public Account {
private:
```

```
   unsigned int _savingRate; // Норма банківського відсотка
   unsigned int _interest; // Прибуток за відсотками
public:
// Конструктор
   SavingAccount(const string owner, const unsigned int balance,
   unsigned int savingRate):
                Account(owner, balance), _savingRate(savingRate) {}
   ~SavingAccount() {}    // Деструктор
// Селектор банківського відсотка
   unsigned int getSavingRate() const {return _savingRate;}
// Модифікатор прибутку за відсотками
   void setInterest(const int interest) {_interest=interest;}
// Селектор прибутку за відсотками
   int getInterest() const {return _interest;}
// Обчислення стану рахунка
   int state() {
     setInterest(getBalance()*static_cast<double>(getSavingRate())/100);
     return Account::state()+getInterest();
   }
};
```

Тепер функція стану рахунка `state()` даватиме залишок, збільшений на розмір прибутку за відсотками. Відбулося заміщення функції `state()` із класу рахунків власною функцією. Однак при цьому сама функція базового класу нікуди не поділася: щоб звернутися до неї, достатньо лише зайти до відповідного простору. Це можна зробити за допомогою оператора визначення області дії, як показано в тексті програми (`Account::state()`), або ж використати імена з областю значень у базовому класі, як це буде зроблено далі.

І, нарешті, третій тип рахунка – бонусний ощадний. На ньому крім відсотків передбачено нарахування фіксованого бонусу.

Лістинг 7.29

```
// Бонусний ощадний рахунок: спеціалізація
// банківського рахунка
class BonusSavingAccount: public SavingAccount {
private:
   unsigned int _bonus; // Особливий бонус
public:
// Конструктор
   BonusSavingAccount(const string owner, int balance,
         const unsigned int savingRate, const unsigned int bonus):
      SavingAccount(owner, balance, savingRate), _bonus(bonus) {}
   ~BonusSavingAccount() {} // Деструктор
// Селектор бонусу
   unsigned int getBonus() const {return _bonus;}
```

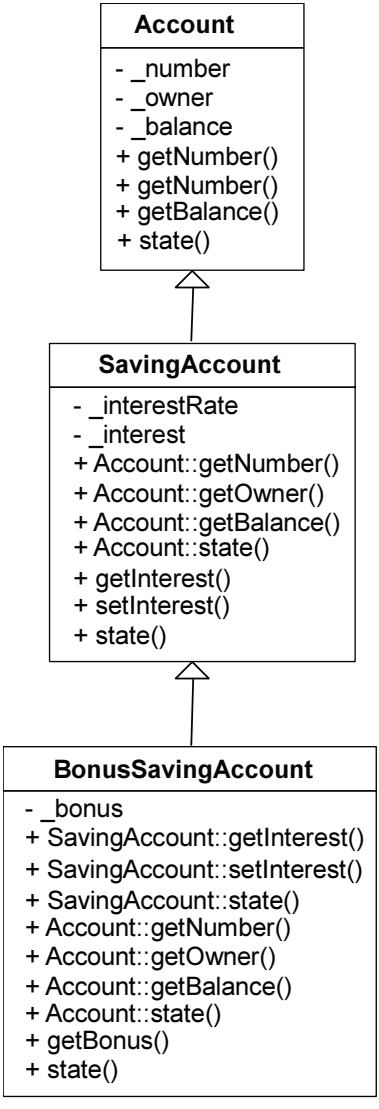

Рис. 7.4. Діаграма класів рахунків

```
// Обчислення стану рахунка
   int state()
      {return SavingAccount::state()+
         getBonus();}
};
```

Функція стану рахунка тепер додає до залишку та відсотків ще й бонус.

Конструктори та деструктори посідають особливе місце в ієрархії класів: вони не можуть бути заміщені. Під час кожного створення похідного об'єкта відпрацьовує весь ланцюжок конструкторів у напрямі, протилежному до успадкування. Це означає, що першим буде створено базовий об'єкт, потім – похідний об'єкт першого рівня, за ним – другого й так далі. Такий порядок виконання конструкторів цілком природний: адже базовий клас нічого не знає про свої похідні класи, тоді як похідний об'єкт може скористатися базовим уже безпосередньо в конструкторі.

Вправа 7.12

Доповніть конструктори та деструктори класів рахунків операціями виведення і простежте за послідовністю їх виконання. Ось приклад, що демонструє використання базового об'єкта в процесі створення похідного:

Лістинг 7.30

```
SavingAccount::
SavingAccount(const string owner,
   const unsigned int balance,
   const unsigned int savingRate):
   Account(owner, balance),
      _savingRate(savingRate) {
         cout<<"A saving account:"<<
         getNumber()<<endl;
}
```

Вправа 7.13

Доповніть клас ощадних рахунків SavingAccount нарахуванням за відкриття рахунка одноразового бонусу, що нараховується на баланс базового рахунка Account.

Вправа 7.14

Доповніть визначення класів Account деструкторами та проаналізуйте порядок їх викликів.

Структуру успадкувань зображено на рис. 7.4. Із нього видно, що ощадний рахунок має всі властивості звичайного і додає до них власні. Так само бонусний ощадний рахунок доповнює ощадний власним атрибутом і методом.

Розглянемо приклад, що демонструє способи вибору типу об'єкта в разі доступу до нього за допомогою указника. Тип об'єкта й інтерфейс доступу до нього визначаються під час компіляції програми.

Лістинг 7.31

```
// Тестова програма для банківських рахунків
int main () {
// 1. Створимо базовий рахунок
   Account ac("Ivanenko", 100);
   Account* pAcc=&ac;   // Непрямий доступ до базового рахунка
   cout<<pAcc->getNumber()<<':'<<pAcc->state()<<endl;
// 2. Створимо ощадний рахунок
   SavingAccount save("Petrenko", 200, 10);
   cout<<save.getNumber()<<':'<<save.state()<<endl;
// Доступ до ощадного рахунка за допомогою інтерфейсу базового рахунка
   pAcc=&save;
   cout<<pAcc->getNumber()<<':'<<pAcc->state()<<endl;
// 3. Створимо бонусний рахунок
   BonusSavingAccount bonus("Pavlenko", 300, 20, 100);
   cout<<bonus.getNumber()<<':'<<bonus.state()<<endl;
// Доступ до бонусного рахунка за допомогою інтерфейсу ощадного рахунка
   SavingAccount* pSave=&bonus;
   cout<<pSave->getNumber()<<':'<<pSave->state()<<endl;
// Доступ до бонусного рахунка за допомогою інтерфейсу базового рахунка
   pAcc=&bonus;
   cout<<pAcc->getNumber()<<':'<<pAcc->state()<<endl;
}
```

На першому кроці створено об'єкт базового класу. Установлений на нього указник такого ж типу не додає нічого нового.

На другому кроці буде створено об'єкт save похідного класу SavingAccount. Доступ до нього можна організувати за допомогою указника базового типу pAcc присвоєнням pAcc=&save, але безпосереднє виведення рахунка save і виведення за допомогою указника будуть принципово відрізнятися. У першому випадку відбудеться виклик функції save.state() похідного класу та нарахування відсотків, а в другому буде викликано функцію базового класу pAcc->state(), оскільки застосовано його інтерфейс.

Третій крок – створення бонусного рахунка – проаналізуйте самостійно.

Як висновок зазначимо, що указник базового класу можна встановити на об'єкт будь-якого похідного від нього класу. При цьому не відбувається перетворення типу, але указник має доступ лише до свого базового об'єкта, а всі похідні залишаються для нього недоступними.

Те саме можна сказати й про відсилки. Як приклад розглянемо функцію виведення рахунка

Лістинг 7.32

```
ostream& operator<<(ostream& os, const Account& ac) {
   os<<ac.getName()<<', '<<ac.getNumber()<<endl;
   return os;
}
```

Другий її параметр може набувати значення як базового рахунка, так і ощадного чи бонусного.

Лістинг 7.33

```
Account ac("Ivanenko", 100); // Базовий рахунок
cout<<ac; // Виведення базового рахунка
SavingAccount save("Petrenko", 200, 10); // Ощадний рахунок
cout<<save; // Виведення ощадного рахунка як базового
BonusSavingAccount bonus("Pavlenko", 300, 20, 100); // Бонусний рахунок
cout<<bonus; // Виведення бонусного рахунка як базового
```

Як і раніше, відсилці базового типу доступні лише засоби базового класу.

Властивість об'єктів бути доступними за допомогою різних інтерфейсів або, інакше кажучи, можливість використовувати об'єкти одного типу замість об'єктів іншого типу – чи не найважливіша риса об'єктно-орієнтованого програмування. Однак застосувати її слід особливо акуратно й обережно.

Тепер проаналізуємо проблеми, що виникають у разі застосування різних інтерфейсів до імен. Якщо відсилку при її визначенні пов'язують з уже наявним об'єктом, і тому для її значення не потрібно відводити окремої пам'яті, то визначення імені спричиняє створення нового

об'єкта. Природно, що визначення імені базового типу зумовлює виділення місця, потрібного саме для такого об'єкта. У наведеному далі прикладі один за іншим буде створено об'єкти базового та двох похідних типів. Після застосування до похідних об'єктів конструкторів копіювання базового типу вони перетворяться на об'єкти базових типів.

Лістинг 7.34

```
// 1. Створюємо базовий рахунок: ім'я ac має тип Account
   Account ac("Ivanenko", 100);
// 2. Створюємо ощадний рахунок: ім'я save має тип SavingAccount
   SavingAccount save("Petrenko", 200, 10);
// Забутливий копіювальний конструктор утрачає властивості
// ощадного рахунка: ім'я forgetful має тип Account і копіюється
// лише базова частина save
   Account forgetful(save);
// 3. Створюємо бонусний рахунок
   BonusSavingAccount bonus("Pavlenko", 300, 20, 100);
// Забутливе присвоєння втрачає властивості бонусного рахунка: ім'я
// forgetful має тип Account і копіюється лише базова частина bonus
   forgetful=bonus;
```

Копіювальний конструктор, застосований до об'єкта похідного типу під час створення базового, називають *забутливим* (forgetful). Так само поводяться копіювальні присвоєння імені базового класу об'єктові похідного класу, оскільки воно призводить до втрати всієї власної частини похідного об'єкта.

Проаналізуємо, як забутливість узгоджується з існуванням стандартних (згенерованих системою програмування) конструкторів і присвоєнь, а також програмованих (визначених розробником) класів.

Розглянемо можливі варіанти. Спочатку візьмемо клас `ShadowMakerNoAssign`, вільний від копіювального конструктора та програмованого присвоєння. Це означає, що до його об'єктів буде застосовано стандартні засоби. Наявність серед атрибутів класу указника дозволить стежити за коректністю поведінки його об'єктів. Об'єкт вважатимемо коректним, якщо його указник `_pChar` встановлено на власний атрибут `_myChar`.

Лістинг 7.35

```
// Базовий клас без власного програмованого присвоєння
class ShadowMakerNoAssign {
private:
   char _myChar; char* _pChar;
public:
   ShadowMakerNoAssign(const char c): _myChar(c), _pChar(&_myChar) {}
```

```
   const char& getChar() const {return _myChar;}
   void* const getPChar() const {return _pChar;}
};
```

Визначимо два об'єкти nshma й nshmb:

```
ShadowMakerNoAssign nshma('a'), nshmb('b');
```

Для спостереження за станом об'єктів цього класу запрограмуємо оператор виведення:

Лістинг 7.36

```
ostream& operator<<(ostream& os,
      const ShadowMakerNoAssign& shm) {
   os<<&shm<<':'<<shm.getChar()<<hex<<', '<<shm.getPChar()<<endl;
   return os;
}
```

Проаналізуємо конструктор класу ShadowMakerNoAssign. Як бачимо, він установлює указник, заданий у другому атрибуті, на значення свого першого атрибута. Результат виконання конструкторів зображено на рис. 7.5.

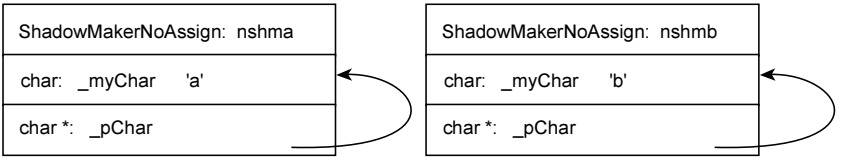

Рис. 7.5. Діаграма об'єктів nshma й nshmb до присвоєння

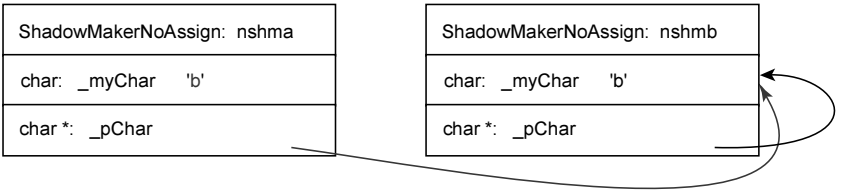

Рис. 7.6. Діаграма об'єктів nshma й nshmb після присвоєння: некоректний результат

Виконаємо присвоєння

```
nshma=nshmb;
```

Його результат, власне, не зовсім коректний, як бачимо з рис. 7.6. Символьний атрибут _myChar в об'єкті nshma присвоєно коректно за допо-

могою почленного присвоєння, проте його указник _pChar показує тепер на чужий атрибут замість власного.

Тепер визначимо клас ShadowMakerDerivedNoAssign, похідний від класу ShadowMakerNoAssign, з тією ж властивістю атрибутів і знову ж із виконаним за замочуванням присвоєнням:

// Похідний клас без власного програмованого присвоєння **Лістинг 7.37**

```
class ShadowMakerDerivedNoAssign: public ShadowMakerNoAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedNoAssign(const char c, const int i):
      ShadowMakerNoAssign(c), _myInt(i), _pInt(&_myInt) {}
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Розглянемо тепер два об'єкти похідного класу

```
ShadowMakerDerivedNoAssign dnshma('a', 1), dnshmb('b', 2);
```

зображені на рис. 7.7.

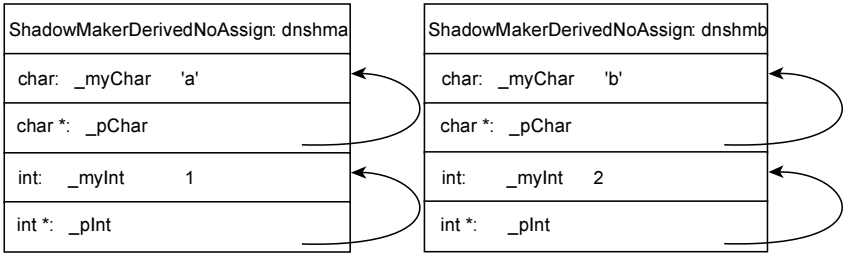

Рис. 7.7. Діаграма об'єктів dnshma й dnshmb до присвоєння

Іще раз виконаємо присвоєння – тепер похідних об'єктів:

```
dnshma=dnshmb;
```

Як бачимо з рис. 7.8, воно торкнулося як базової, так і похідної частини об'єкта некоректно, як і в попередньому випадку для базового класу.

Вправа 7.15

Задайте оператор виведення для похідного класу та протестуйте виконання присвоєння.

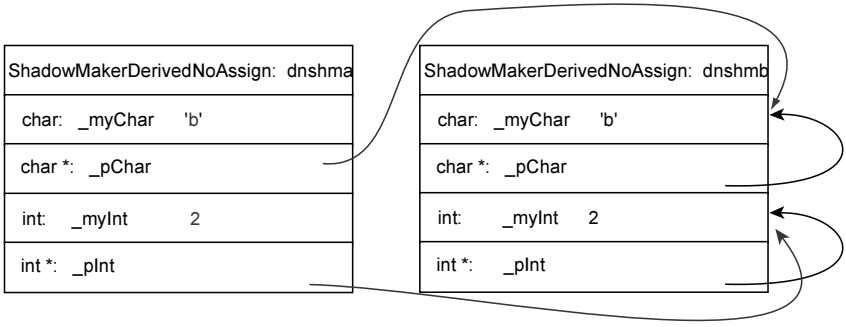

Рис. 7.8. Діаграма об'єктів `dnshma` й `dnshmb` після присвоєння: некоректний результат

Власне, поки що тут немає нічого нового: зрозуміло, що почленне присвоєння указників змістовно некоректне. Подивимося, як поводитимуться схожі класи, коли в них задано програмоване присвоєння. Розглянемо базовий клас `ShadowMakerAssign`, наділений власним присвоєнням:

Лістинг 7.38

```
// Базовий клас із власним програмованим присвоєнням
class ShadowMakerAssign {
private:
   char _myChar; char* _pChar;
public:
   ShadowMakerAssign(const char c): _myChar(c), _pChar(&_myChar) {}
// Програмоване присвоєння
   ShadowMakerAssign& operator= (const ShadowMakerAssign& tar) {
      _myChar=tar._myChar;
      return *this;
   }
   const char& getChar() const {return _myChar;}
   void* const getPChar() const {return _pChar;}
};
```

Візьмемо два його об'єкти й виконаємо присвоєння:

Лістинг 7.39

```
ShadowMakerAssign shma('a'), shmb('b');
shma=shmb;
```

Після цього матимемо ситуацію, проілюстровану на рис. 7.9. Як бачимо, не лише змінилося значення символьного атрибута `_myChar` в об'єкті `shma`, але й указник `_pChar` указує на атрибут у власному, а не чужому об'єкті.

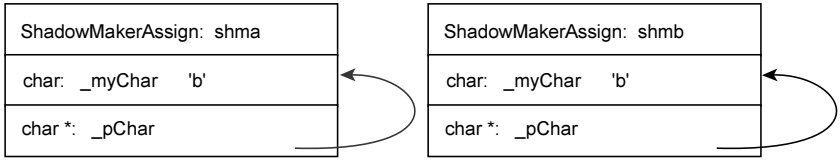

Рис. 7.9. Діаграма об'єктів `shma` й `shmb` після присвоєння: коректний результат

Похідний клас `ShadowMakerDerivedNoAssign` поки що залишимо без програмованого присвоєння.

Лістинг 7.40

```
// Похідний від класу з програмованим присвоєнням
// клас без власного програмованого присвоєння
class ShadowMakerDerivedNoAssign: public ShadowMakerAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedNoAssign(const char c, const int i):
      ShadowMakerAssign(c), _myInt(i), _pInt(&_myInt) {}
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Тепер присвоєння в похідному класі виконуватиметься за тим самим принципом, але з іншим результатом, порівняно з рис. 7.8: спочатку задане за умовчанням присвоєння в похідному класі, а потім – програмоване в базовому.

Лістинг 7.41

```
ShadowMakerDerivedNoAssign dnshma('a', 1), dnshmb('b', 2);
dnshma=dnshmb;
```

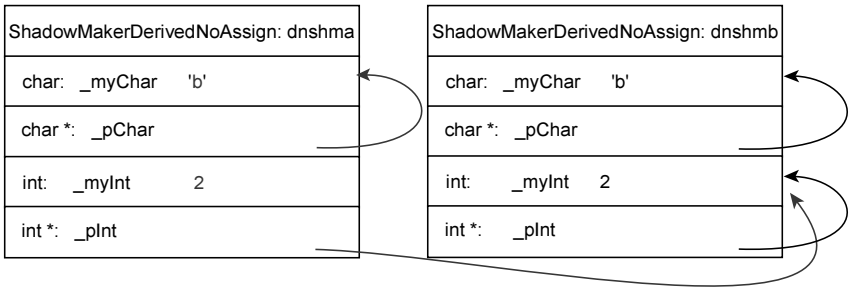

Рис. 7.10. Діаграма об'єктів `dnshma` й `dnshmb` після присвоєння: коректний результат в базовому об'єкті та некоректний в похідному

Результат бачимо на рис. 7.10. Базовий об'єкт присвоєно коректно, а похідну частину – усе ще за правилом стандартного, а тому почленного присвоєння.

Цікаво, що просто визначивши програмоване присвоєння в похідному класі, проблемі не зарадити. Якщо програмоване присвоєння задано в похідному класі, то відбудеться заміщення присвоєння з базового класу присвоєнням похідного: усю відповідальність буде перекладено на нього, і справа не дійде до присвоєння в базовому класі.

Лістинг 7.42

```
// Похідний клас із власним присвоєнням
class ShadowMakerDerivedAssign: public ShadowMakerAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedAssign(const char c, const int i):
      ShadowMakerAssign(c), _myInt(i), _pInt(&_myInt) {}
// Програмоване присвоєння
   ShadowMakerDerivedAssign& operator=
         (const ShadowMakerDerivedAssign& tar) {
      cout<<"derived assignment"<<endl; _myInt=tar._myInt;
      return *this;
   }
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Знову розглянемо схожий приклад:

Лістинг 7.43

```
ShadowMakerDerivedAssign dnshma('a', 1), dnshmb('b', 2);
dnshma=dnshmb;
```

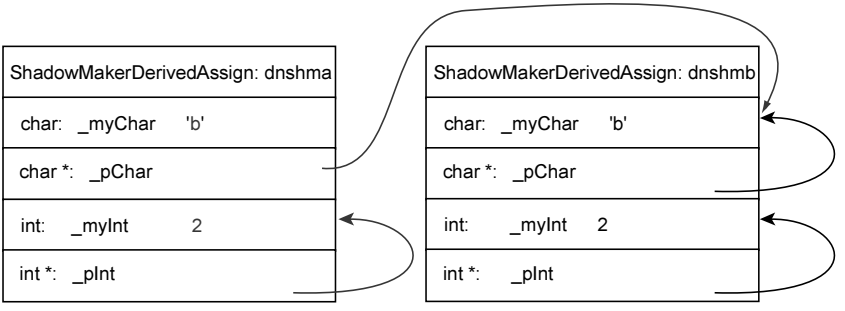

Рис. 7.11. Діаграма об'єктів `dnshma` *й* `dnshmb` *після присвоєння: некоректний результат в базовому об'єкті та коректний в похідному*

Одержаний результат відображено на рис. 7.11. Похідний клас використовує власне програмоване присвоєння, яке «затінило» програмоване присвоєння з базового класу. Присвоєння базового об'єкта знову відбулось як задане за умовчанням.

Нарешті, запишемо повне розв'язання проблеми, у якому задано коректне присвоєння як у базовому, так і в похідному класах:

Лістинг 7.44

```
// Узгоджене присвоєння похідного та базового класів
class ShadowMakerDerivedAssign: public ShadowMakerAssign {
private:
   int _myInt; int* _pInt;
public:
   ShadowMakerDerivedAssign(const char c, const int i):
      ShadowMakerAssign(c), _myInt(i), _pInt(&_myInt) {}
// Програмоване присвоєння похідного класу
   ShadowMakerDerivedAssign& operator=
         (const ShadowMakerDerivedAssign& tar) {
// Виклик програмованого присвоєння базового класу
      this->ShadowMakerAssign::operator=(tar);
      cout<<"derived assignment"<<endl; _myInt=tar._myInt;
      return *this;
   }
   const int& getInt() const {return _myInt;}
   void* const getPInt() const {return _pInt;}
};
```

Тепер у базовому класі явно викликатиметься потрібне присвоєння, задане в ньому сигнатурою

```
ShadowMakerAssign& ShadowMakerAssign::operator=
   (const ShadowMakerAssign& tar);
```

Отримуємо правильний результат (рис. 7.12).

Отже, стає повністю зрозумілою природа забутливого присвоєння: виконуючи присвоєння значення імені об'єкта базового класу, буде взято присвоєння, наявне у базовому класі, навіть якщо джерелом присвоєнням служить об'єкт похідного класу. У цьому випадку частину об'єкта, що не входить до базового класу, буде втрачено (забуто). За місцем призначення до лівої частини присвоєння потрапить лише об'єкт базового класу. Така поведінка цілком зрозуміла: адже за місцем розміщення значення імені об'єкта базового класу не передбачено пам'яті для зберігання атрибутів його похідних класів.

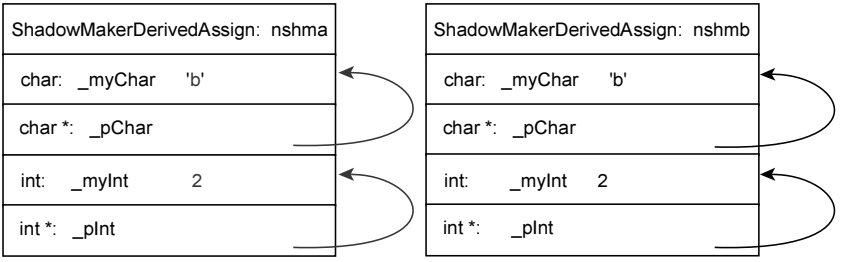

Рис. 7.12. Діаграма об'єктів `dnshma` *й* `dnshmb` *після присвоєння: повністю коректний результат в базовому і похідному об'єктах*

Як висновок скажемо, що, використовуючи ієрархію класів, слід або ***повністю заборонити присвоєння, поміщаючи його в закритій частині, або програмувати в кожному похідному класі всю ієрархію присвоєнь***.

7.4. Уточнення привілеїв доступу до членів базового класу

Поділ класу на відкриту й закриту частини не надає похідному класу жодних привілеїв стосовно доступу до базового. Проте за своїм становищем похідні класи якогось базового класу мали б відрізнятися від усіх інших класів, які його використовують. Бажання наділити похідний клас особливими правами доступу до певної частини базового класу зумовило появу ще однієї (окрім відкритої та закритої) частини класу, яку назвали *захищеною* (protected).

Повернімося до класу прямокутників і змінимо статус його атрибутів із закритого на захищений.

Лістинг 7.45

```
// Тип довжини сторін
typedef double length_t;
// Клас прямокутників із захищеними сторонами
class Rectangle {
// Захищена частина класу
protected:
   length_t _height; // Висота
   length_t _width; // Ширина
public:
   Rectangle (const length_t heigh, const length_t width); // Конструктор
   ~Rectangle(); // Деструктор
```

```
   Rectangle (const Rectangle&); // Копіювальний конструктор
   Rectangle& operator=(const Rectangle&); // Присвоєння
// Селектори сторін
   length_t getA() const;
   length_t getB() const;
// Модифікатори сторін
   Rectangle& setA(const length_t); Rectangle& setB(const length_t);
// Функції обчислення характеристик:
   length_t area() const; // площі
   length_t diagonal() const; // діагоналі
};
// Операція виведення прямокутника
ostream& operator<<(ostream&, const Rectangle&);
```

Захищена частина недоступна нікому ззовні класу, крім його похідних класів. Тепер можна переписати реалізацію методів квадрата, скориставшись замість селекторів прямим доступом до атрибутів прямокутника, що не так вже й добре. Саме визначення класу квадратів залишається без змін таким, як у лістингу 7.10.

Лістинг 7.46

```
// Реалізація копіювального конструктора квадрата
DerivedSquare::DerivedSquare (const DerivedSquare& source):
// Замість Rectangle(source.getA(), source.getB())
   Rectangle(source._height, source._width) {}
// Реалізація функції обчислення площі квадрата
length_t DerivedSquare::area() const {
   return _height*_height; // Замість return getA()*getA();
}
// Реалізація функції обчислення діагоналі квадрата
const double sqrt2=sqrt(2.); // 1.4142135623730950;
length_t DerivedSquare::diagonal() const {
   return _height*sqrt2; // Замість return getA()*sqrt2;
}
```

Ззовні похідного класу захищена частина базового поводиться як закрита. Тому утиліти обчислення площі та діагоналі залишаються незмінними. Вони, як і раніше, використовуватимуть селектори. Для перевірки транзитивності доступу до захищеної частини класу вздовж ланцюжка успадкувань достатньо пересвідчитися в тому, що клас, похідний від класу квадратів, зберігає за собою права доступу до захищеної частини прямокутників. Це можна зробити, скажімо, за допомогою такого простого класу:

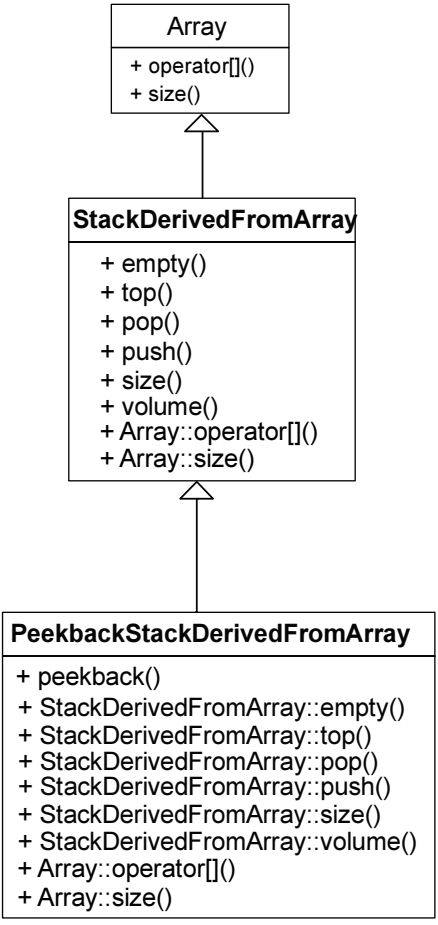

Рис. 7.13. Діаграма успадкувань стеку з підгляданням (відкритий варіант)

```
// Похідний клас                    Лістинг 7.47
// другого рівня від прямокутника
class ProtectionChecker:
      public DerivedSquare {
// Захищена частина класу
// прямокутників залишається
// доступною
   length_t getA()
      {return _width;}
};
```

Але насправді захищена частина класу – серйозна загроза інкапсуляції, оскільки дозволяє неконтрольоване розширення кола класів із винятковими правами доступу до неї: адже кожен похідний клас, у принципі, має право відкрити захищену частину свого базового класу.

Тепер проаналізуємо, як відкрите успадкування можна застосувати до практичної розробки ієрархії класів, знову розглянувши, зокрема, задачу підглядання до стека з підрозділу 7.1, розв'язання якої засобами композиції виявилося недосконалим. Замінимо ієрархію об'єктів стека та масиву ієрархією відповідних класів, як це зображено на рис. 7.13.

Далі наведено визначення стека, похідного від масиву (порівняйте його з текстом лістингу 7.1). Зверніть увагу на особливості звертань до масиву.

```
// Стек як спеціалізація масиву або гібрид масиву зі стеком  Лістинг 7.48
template <class Elem>
class StackDerivedFromArray: public Array<Elem> {
private:
```

```
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
// Більше немає масиву як атрибута
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}   // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}    // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до функції базового класу, адаптуючи її ім'я
      return Array<Elem>::size();
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника класу, похідного від
// масиву, застосовуємо його операцію індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
template <class Elem>
const size_t StackDerivedFromArray<Elem>::_bos=-1;
```

Проаналізуємо деякі важливі особливості нової організації ієрархії.

У разі застосування ієрархії об'єктів масив належав до закритої частини стека. Коли стек надавав своїм клієнтам доступ до масиву, було за-

стосовано делегування. Так, у лістингу 7.1 функції методу `volume()` було делеговано масиву, де їх забезпечував метод `size()`.

Ієрархія класів, реалізована у вигляді відкритого успадкування, надає клієнтам стека повний доступ до відкритої частини масиву, зокрема до функції `size()`. Тому делегування стає непотрібним. Утім стек уже містить іншу функцію з цим самим іменем. У такому разі говорять, що функція похідного класу *затіняє* своїм клієнтам однойменну функцію базового класу. Звісно, до неї можна звернутися, використовуючи відповідний кваліфікатор, як це зроблено в наступному прикладі:

Лістинг 7.49

```
// Визначення стека
StackDerivedFromArray<char> s(100);
cout<<s.size()<<endl; // Виведення розміру стека: буде показано 0
// Виведення розміру масиву: буде показано 100
cout<<s.Array<char>::size()<<endl;
```

Інша можливість полягає в адаптації наявного базового класу під установлений інтерфейс похідного. Тому метод `volume()` похідного класу стеків став адаптером для методу `size()` класу масивів:

```
// Виведення розміру масиву за допомогою адаптера: буде показано 100
cout<<s.volume()<<endl;
```

Крім делегування й адаптації існує ще кілька інших важливих прийомів, відомих під загальною назвою *проектних взірців*, або *патернів* (design pattern). Рекомендуємо ознайомитися з ними самостійно [14].

Завдяки відкритому успадкуванню операція індексування стала тепер успадкованою від масиву, а тому її можна застосовувати до самого стека. У тілі стека це потребує явного звертання до указника поточного об'єкта, наприклад `(*this)[_top]`.

Просте на перший погляд розв'язання проблеми організації взаємодії класу стеків із класом масивів за допомогою ієрархії класів має серйозні наслідки. Так, операція індексування стала тепер доступною також і клієнтам стека, що повністю порушує дисципліну зберігання й обробки даних стеком.

Лістинг 7.50

```
// Некоректна поведінка стека, успадкованого від масиву
StackContainingArray<char> s(100);
// Беремо з порожнього стека
cout<<s[0]<<endl;
// Кладемо безсистемно
s[10]='a'; s[20]='b';
```

```
// Беремо не звідти, куди клали
cout<<s[15]<<endl;
```

Поміркуймо, як зарадити цьому, використовуючи відомі нам засоби. Можна зробити так, щоб стек закривав доступ до операції індексування, але, як побачимо, це лише половинчасте розв'язання проблеми.

Лістинг 7.51

```
// Стек з умовно закритим доступом до операції індексування
template <class Elem>
class StackDerivedFromArray: public Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem>::operator[]; // Закриття доступу до операції індексування
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
// Масив, ініціалізований як базовий клас
      Array<Elem>(size) {}
// Деструктор
   ~StackDerivedFromArray() {}
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до функції базового класу, адаптуючи її ім'я
      return Array<Elem>::size();
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника класу, похідного від
// масиву, застосовуємо його операцію індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
```

```
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
```

Утім цей розв'язок, як і раніше, неповний, бо клієнтові стека відмовлено лише в безпосередньому використанні оператора індексування, але можливий кваліфікований доступ до цього оператора за допомогою оператора визначення області дії. Некоректне застосування стека, приклад якого було наведено в лістингу 7.50, усе ще можливе, якщо його переписати, наприклад, так:

Лістинг 7.52

```
// Некоректна поведінка стека, успадкованого від масиву,
// у разі закритого доступу до індексування
StackContainingArray<char> s(100);
// Беремо з порожнього стека
cout<<s.Array<char>::operator[](0)<<endl;
// Кладемо безсистемно
s.Array<char>::operator[](10)='a'; s.Array<char>::operator[](20)='b';
// Беремо не звідти, куди клали
cout<<s.Array<char>::operator[](15)<<endl;
```

Зверніть увагу на те, як записано оператор індексування. Варіант запису `s[i]` стає тепер недозволеним. Потрібно застосувати кваліфікатор `Array<char>::`, для чого в розгляданому контексті доведеться вжити повну, а не скорочену форму оператора індексування **`operator[]`**.

Успадкування стека від масиву невдале рішення. Залишається лише сподіватися, що клієнт дотримуватиме дисципліни коректного використання стека.

Вправа 7.16

Складіть тестову програму та протестуйте клас `StackDerivedFromArray`.

Подивимося, які можливості надає стек, похідний від масиву, з погляду визначення на його основі інших класів, наприклад уже відомого нам стека з підгляданням. Тепер можна за допомогою успадкування просто додати підглядання до наявної функціональності стека. Справді, кваліфікований доступ до індексування доступний будь-якому клієнтові стека, зокрема похідному від нього класу.

```
// Стек із підглядянням, похідний від стека, який                Лістинг 7.53
// успадковано від масиву: доступ до індексування опосередковано
// через операцію визначення області дії
template <class Elem>
class PeekbackStackDerivedFromArray: public StackDerivedFromArray<Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit PeekbackStackDerivedFromArray (const size_t size):
           StackDerivedFromArray(size) {}
// Деструктор
   ~PeekbackStackDerivedFromArray() {}
// Функція підглядання
   bool peekback(const size_t i, Elem& elem) const {
      if (i>=size()) {elem=top();return false;}
      elem=this->Array<Elem>::operator[](i);
      return true;
   }
};
```

Варто звернути увагу на особливу форму виклику операції індексування у функції `peekback()`:

```
elem=this->Array<Elem>::operator[](i);
```

Вона зумовлена тим, що клас стеків закрив доступ до операції індексування всім, зокрема своїм похідним класам. Тому варіант запису `(*this)[i]`, як у реалізації операцій над стеком із лістингу 7.48, знову став недозволеним, і потрібно застосувати кваліфікатор `Array<Elem>::` до повної форми оператора індексування **`operator`**`[]`. Як бачимо, можливість кваліфікованого доступу залишилась, і її застосування виправдане.

Вправа 7.17

Складіть тестову програму та протестуйте клас `PeekbackStackDerivedFromArray`.

Вправа 7.18

Визначте й реалізуйте операцію виведення вмісту стека з підглядянням.

Спробуємо точніше призначити права доступу до операції індексування. Закриємо її від клієнтів стека, але зробимо доступною для похідних класів, адже саме для цього й призначено захищену область класу.

```
// Стек з операцією індексування, умовно закритою                Лістинг 7.54
// для клієнтів, але доступною для похідних класів
```

```
template <class Elem>
class StackDerivedFromArray: public Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
protected:
   Array<Elem>::operator[]; // Обмеження доступу до операції індексування
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}   // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до функції базового класу, адаптуючи її ім'я
      return Array<Elem>::size();
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника класу, похідного від
// масиву, застосовуємо його операцію індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
```

Тепер у стеку з підгляданням не має бути кваліфікатора, а індексування має виконуватися більш звичним способом `(*this)[i]`:

Лістинг 7.55

```
// Стек із підглядянням, похідний від стека, який
// успадковано від масиву: безпосередній доступ до індексування
template <class Elem>
class PeekbackStackDerivedFromArray: public StackDerivedFromArray<Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit PeekbackStackDerivedFromArray
       (const size_t size): StackDerivedFromArraysize) {}
// Деструктор
   ~PeekbackStackDerivedFromArray() {}
// Функція підглядання
   bool peekback(const size_t i, Elem& elem) const {
      if (i>=size()) {elem=top(); return false;}
   elem=(*this)[i];
   return true;
   }
};
```

Вправа 7.19

Означте й реалізуйте операції порівняння стеків із підгляданням на рівність і нерівність їхнього вмісту.

Знайдений розв'язок кращий за попередній, але все одно незадовільний. Перевагою слід вважати те, що клієнти стека втратили можливість доступу (навіть кваліфікованого) до операції індексування, невластивої для стека. Але «підводні камені» все ще залишилися. Проаналізуємо їх.

Передусім, помилковою була сама концепція відкритого успадкування стека від масиву. У відкритого успадкування має бути лише одне призначення, яке залежно від напрямку руху слід розуміти як спеціалізацію базового класу в похідному чи як узагальнення похідного класу до базового. У такому розумінні ієрархія «стек – стек із підгляданням» цілком коректна. Справді, стекові з підгляданням властива вся поведінка стека, до того ж він має ще один власний метод, у чому й полягає його спеціалізація.

Якщо ж порівняти стек із масивом, то нічого подібного ми не виявимо. Стек не тільки не схожий на масив, але й може існувати цілком незалежно від нього (як, наприклад, стек на базі списку), залишаючись при цьому повноцінним стеком. Оскільки стек у жодному розумінні не слугує прикладом масиву, він не має успадковувати його поведінки. Тому застосування відкритого успадкування стека від масиву слід вважати невиправданим, навіть коли можна закрити доступ до небажаної поведінки.

Окрім того, усе ще залишається можливість відкрити приховане. Виявивши певну кмітливість, можна примусити стек поводитися так, ніби він є масивом, що, узагалі кажучи, неприпустимо.

Лістинг 7.56

```
// «Несерйозний» стек
// Відкритий доступ до індексування
template <class Elem>
class NaughtyStackDerivedFromArray:
   public StackDerivedFromArray<Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit NaughtyStackDerivedFromArray
       (const size_t size): StackDerivedFromArray<Elem> (size) {}
// Деструктор
   ~NaughtyStackDerivedFromArray () {}
// Відкривання доступу до операції індексування
   Array<Elem>::operator[];
};
```

Тепер над «несерйозним» стеком можна повністю виконати експеримент із несанкціонованого доступу з лістингу 7.52 за таким зразком:

Лістинг 7.57

```
// Несподівана поведінка «несерйозного» стека
NaughtyStackDerivedFromArray<char> s(100);
s.push('A'); s.push('B'); s.push('C');
cout<<s[0]<<endl; cout<<s[1]<<endl; cout<<s[2]<<endl;
```

Зроблений аналіз застосування відкритого успадкування приведе до пошуків інших способів організації успадкувань, які ми почнемо із захищеного успадкування.

7.5. Захищене успадкування

Маніпуляції з правами доступу для відкритого успадкування, власне, свідчать про невдалі спроби застосувати його в неадекватній ситуації. Ми виходили з того, що відкрите успадкування дає змогу віднести об'єкти похідного класу до типу, заданого базовим класом. Так, квадрат і прямокутник мають спільний тип прямокутника, а стек і стек із підгляданням обидва належать до стеків. Інша справа – масив і стек. Стек не має й не може мати поведінки масиву. Його відношення з масивом принципово інші: стек використовує масив для власної реалізації, хоча для цього придатна й інша структура даних, наприклад список. Від цього тип стека не змінився б: він би й надалі залишався стеком у разі дотримання принципу доступу «останнім прийшов – першим обслужили».

Для взаємодії стека з масивом ми вже використовували композицію й агрегацію. Кожна з них була б цілком прийнятним типом ієрархії, якби не потреба передати класам, похідним від стека, права доступу до методів масиву, чого нам не вдалося забезпечити.

Розглянемо розв'язання цієї проблеми згідно з концепцією *захищеного успадкування* (protected inheritance). Таке успадкування перетворює статус функцій, відкритих у базовому класі, на захищений у похідному. У разі застосовування цього виду успадкування похідним класам буде передано права доступу як до відкритої, так і до захищеної частин базового класу. Проте для клієнтів похідних класів доступ до базового класу закрито. Порівнявши наведене далі визначення стека з текстом лістингу 7.51, побачимо, що змінився тип успадкування та зникла потреба закривати доступ до операції індексування.

Лістинг 7.58

```
// Стек закриває доступ до масиву своїм клієнтам,
// але відкриває доступ до нього своїм похідним класам
template <class Elem>
class StackDerivedFromArray: protected Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}  // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Тепер звертаємося до функції базового класу,
// використовуючи оператор визначення області дії
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
// Звертаємося до методу базового класу size(),
// делегуючи йому функції методу volume()
      return Array<Elem>::size();
   }
```

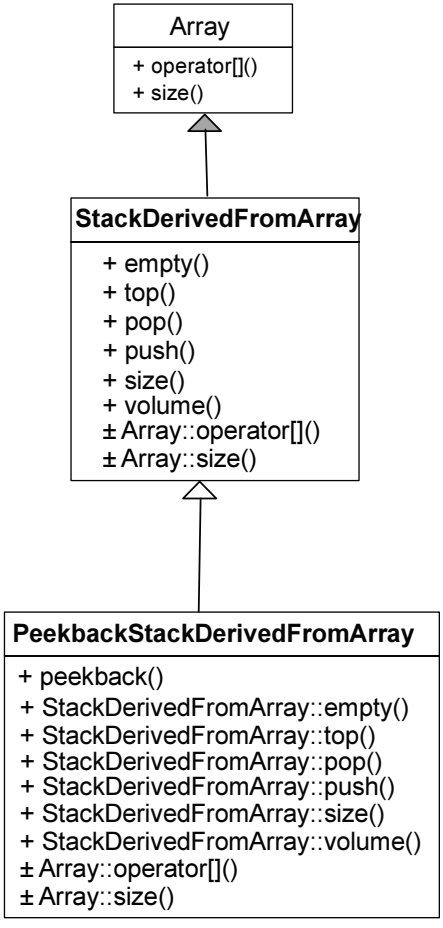

Рис. 7.14. Діаграма успадкувань стеку з підгляданням (захищений варіант)

```
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
// До стека як представника
//класу, похідного від масиву,
// застосовуємо його операцію
//індексування
      return (*this)[_top];
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {
      (*this)[++_top]=value;
// Знову застосовуємо операцію
// індексування
   }
};
```

Залишивши ієрархію стека та стека з підгляданням без змін у вигляді відкритого успадкування, що, як бачимо, адекватно відтворює суть зв'язку між ними, одержимо трирівневу ієрархію, зображену на рис. 7.14. Захищене успадкування тут позначено сірою стрілкою. Видно, що відкриті методи класу масивів `Array` перетворюються на захищені методи класу стеків `StackDerivedFromArray`, тому стають доступними (знову ж як захищені методи) похідному класу стеків із підгляданням `PeekbackStackDerivedFromArray`.

Останнє визначення стека з підгляданням із попереднього підрозділу залишається без змін. Як і раніше, це повноцінний стек із додатковою функціональністю – методом підглядання.

Вправа 7.20

Складіть тестову програму та виконайте тестування класів `StackDerivedFromArray` і `PeekbackStackDerivedFromArray`.

Тепер стає неможливим некоректне використання клієнтами похідних класів операції індексування. Спроби добратися до елемента масиву як зі звичайного стека, так і зі стека з підгляданням призводять до синтаксичних помилок:

Лістинг 7.59

```
StackDerivedFromArray<char> stachar(100);
stachar.Array<char>::operator[](30)='A';   // ERROR!
PeekbackStackDerivedFromArray<char> pbstachar(100);
pbstachar.Array<char>::operator[](30)='A'; // ERROR!
```

Наведене розв'язання цієї проблеми у формі захищеного успадкування має суттєву перевагу порівняно зі спробами закрити доступ у разі відкритого успадкування. Вона полягає в послідовному дотриманні відповідних прав доступу до об'єктів: навіть застосування операції визначення області дії не надає клієнтам доступу до операції індексування.

Вправа 7.21

Задайте ієрархію з трьох класів `Father`, `Son` і `Grandson`, пов'язавши `Father` і `Son` захищеним, а `Son` і `Grandson` – відкритим успадкуванням. Означте один відкритий

```
void Father::open(void);
```

й один захищений метод

```
void Father::protect(void);
```

у базовому класі. Проаналізуйте можливість доступу до них як у кожному з похідних класів, так і з боку клієнтів цих класів. Відкрийте доступ до захищеного методу почергово в усіх похідних класах і простежте за доступом клієнтів до методу, що став відкритим.

7.6. Закрите успадкування

Перш ніж розглянути ще один тип успадкувань, порівняємо два попередні. Відкрите успадкування наділяє всі похідні класи типом базового класу, а тому відкриває доступ до відкритої частини базового класу клієнтам усіх його похідних класів. Захищене успадкування не поширює тип базового класу на похідні, закриваючи від їхніх клієнтів відкриту частину базового класу, але похідні класи далі передають доступ до відкритої частини базового класу всім класам, одержаним із них за допомогою відкритого успадкування.

Тепер похідному класові залишилося закрити базовий клас не лише від клієнтів, але й від похідних від нього самого класів. Цю функцію виконує *закрите успадкування* (private inheritance), яке перетворює відкриту частину базового класу на закриту частину похідного.

Задамо стек як результат закритого успадкування від масиву:

Лістинг 7.60

```
// Стек закриває доступ до масиву як своїм клієнтам,
// так і похідним класам
template <class Elem>
class StackDerivedFromArray: private Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {} // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {} // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
      return _top==Array<Elem>::size()-1; // Тепер звертаємося до функції
         // базового класу, використовуючи оператор визначення області дії
   }
// Видати поточний розмір стека
   size_t size() const {
      return _top+1;
   }
// Видати ємність контейнера
   size_t volume() const {
      return Array<Elem>::size(); // Звертаємося до методу базового
                // класу size(), делегуючи йому функції методу volume()
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
      return (*this)[_top]; // До стека як представника класу,
       // похідного від масиву, застосовуємо його операцію індексування
   }
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
```

```
   void push(const Elem& value) {
      (*this)[++_top]=value; // Знову застосовуємо операцію індексування
   }
};
```

Якщо тепер знову спробувати задати стек із підгляданням як клас, похідний від класу стеків, то не буде доступу до масиву, а отже, і до його операції індексування. Щоб уникнути цього, відкриємо похідному класу доступ до операції індексування вже відомим способом – уточненням прав доступу через захищену частину стека:

Лістинг 7.61

```
// Стек закриває доступ до масиву як своїм клієнтам,
// так і похідним класам, але похідним класам відкриваємо доступ
// до операції індексування, помістивши її в захищеній частині
template <class Elem>
class StackDerivedFromArray: private Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
protected:
// Відкриваємо похідним класам доступ до операції індексування
   Array<Elem>::operator[];
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}  // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
      return _top==Array<Elem>::size()-1; // Тепер звертаємося до
// функції базового класу, використовуючи оператор визначення області дії
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
      return Array<Elem>::size(); // Звертаємося до методу базового
                 // класу size(), делегуючи йому функції методу volume()
   }
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {
```

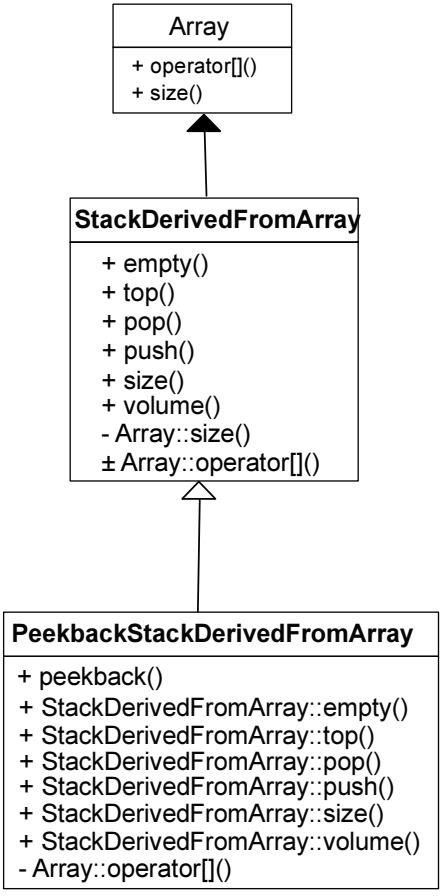

Рис. 7.15. Діаграма успадкувань стеку з підгляданням (закритий варіант)

```
        return (*this)[_top];
// До стека як представника
// класу, похідного від масиву,
// застосовуємо його операцію
// індексування
    }
// Виштовхнути верхівку стека
    void pop() {_top--;}
// Помістити до стека
    void push(const Elem& value) {
        (*this)[++_top]=value;
// Знову застосовуємо операцію
// індексування
    }
};
```

Визначення стека з підгляданням знову ж залишається без змін. Одержану ієрархію показано на рис. 7.15. Захищене успадкування позначено чорною стрілкою. Функція `size()` класу масивів стає закритою функцією класу стеків, тоді як оператору індексування в ньому надано статус захищеного.

Якщо знову утворити стек із підгляданням за допомогою відкритого успадкування, то операція індексування стане в ньому закритою. Це означає, що методи стека з підгляданням можуть використовувати її для власної реалізації, але ця операція закрита від клієнтів стека з підгляданням.

Підіб'ємо підсумки та спробуємо охарактеризувати виявлені властивості різних видів успадкувань. Для організації ієрархії стека відносно масиву ми почергово спробували застосувати кожен із трьох видів успадкування: відкрите, захищене та закрите. Розглянемо їхні переваги та вади.

Відкрите успадкування не відповідає характеру залежності стека від масиву, оскільки в разі його застосування стек перетворюється на різновид масиву. Доводиться штучно забороняти клієнтам стека доступ до ма-

сиву, що, власне, зводить нанівець принципи відкритого успадкування. До того ж класи, похідні від стека з підгляданням, усе одно можуть відкрити своїм клієнтам доступ до функцій масиву.

Захищене успадкування закриває масив від клієнтів стека, що усуває проблему неадекватної поведінки стека, властиву відкритому успадкуванню. Тепер стек сам використовує масив, проте не надає цієї послуги своїм клієнтам. Окрім того, захищене успадкування відкриває похідним класам доступ до масиву, що дає змогу легко реалізувати стек із підгляданням як похідний від масиву клас. Щоправда, класи, похідні від стека з підгляданням, можуть знову ж таки відкрити доступ до масиву.

Закрите успадкування, можливо, найточніше відповідає ідеї залежності стека від масиву. Стек використовує масив для власної реалізації та закриває його від своїх клієнтів і похідних класів. Якщо потрібно надати власному похідному класу доступ до масиву, стек може уточнити права доступу до нього, замінивши його статус із закритого на захищений. Щоправда, тоді знову будь-який похідний клас зможе відкрити своїм клієнтам доступ до операції індексування, і це порушить принципи використання стека.

Повністю адекватним розв'язанням задачі організації ієрархії стека виявилося відкрите успадкування стека з підгляданням від стека. Справді, незалежно від способу організації самого стека відкрите успадкування дає змогу кожному клієнтові стека з підгляданням виконувати всі дії, задані над стеками, а також функцію підглядання. У такому разі очевидно, що стек із підгляданням являє собою спеціалізацію стека, а тому успадковує його тип, тобто сигнатуру його методів, і поведінку – їх реалізацію.

Успадкування типу зазвичай називають *успадкуванням інтерфейсу*, а успадкування поведінки – *успадкуванням реалізації*. Об'єднавши їх, можна говорити про *повне успадкування*. Зазначимо, що для похідних класів можна як скасувати успадкування інтерфейсу, закривши доступ до будь-якого з методів, так і відмовитися від успадкованої реалізації, замінивши її власною. Усе це потребує дотримання чіткої дисципліни в організації ієрархій.

Вправа 7.22

Задайте ієрархію з трьох класів `Father`, `Son` і `Grandson`, пов'язавши `Father` та `Son` закритим успадкуванням, а `Son` і `Grandson` –відкритим. Означте в базовому класі відкритий метод

```
void Father::open(void);
```

Проаналізуйте можливість доступу до нього як у кожному з похідних класів, так і з боку клієнтів усіх похідних класів. Відкрийте доступ до закритого методу почергово в кожному похідному класі та простежте за доступом клієнтів похідних класів до щойно відкритого методу.

7.7. Комбінована ієрархія

Розглянувши два фундаментальні типи ієрархії, варто порівняти результати їх застосування. Спільною рисою успадкування та композиції залишається вкладення базового об'єкта до похідного (у разі успадкування) та компонента до композита (у разі композиції). Принципова відмінність композиції від застосування відкритого успадкування полягає в тому, що клієнти компонента не мають жодного доступу до композита (у разі композиції), тоді як клієнти об'єктів похідного класу мають доступ до відкритої частини базового класу.

Якщо ж порівняти закрите успадкування з композицією, то спільна риса залишається тією самою, а відмінність повністю зникає, оскільки клієнти об'єктів похідного класу тепер утрачають доступ до базового.

Розглянемо приклад. Доповнимо визначення стека з лістингу 7.1 делегуванням операції індексування.

Лістинг 7.62

```
// Стек із вкладеним до нього масивом
template <class Elem>
class StackContainingArray {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem> _stackArray; // Масив для розміщення стека
protected:
// Делегування масиву операції індексування: забезпечення доступу
// до неї в похідних класах
   const Elem& operator[] (const size_t index) const {
      return _stackArray[index];
   }
public:
// Конструктор, закритий від конвертування типів
   explicit StackContainingArray(const size_t size):
      _stackArray(size), _top(_bos) {}
   ~StackContainingArray() {}  // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
```

```
// Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {return _stackArray.size();}
// Керування стеком
// Видати верхівку стека
   const Elem& top() const {return _stackArray[_top];}
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(const Elem& value) {_stackArray[++_top]=value;}
};
template <class Elem>
const size_t StackContainingArray<Elem>::_bos=-1;
```

Далі утворимо стек із підгляданням звичним способом – за допомогою відкритого успадкування.

Лістинг 7.63

```
// Стек із підгляданням, похідний від стека
// з вбудованим масивом. Застосування делегованого індексування
template <class Elem>
class PeekbackStackDerivedFromArray:
      public StackContainingArray <Elem> {
public:
// Конструктор, закритий від конвертування типів
   explicit PeekbackStackDerivedFromArray
       (const size_t size): StackDerivedFromArraysize) {}
   ~PeekbackStackDerivedFromArray() {}  // Деструктор
// Функція підглядання
   bool peekback(const size_t i, Elem& elem) const {
      if (i>=size()) {elem=top(); return false;}
   elem=(*this)[i];
   return true;
   }
};
```

Відношення між класами показано на рис. 7.16.

Вправа 7.23

Складіть тестову програму та протестуйте щойно заданий стек із підглядянням.

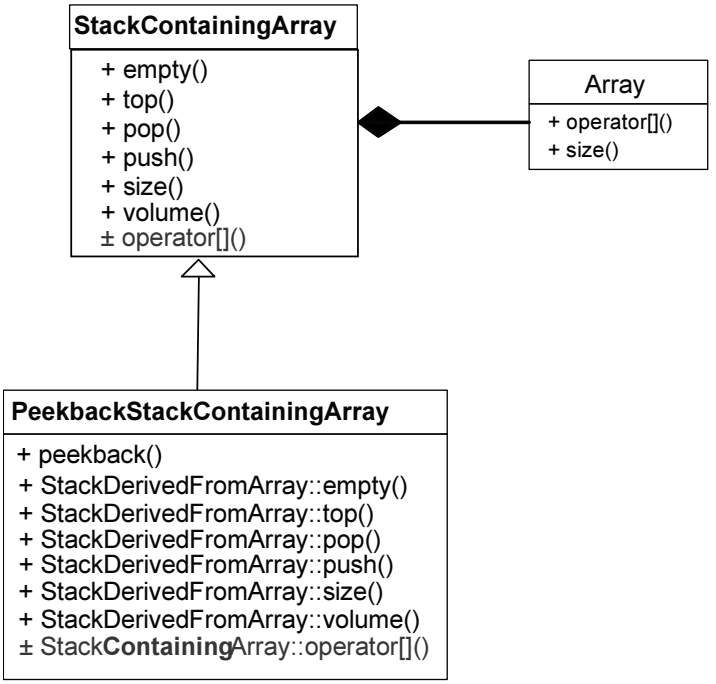

Рис. 7.16. Діаграма успадкувань стеку з підгляданням (вкладений масив)

Побудована тут ієрархія має принципову перевагу над попередніми. Вона полягає в точнішому виборі операції, доступ до якої відкрито похідному класу. Якщо у варіантах із попередніх підрозділів був відкритим доступ до всіх операцій індексування, то застосований тут метод делегування надає доступ тільки до сталої операції індексування, здатної функціонувати лише як селектор, але не як модифікатор. Тому немає небезпеки несанкціонованого втручання у внутрішність стека.

Вправа 7.24

У стеку, утвореному за допомогою закритого успадкування з масиву (лістинг 7.61), реалізуйте делегування до масиву захищеної сталої операції індексування для її використання в стеку з підгляданням. Виконайте тестування побудованих класів.

Вправа 7.24 демонструє, що вирішення проблеми організації доступу до закритої частини базового класу (зокрема, до сталої операції індексу-

вання з боку стека з підгляданням) за допомогою закритого успадкування та композиції рівноцінні. Це свідчить про те, що ієрархія класів та ієрархія об'єктів можуть певною мірою заміняти одна одну. Зауважимо, що замість композиції для вирішення зазначеної проблеми може бути успішно використана і агрегація масиву за допомогою указника.

Вправа 7.25

У стеку, до якого агреговано масив (вправа 7.3), реалізуйте делегування до захищеної сталої операції індексування для її використання в стеку з підгляданням. Протестуйте побудовані класи.

Порівнюючи вкладення (тобто композицію чи агрегацію) як ієрархію об'єктів із закритим успадкуванням як ієрархією класів, зазвичай перевагу надають вкладенню, передусім агрегації. У розглянутому прикладі застосування агрегації послабить залежність одного класу від іншого: реалізація масиву стане прихованою, адже масив із частини класу перетворюється на приєднаний до стека об'єкт. Це зумовлює більшу незалежність стека від масиву: наприклад, зміна структури масиву не спричинятиме повторної компіляції стека та похідних від нього класів. Наголосимо, що в разі застосування успадкувань залежність між класами залишається найсильнішою.

7.8. Кратне успадкування

Особливо обережно слід застосовувати *кратне успадкування* (multiple inheritance). Знову розглянемо приклад. Візьмемо класи кішок, хижаків і кімнатних тварин. На перший погляд, походження класу кішок `Cat` як від класу хижаків `Predator`, так і від класу кімнатних тварин `Pet` цілком коректне, однак тут криється певна суперечливість.

Почнемо з класу хижаків, представник якого має кличку та характеризується улюбленою здобиччю:

Лістинг 7.64

```
// Клас хижаків
class Predator {
private:
   string _name; // Прізвисько
   string _favoritePrey; // Переважна здобич
public:
   Predator(const string& prey, const string& name); // Конструктор
   ~Predator(); // Деструктор
   void setFavotitePray (const string&); // Модифікатор здобичі
```

```
   const string& getFavotitePray() const; // Селектор здобичі
   void setName (const string&); // Модифікатор прізвиська
   const string& getName() const; // Селектор прізвиська
};
```

А це клас кімнатних тварин, які також мають прізвиська, але характеризуються улюбленою іграшкою.

Лістинг 7.65

```
// Клас кімнатних тварин
class Pet {
private:
   string _name; // Прізвисько
   string _favoriteToy; // Улюблена іграшка
public:
   Pet (const string& prey, const string& name); // Конструктор
   ~Pet (); // Деструктор
   void setFavotiteToy (const string&); // Модифікатор іграшки
   const string& getFavotiteToy () const; // Селектор іграшки
   void setName (const string&); // Модифікатор прізвиська
   const string& getName() const; // Селектор прізвиська
};
```

Кожен із цих класів задає сукупність його повноцінних об'єктів – хижаків і кімнатних тварин. Окрім того, можливий варіант *змішування* (mixing) цих класів у одному похідному, наприклад у класі кішок `Cat` (або песиків, пацюків тощо). Схему успадкувань у цьому разі зображено на рис. 7.17.

Клас кішок може мати такий вигляд:

Лістинг 7.66

```
// Клас кішок: хижаки та кімнатні тварини водночас
class Cat: public Predator, public Pet {
public:
// Конструктор
   Cat(const string& prey, const string& toy, const string& name);
   ~Cat();  // Деструктор
};
```

Зверніть увагу: клас кішок не має власної закритої частини. Це не можна вважати великою несподіванкою: власної закритої частини не було й у класу квадратів із підрозділу 7.2 (див., наприклад, лістинг 7.10). Похідний клас успадковує закриту частину базового класу, а доступ до неї в такому разі надають селектори та модифікатори. Інша річ, що клас кішок утворено змішуванням двох базових класів, а тому

він успадкував дві закриті частини – по одній від кожного класу. Тепер атрибут прізвиська є в об'єкті класу кішок двічі, а тому виникає запитання: як забезпечити узгодженість цих імен або уникнути надлишкового кодування?

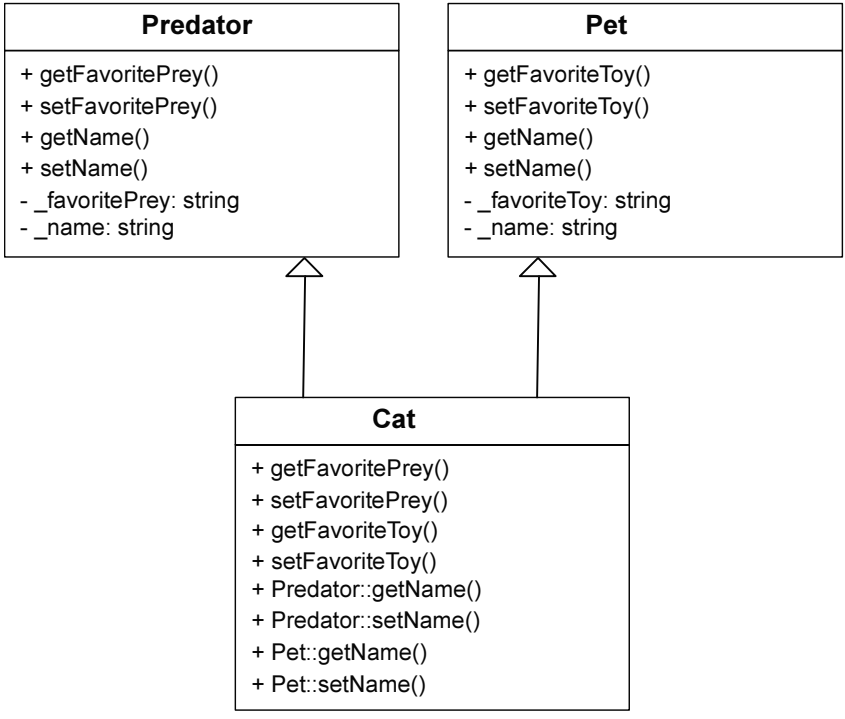

Рис. 7.17. Кратне успадкування

Розглянемо приклад, у якому використано кратне успадкування. Створимо об'єкт похідного класу, а потім змінимо атрибут одного з базових класів (лістинг 7.67).

Лістинг 7.67

```
Cat myCat("мишка", "клубок", "Мурка");
myCat.Predator::setName("Багіра");
```

Яку кличку матиме тепер кішка `myCat`? Як бачимо, застосовувати кратне успадкування слід украй обережно, тому що є суперечності між однойменними атрибутами та функціями в базових класах.

Вправа 7.26

Як коректно задати утиліти виведення в кожному з класів хижаків, кімнатних тварин і кішок? Означте й обґрунтуйте.

Проблема стає ще гострішою, якщо ланцюжок кратних успадкувань починається зі спільного базового класу. Подивимося, як відбувається успадкування в такому разі.

Розглянемо класи прямокутників і ромбів, які походять від спільного базового класу паралелограмів і, крім того, є базовими класами для квадратів.

Лістинг 7.68

```
typedef double length_t;
typedef double angle_t;
// Клас паралелограмів
class Parallelogram {
protected:
   length_t _height; length_t _width; // Сторони паралелограма
   angle_t _angle;   // Кут між сторонами
public:
   Parallelogram(const length_t h, const length_t w, const length_t a):
// Конструктор
                 _height (h), _width (w), _angle (a) {}
   ~Parallelogram() {}   // Деструктор
};
const double pi=3.141592653589793;
// Клас прямокутників
class Rectangle: public Parallelogram {
public:
   Rectangle(const length_t h, const length_t w):  // Конструктор
      Parallelogram (h, w, pi/2) {}
   ~Rectangle() {} // Деструктор
};
// Клас ромбів
class Rhombus: public Parallelogram {
public:
   Rhombus(const length_t side, angle_t a): // Конструктор
      Parallelogram (side, side, a) {}
   ~Rhombus() {} // Деструктор
};
// Квадрат походить водночас від прямокутника та ромба
class Square: public Rectangle, public Rhombus {
public:
```

```
   Square(short side): // Конструктор
      Rectangle(side, side), Rhombus (side, pi/2) {}
   ~Square() {} // Деструктор
};
```

Тепер у кожному екземплярі квадрата буде два екземпляри паралелограмів: один – від базового класу прямокутників, другий – від ромбів.

Вправа 7.27

Доповніть класи паралелограма, прямокутника, ромба та квадрата потрібними селекторами й модифікаторами, функціями обчислення площі та периметра, а також утилітою виведення. Складіть тестову програму та проаналізуйте виклики конструкторів під час створення об'єктів похідних класів.

На рис. 7.18 показано ієрархічну залежність створених класів.

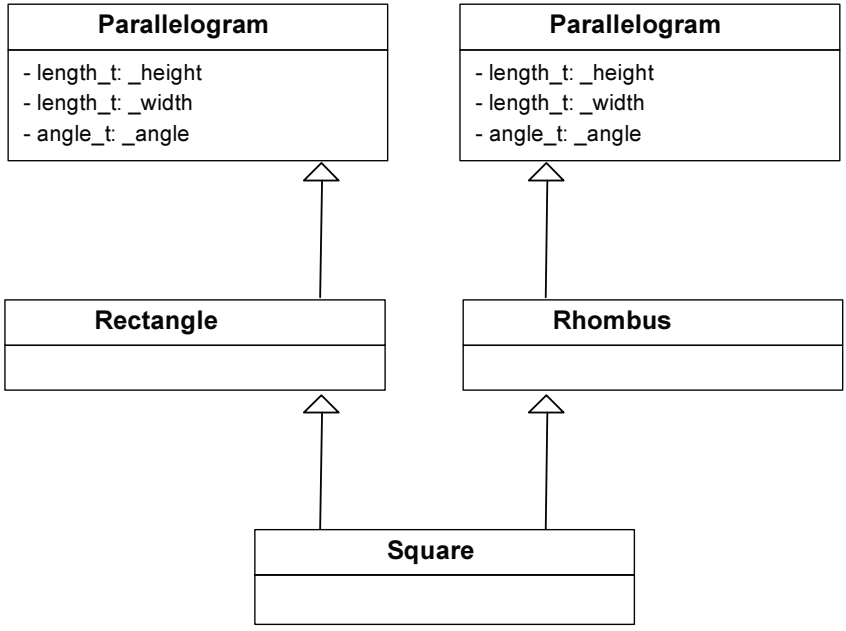

Рис. 7.18. Кратне успадкування із спільного базового класу

Однак нашому уявленню про взаємозв'язок між розглянутими видами чотирикутників більше відповідає рис. 7.19.

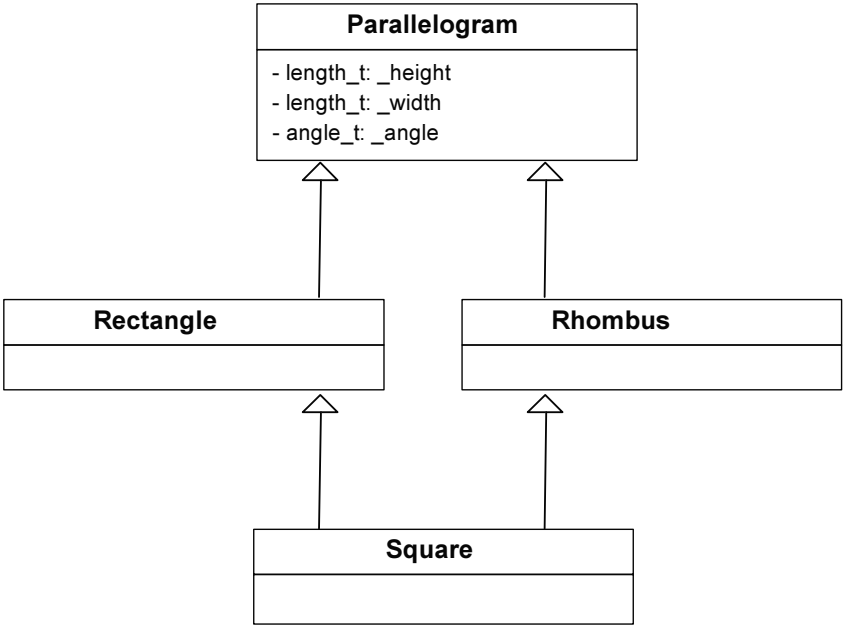

Рис. 7.19. Ієрархія успадкувань із спільного базового класу

Повністю досягти цього не вдається, однак є штучний прийом, який дає змогу скасувати стандартний порядок виклику конструкторів. Він має назву *віртуального успадкування* (virtual inheritance). Таке успадкування делегує виклик конструктора базового класу на рівень нижче.

Підправимо визначення класів так:

Лістинг 7.69

```
typedef double length_t;
typedef double angle_t;
// Клас паралелограмів (залишається без змін)
class Parallelogram {
protected:
   length_t _height; length_t _width; // Сторони паралелограма
   angle_t _angle;                    // Кут між сторонами
public:
// Конструктор
   Parallelogram(const length_t h, const length_t w, const length_t a):
                _height (h), _width (w), _angle (a) {}
   ~Parallelogram() {} // Деструктор
};
```

```
const double pi=3.141592653589793;
// Клас прямокутників: віртуальне успадкування
class Rectangle: virtual public Parallelogram {
public:
   Rectangle(const length_t h, const length_t w): // Конструктор
      Parallelogram (h, w, pi/2) {}
   ~Rectangle() {} // Деструктор
};
// Клас ромбів: віртуальне успадкування
class Rhombus: virtual public Parallelogram {
public:
   Rhombus(const length_t side, angle_t a): // Конструктор
      Parallelogram (side, side, a) {}
   ~Rhombus() {} // Деструктор
};
// Квадрат походить водночас від прямокутника та ромба;
// додано ініціалізацію паралелограма
class Square: public Rectangle, public Rhombus {
public:
   Square(short side): // Конструктор
      Parallelogram (side, side, pi/2),
      Rectangle(side, side),
      Rhombus (side, pi/2) {}
   ~Square() {} // Деструктор
};
```

Однак тут не повністю реалізовано схему з рис. 7.19. Означенню з лістингу 7.69 більше відповідає рис. 7.20.

Розглянемо створення прямокутника та ромба як самостійних об'єктів. У цьому разі віртуальність успадкування не проявляється. Створенню як прямокутника, так і ромба передує створення базових прямокутників.

Вправа 7.28

Виконайте трасування наведеного далі коду й простежте за послідовністю викликів конструкторів прямокутника та паралелограма, ромба та паралелограма.

Лістинг 7.70

```
// Створення ромба та базового паралелограма
   Rhombus(10, pi/3);
// Створення прямокутника та базового паралелограма
   Rectangle(10, 20);
```

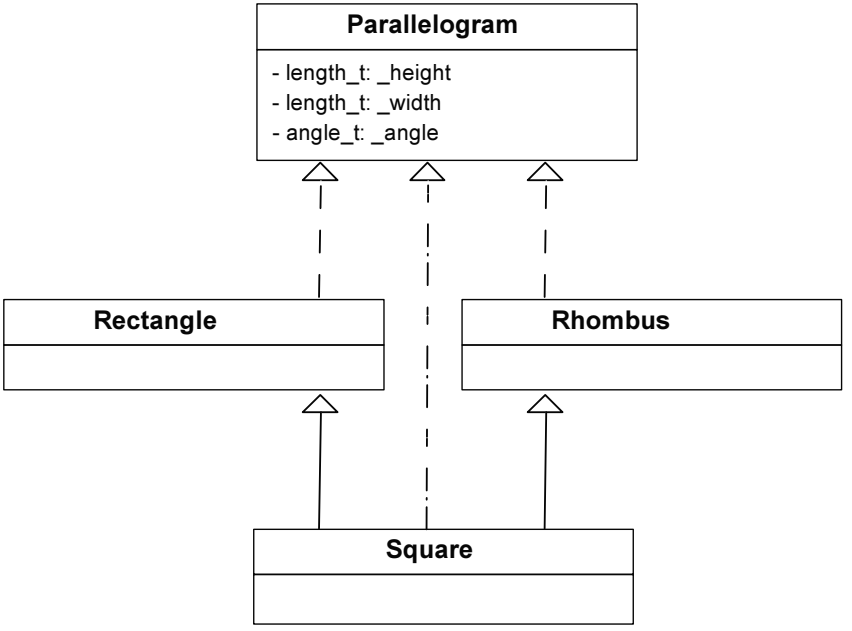

Рис. 7.20. Кратне віртуальне успадкування із спільного базового класу

Тепер розглянемо створення квадрата. Коли конструктор квадрата викликає конструктор прямокутника, спрацьовує віртуальне успадкування. Прямокутник не відповідає більше за створення паралелограма, як і ромб. За виклик конструктора паралелограма тепер відповідає квадрат, а тому в його конструкторі записано три ініціалізації.

Вправа 7.29

Виконайте трасування коду

```
// Створення квадрата, паралелограма, ромба та прямокутника
   Square(10);
```

і простежте за послідовністю викликів конструкторів прямокутника, ромба й паралелограма. Поміняйте порядок ініціалізації конструкторів базових класів у конструкторі квадрата й порівняйте послідовність виклику конструкторів із попереднім випадком. Видаліть ініціалізацію паралелограма з конструктора прямокутника та прокоментуйте результат компіляції.

На рис. 7.20 звичайні виклики конструкторів прямокутника та ромба позначено суцільною лінією, додатковий виклик конструктора паралелограма конструктором квадрата – штрих-пунктирною лінією, а скасовані виклики конструктора паралелограма з конструкторів прямокутника та ромба (віртуальне успадкування) – пунктирними.

Проблеми, виявлені під час вивчення кратного успадкування, – вагома підстава для того, щоб не використовувати його в практиці програмування. Точніше, не варто застосовувати повне кратне успадкування. Кратне успадкування ефективне в разі успадкування інтерфейсів.

7.9. Успадкування інтерфейсу й успадкування реалізації; абстрактні класи

Приклади класу стеків, розглянуті в цьому розділі, подають кожну конкретну реалізацію стека як окремий тип, не пов'язаний з іншими типами стеків. Водночас вони являють собою втілення однієї й тієї самої абстракції стека. Щоб запровадити цю абстракцію на рівні програми, використовують поняття *абстрактного інтерфейсу*, тобто інтерфейсу, реалізацію якого відкладено чи делеговано іншим класам. У мові C++ цього досягають за допомогою *абстрактних класів* (abstract class).

Абстрактний клас може містити лише сигнатури методів, які можуть не мати реалізації. Об'єкт цього класу не можна створити конструктором інакше, як через його похідні класи. Тому екземпляри абстрактних класів можуть з'явитися лише як базова частина конкретних екземплярів похідних класів. Виникає питання, чи можна вживати імена, типізовані абстрактними класами. Нехай `class Abstract` абстрактний клас, а `class Concrete: public Abstract` конкретний клас, похідний від нього. Тоді визначення `Abstract nothing;` некоректне, оскільки його виконання вимагало б особного виклику конструктора абстрактного класу, що неможливо, інакше в програмі з'явився б об'єкт з невизначеними методами. Але цілком можливим стає визначення указника `Abstract* p = new Concrete` так само, як відсилки абстрактного класу до конкретного об'єкта `Abstract& r = p`.

Як ми побачимо далі, основне застосування указників і відсилок до абстрактних класів полягає у передачі параметрів. Якщо `Stack` – це абстрактний клас, то його можна використати у функції як тип параметра, який передано за допомогою відсилки, наприклад так:

```
template <type Elem>
void process(Stack<Elem>& stackToProcess);
```

або як тип указника, значення якого буде конкретизовано під час створення:

```
Stack<char>* myStack= new StackContainingArray<char>(100);
```

Тепер, щоправда, виникне проблема типізації. Згідно з домовленостями щодо використання об'єктів похідних типів замість об'єкта базового типу, у цьому разі над об'єктами похідних класів виконуються функції базового класу. Однак можуть виникнути проблеми використання цих функцій, якщо доступ до об'єкта забезпечує базовий абстрактний клас, бо, як щойно було сказано, його методи можуть узагалі не мати реалізації. Вихід полягає в застосуванні *динамічного зв'язування* (dynamic binding) і базованого на ньому *поліморфізму* (polymorphism), що забезпечується механізмом *віртуальних функцій* (virtual function).

Наведемо та прокоментуємо приклад визначення абстрактного класу із віртуальними функціями.

Лістинг 7.71

```
// Інтерфейс стеків
template <class T>
class Stack {
public:
   virtual ~Stack() {}
   virtual bool empty() const=0; virtual const T& top() const=0;
   virtual void pop()=0; virtual void push(const T& value)=0;
};
```

У цьому визначенні використано кілька домовленостей.

Перша домовленість стосується регулювання механізмів *заміщення* (overriding) методів. Досі вибір функції для виклику ми задавали статично типом імені (псевдоніма, указника), використаного для доступу до об'єкта. Цей механізм називають статичним зв'язуванням імен. Таке зв'язування можна скасувати й замінити його динамічним, якщо оголосити функцію віртуальною. Виклик віртуального методу відбувається не за типом імені, а за його значенням. Тому у віртуальних методах можливе динамічне зв'язування залежно від типу значення, наявного на момент виклику: функція базового класу заміщується відповідною функцією найнижчого в ієрархії похідного класу, якому належить об'єкт.

Динамічне зв'язування дає змогу застосовувати в базовому класі віртуальні методи, що не матимуть реалізації. Їх називають *чисто віртуальними* (pure virtual). Наявність таких методів свідчить про належність класу до категорії абстрактних.

Друга зі згаданих раніше домовленостей стосується самого віднесення класу до категорії абстрактних. Про абстрактність класу свідчить наявність хоча б одного чисто віртуального методу, позначеного за допомогою виразу «=0».

Наведемо кілька прикладів використання інтерфейсу стеків.

Лістинг 7.72

```
// Обмежений стек із вбудованим масивом, похідний
// від абстрактного стека
template <class Elem>
class StackContainingArray: public Stack<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
   Array<Elem> _stackArray; // Масив для розміщення стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackContainingArray(const size_t size):
      _stackArray(size), _top(_bos) {}
   ~StackContainingArray() {} // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {return _top==_stackArray.size()-1;}
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const
      {return _stackArray.size();}
// Керування стеком
// Видати верхівку стека
   const Elem& top() const
      {return _stackArray[_top];}
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(Elem value)
      {_stackArray[++_top]=value;}
};
template <class Elem>
const size_t StackContainingArray<Elem>::
   _bos=-1;
```

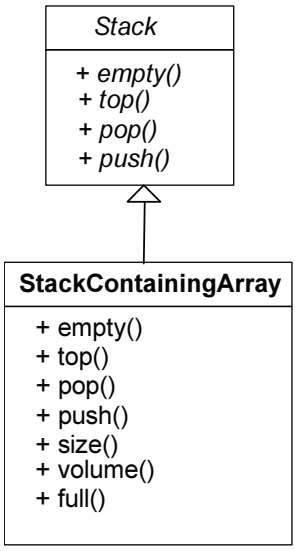

Рис. 7.21. Успадкування базового абстрактного класу

Схему класів зображено на рис. 7.21. Абстрактні класи та чисто віртуальні методи, як заведено, виділено курсивом.

Зверніть увагу на те, що власні функції стека на базі масиву, наприклад `size()` або

volume(), стануть недоступними через інтерфейс абстрактного класу. Тому, реалізуючи абстрактний клас, не варто додавати до інтерфейсу нові функції, оскільки вони виявляться недоступними через базовий інтерфейс.

Вправа 7.30

Базуючись на абстрактному класі стеків із лістингу 7.71, визначте абстрактний клас обмежених стеків так, щоб через нього був би доступний повністю весь інтерфейс як стеків із вбудованим масивом (лістинг 7.1), так і стека, до якого масив агреговано (вправа 7.3).

Подивимося тепер, на що перетвориться ієрархія класів, якщо реалізувати абстрактний клас стеків через стек, похідний від масиву.

Лістинг 7.73

```
// Обмежений стек, похідний від масиву та від
// абстрактного стека
template <class Elem>
class StackDerivedFromArray: public Stack<Elem>, protected Array<Elem> {
private:
   static const size_t _bos; // Дно стека
   size_t _top; // Верхівка стека
public:
// Конструктор, закритий від конвертування типів
   explicit StackDerivedFromArray(const size_t size): _top(_bos),
      Array<Elem>(size) {}  // Масив, ініціалізований як базовий клас
   ~StackDerivedFromArray() {}   // Деструктор
// Чи не порожній стек?
   bool empty() const {return _top==_bos;}
// Чи не переповнений контейнер?
   bool full() const {
// Виклик функції базового класу
      return _top==Array<Elem>::size()-1;
   }
// Видати поточний розмір стека
   size_t size() const {return _top+1;}
// Видати ємність контейнера
   size_t volume() const {
      return Array<Elem>::size();  // Звертаємося до функції базового
                                   // класу, адаптуючи її ім'я
   }
// Керування стеком
// Видати верхівку стека
```

```
const Elem& top() const {
   return (*this)[_top]; // До стека як представника класу,
    // похідного від масиву, застосовуємо його операцію індексування
}
```

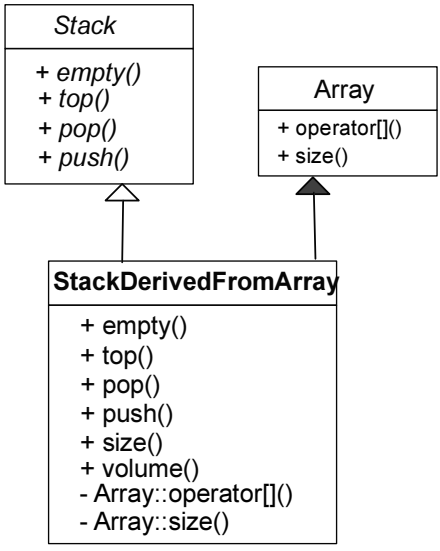

Рис. 7.22. Кратне успадкування інтерфейсу і реалізації

```
// Виштовхнути верхівку стека
   void pop() {_top--;}
// Помістити до стека
   void push(Elem value) {
// Знову застосовуємо операцію
// індексування
      (*this)[++_top]=value;
   }
}
template <class Elem>
const size_t
   StackDerivedFromArray<Elem>::
   _bos=-1;
```

Побудовану ієрархію зображено на рис. 7.22.

Тепер у контексті оголошення функції

```
template <typename Elem>
   int process(Stack<Elem>&
   stackToProcess);
```

можна записати її виклик до стека, похідного від масиву:

```
StackDerivedFromArray<int> stack(100);
process(stack);
```

Лістинг 7.74

або до стека з вбудованим масивом:

```
StackContainingArray<int> stack(100);
process(stack);
```

Лістинг 7.75

Ось простий приклад реалізації функції обробки стека, яка полягає в обчисленні суми його елементів:

```
// Функція підрахунку суми елементів стека,
// визначена з використанням абстрактного стека
int process(Stack<int>& stackToProcess) {
   int sum=0;
```

Лістинг 7.76

```
   while(!stackToProcess.empty()) {
      sum+=stackToProcess.top(); stackToProcess.pop();
   }
   return sum;
}
```

Вправа 7.31

Реалізуйте варіант функції підрахунку суми елементів стека, яка залишає його вміст незмінним.

Функцію, яка використовує абстрактні класи, максимально налаштовано на нові застосування. Адже вона не містить зайвих припущень про реалізацію класу, до якого її застосовують (у розгляданому випадку – стека), способу його реалізації, його обмеженості тощо, а тому придатна для обробки стеків будь-яких типів, зокрема нових, що можуть з'явитися вже після її створення й навіть компіляції. Система, яка виконуватиме таку функцію, має бути здатна до відкладеного зв'язування виклику функції залежно від конкретного типу фактичного параметра (наперед не відомого).

Розглянемо ще один простий приклад. Звернімося до схеми, зображеної на рис. 7.23.

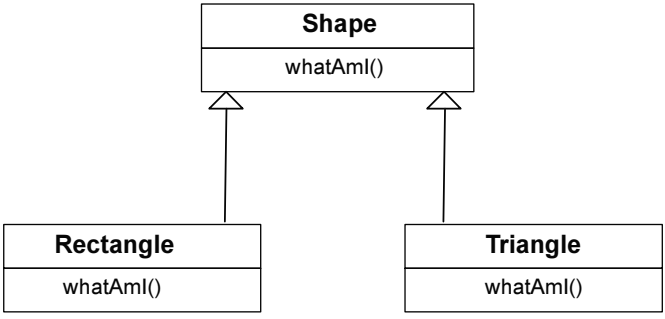

Рис. 7.23. Ієрархія геометричних фігур

Тут немає абстрактних класів, але використано віртуальні функції. На лістингу 7.77 подано можливий варіант визначення класів зі схеми рис. 7.23 в разі застосування раннього (статичного) зв'язування.

Лістинг 7.77

```
// Клас фігур невідомого типу
class Shape {
public:
// Виведення типу фігури
   void whatAmI() {
```

```
      cout<<"I don't know what kind of shape I am! "<<endl;
   }
}
// Клас прямокутників
class Rectangle: public Shape {
public:
// Виведення типу фігури
   void whatAmI() {cout<<"I'm a rectangle! "<<endl;}
};
// Клас трикутників
class Triangle: public Shape {
public:
// Виведення типу фігури
   void whatAmI() {cout<<"I'm a triangle! "<<endl;}
};
```

Запропоновану реалізацію важко назвати вдалою. Як видно з наступного лістингу, правильну відповідь про тип фігури вдається одержати не завжди, позаяк один і той самий об'єкт – прямокутник `s` – по-різному реагує на спробу застосувати до нього метод `whatAmI()`.

Лістинг 7.78

```
// Використання раннього зв'язування
int main() {
   Shape* p; Rectangle s(10, 20);
   s.whatAmI();  // Виклик функції прямокутника
   p=&s;
   p->whatAmI(); // Виклик функції фігури невідомого типу
   return 0;
}
```

Справді, статичне зв'язування не може адекватно реагувати на тип значення указника. Надамо тепер функції виведення типу фігури статус віртуальної:

Лістинг 7.79

```
// Клас фігур невідомого типу
class Shape {
public:
// Віртуальна функція виведення типу фігури
   virtual void whatAmI() {
      cout<<"I don't know what kind of shape I am! "<<endl;
   }
};
// Клас прямокутників
class Rectangle: public Shape {
```

```
public:
// Віртуальна функція виведення типу фігури
   void whatAmI() {cout<<"I'm a rectangle! "<<endl;}
};
// Клас трикутників
class Triangle: public Shape {
public:
// Віртуальна функція виведення типу фігури
   void whatAmI() {cout<<"I'm a triangle! "<<endl;}
};
```

Ось простий приклад, який демонструє результат викликів віртуальної функції `whatAmI()` засобами пізнього (динамічного) зв'язування:

Лістинг 7.80

```
// Використання пізнього зв'язування
int main() {
   Shape *s1, *s2, *s3;
   s1=new Rectangle; s2=new Triangle; s3=new Shape;
   s1->whatAmI(); // Виклик функції виведення прямокутника
   s2->whatAmI(); // Виклик функції виведення трикутника
   s3->whatAmI(); // Виклик функції фігури невідомого типу
   return 0;
}
```

Зверніть увагу, що в усіх трьох випадках виклику методу `whatAmI` використано указник одного й того самого типу, але буде викликано різні методи залежно від типу значення, на яке встановлено указник.

Вправа 7.32

Запрограмуйте й випробуйте меню створення фігури на замовлення користувача, виражене в діалозі з програмою.

У розглянутих прикладах бачимо прояви поліморфізму – можливості використання об'єктів одного типу (наприклад, прямокутників або трикутників) замість об'єктів іншого типу (фігур). Конкретно поліморфізм проявляється як можливість викликати функції похідного класу за допомогою інтерфейсу базового класу. Для реалізації поліморфізму потрібен механізм динамічного розпізнавання типу об'єкта. Справді, у наведеному прикладі кожен із трьох указників має один і той самий тип `Shape`, але їхні значення відрізняються типами: для `s1` це `Rectangle`, для `s2` – `Triangle` та, нарешті, для `s3` – сам тип `Shape`. Отже, виклик віртуальної функції має не прямо виконуватись, а бути схожим на виклик функції за указником, який відсилає до методу в потрібному класі. Для реалізації такого доступу кожен клас із

віртуальними функціями наділено спеціальною таблицею віртуальних функцій, що вже відсилає до самих функцій.

Тому виклик віртуальної функції має складатися з двох кроків: 1) на етапі виконання програми перед викликом функції слід визначити клас, якому належить об'єкт, до котрого застосовано цю функцію; 2) потрібно звернутися до таблиці віртуальних функцій цього класу та взяти з неї потрібну функцію. Статично (до виконання програми) цього зробити не можна, тому що одна й та сама команда виклику функції може виконуватися багаторазово та щоразу використовувати об'єкт іншого типу.

Наступний приклад демонструє, що навіть один і той самий програмний код може залежно від стану процесу виконання програми зумовлювати виклики різних функцій.

Лістинг 7.81

```
// Гра з фігурами
int main() {
   Shape* s[10]; char chr;
// Створення фігур на замовлення
   for (int i=0; i<10; ++i) {
      cout<<"what to place to s["<<i<<"]? Aswer s/r/t"; cin>>chr;
      switch (chr) {
      case 's': s[i]=new Shape; break;
      case 'r': s[i]=new Rectangle; break;
      case 't': s[i]=new Triangle; break;
      default: --i;
      };
   }
   for(i=0; i<10; ++i)     // Виведення створених фігур
      s[i]->whatAmI(); // У цій команді кожного разу може виконуватись
                       // інший код
   return 0;
}
```

Особливу роль у поліморфізмі відіграють *віртуальні деструктори*. Вони призначені для встановлення правильної послідовності виклику деструкторів у разі поліморфного використання об'єктів. Розглянемо простий приклад успадкування, у якому як базовий, так і похідний класи використовують динамічну пам'ять.

Лістинг 7.82

```
// Базовий клас
class Person {
   char* _name;
public:
```

```
// Конструктор
   Person (const char* name): _name(new char[strlen(name)+1]) {
      strcpy(_name, name);
   }
// Віртуальний деструктор
   virtual ~Person() {delete [] _name;}
};
// Похідний клас
class Student: public Person {
   char* _inmatriculation;
public:
// Конструктор
   Student (const char* name, const char* m):
      Person(name), _inmatriculation(new char[strlen(m)+1]) {
      strcpy(_inmatriculation, m);
   }
// Віртуальний деструктор
   virtual ~Student() {delete [] _inmatriculation;}
};
```

Створимо об'єкт похідного класу за указником:

```
Person* p=new Student("Qwerty", "123456789");
```

Зрозуміло, що виклик конструктора `Student("Qwerty", "123456789")` передбачає виконання обох конструкторів як похідного, так і базового класів. Виникає запитання: як бути з викликом деструктора командою **`delete`** `p`? Ось тут виявляється потрібним віртуальний деструктор: у цьому разі завдяки поліморфізму буде викликано деструктор об'єкта класу студентів, який видалить свою частину динамічної пам'яті, а саме значення указника `_inmatriculation`, а потім на загальних підставах відпрацювання деструкторів уздовж ланцюжка успадкувань спрацює деструктор базового класу, який видалить значення указника `_name`. Отже, для застосовування поліморфізму потрібні віртуальні деструктори.

Вправа 7.33

Протестуйте видалення об'єктів класів `Person` і `Student` у разі використання віртуальних і невіртуальних деструкторів.

7.10. Архітектура ієрархій об'єктів і класів

У попередньому розділі було з'ясовано, що відкриту віртуальну функцію можна розглядати водночас не тільки як частину інтерфейсу класу, який надано зовнішньому коду, але і як засіб налаштувати його по-

ведінку, наданий похідним класам. Використання цієї, здавалось би, переваги призводить до суттєвого зниження якості програмного забезпечення. Доцільно поділити частини базового класу за цільовим призначенням, виділивши ту, яку призначено для успадкування інтерфейсу, і відділивши її від частини, що забезпечуватиме налаштування поведінки. Пам'ятаймо, що успадкування інтерфейсу призначене для клієнтів класу, а налаштувення поведінки – для його похідних класів. Якість архітектури підвищиться, якщо заборонити користувачам різного типу (клієнтам класу та похідним класам) застосовувати один і той самий засіб з різною метою. Оскільки інтерфейсна функція відкрита, то вона не повинна вживатися для налаштувення поведінки, а тому не варто робити її віртуальною.

Тому рекомендовано визначати інтерфейсні функції як відкриті невіртуальні. Тоді всі похідні класи використовують спільний інтерфейс, не призначений для налаштувення поведінки. Це правило відоме під назвою *ідіоми невіртуального інтерфейсу NVI* (Non Virtual Interface).

Отже, завдяки ідіомі невіртуального інтерфейсу у відкритій частині класу не може з'явитися жодної віртуальної функції. Проте саме це правило ще не гарантує стійкої стосовно використаного інтерфейсу поведінки об'єктів. Як відомо з попереднього розділу, неоднозначність поведінки виявляється в разі перевизначення відкритих невіртуальних базових функцій у похідних класах.

Тому запровадимо ще одне правило невіртуальності інтерфейсу, а саме: інтерфейсні функції не можна перевизначати в похідних класах. Його називають *ідіомою вільного від заміщень інтерфейсу*.

Іще раз наголосимо, що введені нами норми визначення структури класів виділяють з усіх можливих синтаксично правильно побудованих класів ті, будова яких коректна з прагматичних позицій. Із синтаксичного погляду відкрита функція може з'явитись у будь-якому класі як така, котру довільний клієнт класу може використати з будь-якою метою. У контексті ідіоми невіртуального інтерфейсу відкрита функція – це така, яку призначено клієнтові для визначення спільної точки входу до всіх різновидів поліморфного типу.

Аналогічно, реалізацію будь-якої відкритої функції базового класу можна синтаксично змінити в похідному класі. Із погляду ідіоми вільного від заміщень інтерфейсу така структура успадкувань перестає відповідати прагматичним вимогам до їхньої архітектури.

Застосуємо ідіому NVI до ієрархії «прямокутник – квадрат».

Лістинг 7.83

```
typedef unsigned int length_t;
// Клас прямокутників, розділений на інтерфейс і налаштування
class NVIRectangle {
// Відкритий інтерфейс NVI
public:
// Конструктор
   NVIRectangle(const length_t a, const length_t b): _a(a), _b(b) {}
   NVIRectangle() {}  // Деструктор
// Селектори:
// реалізований у базовому класі
   length_t getA() const {return _a;}
// призначений для налаштування
   length_t getB() const {return do_getB();}
// Модифікатори:
// реалізований у базовому класі
   void setA(const length_t a) {_a=a;}
// призначений для налаштування
   void setB(const length_t b) {return do_setB(b);}
// Обчислення площі прямокутника
   length_t area() const {return do_area();}
private:
   length_t _a, _b;  // Сторони прямокутника
// Закрите налаштування
// Налаштування селектора другої сторони
   virtual length_t do_getB() const {return _b;}
// Налаштування модифікатора другої сторони
   virtual void do_setB(const length_t b) {return _b=b;}
// Налаштування функції обчислення площі
   virtual length_t do_area() const {return getA()*getB();}
};
// Клієнтський код
void increase_a(NVIRectangle& r, const unsigned int m) {
   r.setA(r.getA()*m);
}
```

Тепер похідний клас квадратів здобуває право налагоджувати віртуальні функції `do_getB()`, `do_setB()` та `do_area()`, але не може заміщувати невіртуальні функції `getA()`, `setA()` й `area()`, як і утиліту `increase_a()`, яку, з огляду на її відкритий статус, прирівнюють до інтерфейсу.

Лістинг 7.84

```
// Клас квадратів
// Успадкування інтерфейсу та налаштування реалізації
class NVISquare: public NVIRectangle {
```

```
public:
// Конструктор
   NVISquare(const length_t x): NVIRectangle(x, x) {
// Якщо це судження не справджується, в селекторах припущено помилки
      assert(getA()==getB());
   }
   ~NVISquare() {} // Деструктор
private:
// Налаштування поведінки прямокутника на поведінку, властиву квадрату
// Налаштування селектора другої сторони
   virtual length_t do_getB() const {return getA();}
// Налаштування модифікатора другої сторони
   virtual void do_setB(const length_t b) {return setA(b);}
// Налаштування функції обчислення площі
   virtual length_t do_area() const {return getA()*getA();}
};
```

Проаналізуємо поведінку кожного з класів.

Застосування ідіоми невіртуального інтерфейсу гарантує, що в контексті як прямокутника, так і квадрата буде використано один і той самий інтерфейс. Справді, візьмемо код, який оперуватиме безпосередньо з квадратом –

```
NVISquare sqr(10);
```

і викличемо модифікатор сторони a:

```
sqr.setA(15);
```

Оскільки квадрат не має власних відкритих функцій, окрім конструктора та деструктора, то буде викликано функцію базового класу `NVIRectangle::setA(15)`, яку в ньому ж і реалізовано. Тепер обчислюємо площу:

```
cout<<sqr.area()<<endl;
```

Це знову ж функція класу прямокутників `NVIRectangle::area()`, але, на відміну від попередньої, її не реалізовано в ньому, а призначено для налаштування за допомогою закритої віртуальної функції. У виклику `do_area()` закрита віртуальна функція класу прямокутників `NVIRectangle::do_area()` заміщується такою самою функцією класу квадратів `NVISquare::do_area()`. Якщо тепер викликати не метод, а утиліту

```
increase_a(sqr, 2);
```

то відповідно збільшиться сторона a прямокутника. У разі ж спроби вивести іншу сторону –

```
cout<<sqr.getB()<<endl;
```

виклик do_getB() знову зумовить її заміщення в класі квадратів, а тому врешті-решт буде виконано функцію NVIRectangle:: getA(), яка забезпечить потрібний результат.

Нічого не зміниться, якщо відразу застосувати інтерфейс прямокутника:

Лістинг 7.85

```
NVIRectangle* rec=new NVISquare(10);
rec->setA(15); cout<<rec->area()<<endl;
increase_a(rec, 2); cout<<rec->getB()<<endl;
```

Буде викликано всі ті самі функції в такому самому порядку, оскільки всі зовнішні виклики завжди були викликами методів інтерфейсного класу прямокутників.

Вправа 7.34

Застосуйте ідіому невіртуального вільного від налаштування інтерфейсу до класу фігур і його похідних класів.

Отже, віртуальні функції краще закрити. Тоді похідні класи можуть пропонувати власну реалізацію закритої й тому недоступної для клієнтського коду частини базового класу. Приходимо до *ідіоми закритих віртуальних налаштувань*.

У найзагальнішому вигляді цей важливий *архітектурний принцип успадкувань інтерфейсу* можна сформулювати так: потрібно використовувати невіртуальний вільний від заміщень інтерфейс у поєднанні із закритими віртуальними налаштуваннями.

Вправа 7.35

Застосуйте архітектурний принцип успадкувань інтерфейсу до ієрархій «паралелограм – прямокутник», «ромб – квадрат».

Із правила закритих віртуальних налаштувань є один виняток, який стосується деструкторів: вони хоч і відкриті, але мають бути віртуальними, щоб вивільнялася пам'ять, зайнята похідними об'єктами.

До запропонованої ієрархії все ще можна зробити зауваження. Перевага об'єктно-орієнтованого підходу, як уже було зазначено, полягає в побудові таких ієрархій, які сприятимуть зменшенню, а точніше впорядкуванню складності великих програмних систем. Розглядаючи ієрархію в цьому ас-

пекті, можна помітити, що структура похідних класів зовсім не впливає на структуру базових, тоді як самі похідні класи суттєво залежать від базових. Це означає, що надмірна конкретизація базових класів звужує сферу застосування їхніх похідних класів, а спроба модифікувати базовий клас (наприклад, із метою узагальнення, щоб розширити сферу його застосувань) зумовить потребу переглянути всі базові класи. У цьому розумінні модифікація базових класів у будь-якій програмній системі – найвитратніша процедура, оскільки вона стосується всієї архітектурної піраміди.

У кожній ієрархії класів можна виділити *кореневий клас* (краще, коли він один), який породжує всі інші класи в ієрархії, і *термінальні класи*, які не мають похідних. Як кореневий доцільно використовувати абстрактний клас із невіртуальним інтерфейсом. Виявляється, усі інші нетермінальні класи теж мають бути абстрактними.

Виходячи зі сказаного, будемо намагатися задавати нетермінальні класи якомога неконкретними. Як міру конкретності класу запропонуємо взяти його реальність, тобто здатність класу до створення власних екземплярів. Говорячи формально, наявність у класі хоча б однієї чисто віртуальної функції відразу перетворює його на абстрактний, який не може створювати самостійні екземпляри. Оскільки деструктор кожного базового інтерфейсного класу має бути віртуальним, то, формально перетворивши деструктор на абстрактний метод, можна зробити будь-який клас абстрактним. Зрозуміло, що таким способом не вдасться досягнути підвищення змістовної абстрактності класу, але принаймні стане неможливим існування його незалежних екземплярів.

Приходимо до ідіоми застосування *абстрактного деструктора*: у нетермінальному класі має бути абстрактний деструктор (тобто віртуальний, позначений нулем).

У кожному ланцюжку успадкувань можна виділити класи, від яких не успадковано жодного похідного класу. Ми назвали їх термінальними. Термінальність – це не властивість класу, адже будь-коли хто завгодно може формально створити з нього похідний. Щоб усунути можливість подальших спадкувань необхідно закрити його конструктор. Приходимо до ідіоми *примусової термінальності класу*: примусово термінальним називатимемо клас, у якому функції конструкторів виконують так звані *псевдоконструктори* (pseudo constructor), які також називають *функціями клонування* (clone).

Наведемо приклад структури типового примусово термінального класу, наділеного псевдоконструктором:

```
// Термінальний клас, у якого не може бути похідних класів
class Terminal {
public:
   static Terminal* clone(); // Псевдоконструктор
   static Terminal* clone(const Terminal&); // Псевдоконструктор копіювання
private:
// Закриті, а тому недоступні потенційним похідним класам:
   Terminal();  // Конструктор
   Terminal(const Terminal&);  // Конструктор копіювання
};
// Реалізація псевдоконструкторів
Terminal* Terminal::clone() {return new Terminal();}
Terminal* Terminal::clone(const Terminal& a) {return new Terminal(a);}
```

Лістинг 7.86

Наявність неабстрактних класів на ланцюжку спадкувань приводить по появи об'єктів з різних його рівнів. Так поява екземпляру класу NVIRectangle одночасно з NVISquare приводить до виникнення проблем забутливого присвоєння (підрозділ 7.3). Тому ієрархію прямокутник-квадрат варто розділити на дві частини: абстрактний інтерфейс ний клас і конкретні термінальні класи.

Поєднання ідіом невіртуального інтерфейсу та нетермінальної абстрактності зумовлюють виникнення поняття *стабільного базового інтерфейсу* – невіртуального, вільного від заміщень інтерфейсу абстрактного базового класу з віртуальним деструктором.

Стабільний базовий інтерфейс гарантує можливість *нових застосувань* (reuse) лише інтерфейсу, а не реалізації. Тому немає сенсу наповнювати базовий клас атрибутами та реалізаціями методів. У цьому разі його типовий склад – чисто віртуальні закриті функції для налаштування відкритого інтерфейсу. Це мінімізує припущення про базовий клас, а отже, уможливлює якнайширше коло застосувань.

Лістинг 7.87

```
// Стабільний базовий інтерфейс прямокутників
class InterfaceRectangle {
public:
   InterfaceRectangle() {}  // Конструктор
   virtual ~InterfaceRectangle() {}  // Деструктор
// Стабільний невіртуальний інтерфейс
   unsigned int& a() {return do_a();}
   unsigned int& b() {return do_b();}
   const unsigned int& a() const {return do_a();}
   const unsigned int& b() const {return do_b();}
   unsigned int area() const {return do_area();}
```

```
private:
// Налаштування реалізації
   virtual unsigned int& do_a()=0;
   virtual const unsigned int& do_a() const=0;
   virtual unsigned int& do_b()=0;
   virtual const unsigned int& do_b() const=0;
   virtual unsigned int do_area() const=0;
};
// Функція масштабування сторони a
void increase_a(InterfaceRectangle& r, const unsigned int m) {
   r.a()*=m;
}
// Функція масштабування сторони b
void increase_b(InterfaceRectangle& r, const unsigned int m) {
   r.b()*=m;
}
```

Тепер класи як прямокутників, так і квадратів стають повноцінними різновидами поліморфного типу, заданого стабільним базовим інтерфейсом: у них немає власних відкритих функцій (крім стандартного набору конструкторів, деструктора та присвоєнь). Досягнута гармонія в ієрархії полягає в гарантованому використанні єдиного інтерфейсу незалежно від способів доступу до похідних об'єктів, а закритий характер віртуальних функцій перетворює налаштування поведінки на внутрішню, закриту від клієнта, проблему реалізації похідних класів.

Лістинг 7.88

```
// Термінальний клас прямокутників
class Rectangle: public InterfaceRectangle {
public:
// Конструктор
   Rectangle(const unsigned int a, const unsigned int b): _a(a), _b(b) {}
private:
   unsigned int _a, _b; // Сторони
// Закрите налаштування
   virtual unsigned int& do_a() {return _a;}
   virtual const unsigned int& do_a() const  {return _a;}
   virtual unsigned int& do_b() {return _b;}
   virtual const unsigned int& do_b() const {return _b;}
   virtual unsigned int do_area() const {return _a*_b;}
};
// Термінальний клас квадратів
class Square: public InterfaceRectangle {
```

```
public:
   Square(const unsigned int x): _a(x) {}  // Конструктор
private:
   unsigned int _a; // Сторона
// Закрите налаштування
   virtual unsigned int& do_a() {return _a;}
   virtual const unsigned int& do_a() const {return _a;}
   virtual unsigned int& do_b() {return _a;}
   virtual const unsigned int& do_b() const {return _a;}
   virtual unsigned int do_area() const {return _a*_a;}
};
```

Вправа 7.36

Доповніть класи ієрархії прямокутників програмованими конструкторами, деструкторами та присвоєннями.

Вправа 7.37

Застосуйте до класів ієрархії прямокутників ідіому нетермінальної абстрактності, заборонивши можливість створення класів, похідних від неінтерфейсних.

Застосуємо стабільний базовий інтерфейс до класу стеків. Визначимо абстрактний інтерфейсний клас (порівняйте з лістингом 7.71):

Лістинг 7.89

```
// Стабільний базовий інтерфейс стека елементів типу Elem
template <class Elem> class Stack {
public:
   Stack() {}  // Конструктор стека
   virtual ~Stack() {}  // Деструктор стека
   bool empty() const {return doEmpty();} // Перевірити, чи порожній стек
   const Elem& top() const {return doTop();} // Показати верхівку стека
   void pop() {doPop();}  // Виштовхнути верхівку стека
   void push(const Elem& el) {doPush(el);} // Проштовхнути елемент у стек
private:
// Налаштування поведінки стеків
   virtual bool doEmpty() const=0;
   virtual const Elem& doTop() const=0;
   virtual void doPop()=0;
   virtual void doPush(const Elem&)=0;
// Закриті нереалізовані операції
   Stack operator=(const Stack&);
};
```

Відкрите успадкування як засіб підтримки типізації всім похідним класам надає доступ до стабільного спільного інтерфейсу. Для налаштування базового класу використовують його закриті віртуальні методи. Головна перевага стабільного базового інтерфейсу полягає в тому, що можливі різні його реалізації, без жодних обмежень. Можна реалізувати стек на вбудованому контейнері, наприклад масиві чи списку, або застосувати закрите чи захищене успадкування від контейнера (рис. 8.9). Утім найголовніше те, що абстрактний клас не лише не має обмежень на нові застосування для самого себе, але й гарантує нові застосування будь-якому з клієнтів, які використовують його як інтерфейс.

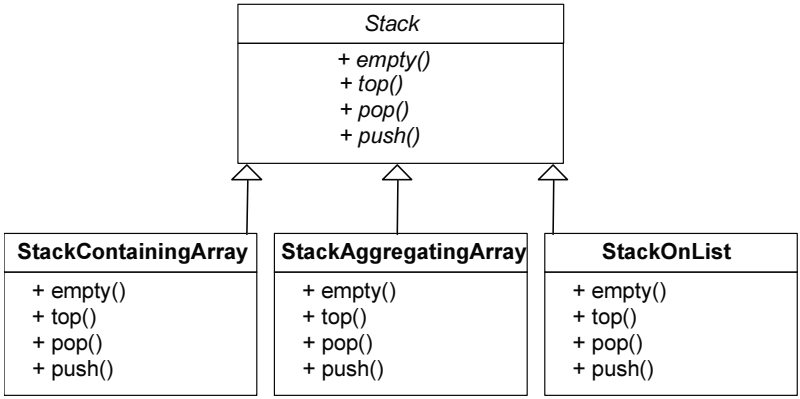

Рис. 7.24. Діаграма стеків, похідних від абстрактного класу

Вправа 7.38

Реалізуйте стабільний базовий інтерфейс стека для стеків, наведених на схемі класів (рис. 7.24): із вбудованим масивом, з агрегованим масивом і на базі списку.

Завдання до розділу 7

1. Визначте абстрактний клас трикутників і дві його реалізації: за трьома вершинами і за трьома сторонами.
2. Доповніть клас трикутників конструктором за основами його трьох медіан. Реалізуйте метод обчислення довільної медіани трикутника.
3. За довільною стороною трикутника знайдіть паралельну їй середню лінію.
4. Визначте ієрархію трикутник-прямокутний трикутник. Реалізуйте функції обчислення довжин радіусів вписаного і описаного кіл.

5. Визначте абстрактний клас трикутних призм `Prism`, передбачивши функції доступу до його складових: основ `Base`, бічних граней `Face`, ребер `Edge` і вершин `Vertex`.
6. Визначте абстрактний клас трикутних пірамід `Pyramid` і побудуйте ієрархію: піраміда, правильна піраміда, зрізана піраміда.
7. Визначте клас двосторонніх черг `DeqList` на базі списків.
8. Визначте клас двосторонніх черг `DeqArray` на базі масивів.
9. Організуйте абстрактний клас інтерфейсу черг `Queue` для обох типів черг.
10. Включіть до інтерфейсу і реалізуйте в кожному з класів функцію інвертування черги.
11. Визначте ієрархію черг, що міститиме абстрактну чергу, чергу з підгляданням `PeekBackQueue` і чергу з шахраюванням `CheatQueue` (остання дає змогу не лише підглянути, але й замінити заданий елемент черги).
12. Визначте абстрактний клас виробів `Goods`. Кожен виріб має інвентарний номер (штрих-код), дату виготовлення, вартість, назву і додаткові характеристики. Вироби можна виводити. Виводяться штрих-код, дата виготовлення, вартість, назва і додаткові характеристики.
13. На базі класу `Goods` визначте конкретний клас продуктів харчування `Product`. Додатковою характеристикою є термін придатності.
14. На базі класу `Goods` визначте конкретний клас предметів одягу `Clothes`. Додатковою характеристикою є розмір.
15. Визначте комору `Storehouse` для зберігання виробів, відсортованих за штрих-кодами. В кожній програмі існує лише одна комора. Визначте функцію виведення інвентарної відомості комори.
16. Реалізуйте метод `find()` підрахунку за даним штрих-кодом кількості виробів у коморі.
17. Реалізуйте метод `include()` занесення виробу до комори.
18. Реалізуйте метод `exclude()` вилучення виробу з комори.
19. Запрограмуйте створення і обробку аварійних ситуацій а) `Lack`, спричиненої наявністю критичної кількості виробів з даним штрих-кодом в коморі; б) `ShelfLife` – критичністю терміну зберігання.
20. Визначте клас касових апаратів `CashRegister` для підрахунку вартості та створення чеку покупки за прейскурантами комори.
21. Запропонуйте засоби підрахунку скидок залежно від: а) терміну придатності продукту; б) кількості одиниць виробів в одному чеку; в) загальної вартості всього чеку.
22. Визначте клас плоских фігур `Shape` на координатній площині, доповнивши типову структуру класу мультиконструктором копіювання, вір-

туальними утилітами введення, виведення, дзеркального відображення відносно осей координат, центрально-симетричного відображення відносно початку координат, повороту, паралельного переносу.

23. Клас многокутників `Polygon` є варіантом класу плоских фігур. Запропонуйте дві реалізації класу `Polygon` на базі масиву і на базі списку. Реалізуйте утиліту дзеркального відображення многокутника за довільною віссю координат.
24. В коло заданого радіуса r з центром у початку координат вписано правильний многокутник. Реалізуйте клас `RegPolygon` правильних вписаних многокутників, спираючись на поведінку класу `Shape`. Конструктор будує його вершини в напрямку, протилежному руху годинникової стрілки, починаючи з вершини $(r,0)$.
25. Реалізуйте в класі `RegPolygon` віртуальні утиліти введення, виведення, дзеркального відображення відносно осей координат, центрально-симетричного відображення відносно початку координат, повороту, паралельного переносу.
26. Визначте абстрактний ітератор та запрограмуйте з його допомогою злиття двох впорядкованих списків довільного виду в один.
27. Визначте клас `Functor` функторів – параметричних елементарних функцій-методів класу виду $y=f(a,x)$. Параметр `a` реалізувати атрибутом класу, а аргумент `x` – формальним параметром методу. Передбачити методи `myexp` для a^x, `mylog` для $\log_a x$, `mysin` для $\sin ax$, `mycos` для $\cos ax$.
28. Визначте клас `Oper` калькулятора арифметичних операцій над елементарними функціями, передбачивши в ньому метод `execute` виконання довільної арифметичної операції над довільною парою функцій в заданій точці x.
29. Реалізуйте функцію `tabulate` для табулювання значень довільної елементарної функції або результату арифметичної операції над функціями класу `Functor` на інтервалі $[x_0, x_1]$.

Навчальні проекти

1. Задача побудови опуклої оболонки множини точок на площині та обчислення її числових характеристик: площі і периметра.
   Опукла оболонка визначається рекурсивно:

   Convex(Ø) = Ø, де Ø – порожня множина точок;
   Convex($\{a\}$) = $\{a\}$, де a – точка площини;
   Convex($\{a, b\}$) = $\{a, b\}$, де a і b – точки площини;
   Convex($\{a_1, a_2, \ldots, a_n, a_{n+1}\}$) = Convex($\{$Convex($\{a_1, a_2, \ldots, a_n\}$), $a_{n+1}\}$), n=2,3,…, де $a_1, a_2, \ldots, a_n, a_{n+1}$ – точки площини.

Визначте ієрархії об'єктів і класів для реалізації опуклої оболонки довільної послідовності точок площини. Реалізуйте проект.

2. Компілятор арифметичних формул. Арифметичні формули – це вирази, побудовані за допомогою арифметичних операцій з числових змінних і сталих з використанням дужок для визначення порядку виконання операцій. Вони визначаються індуктивно. Змінна *x* – це формула, стала *c* – теж формула. Якщо *a* і *b* – формули, то (*a*),+*a*, –*a*, *a*+*b*, *a*–*b*, *a***b*, *a*/*b* – теж формули. На вхід калькулятора арифметичних формул надходять арифметична формула і набір значень змінних, на виході – значення цієї формули при заданому наборі значень аргументів. Визначте ієрархію об'єктів і класів для реалізації калькулятора арифметичних формул. Реалізуйте проект.
3. Квазіузагальнений стек. З метою уникнення розбухання коду замовник попросив розробника зберігати в контейнері стеку безтипові указники об'єктів `void*`. Спроектуйте абстрактний параметризований клас стеків, спираючись на клас стеку безтипових указників. Потурбуйтесь про збережність об'єктів, указники яких будуть заноситися до стеку, до моменту їх вилучення зі стеку. Захистіть стек безтипових указників від спроби його створення і використання поза інтерфейсним параметризованим класом. Обґрунтуйте обраний спосіб організації ієрархії.
4. На умовах проекту квазіузагальненого стеку (3) спроектуйте архітектуру параметризованих класів черг з пріоритетами (завдання 1, 3, 18 до розділу 6) на основі класу циклічного масиву безтипових указників. Додайте до черги операцію підглядання.
5. Виконайте рефакторинг класів банківських рахунків з підрозділу 7.3 (лістинги 7.27–7.30) з метою використання ідіоми вільного від замінщень (стабільного) невіртуального інтерфейсу.