Міністерство освіти і науки України
Житомирський державний технологічний університет

В.Ю. Вінник

Основи об'єктно-орієнтованого програмування мовою Сі++

Навчальний посібник

Друкується за рішенням Вченої Ради
Житомирського державного
технологічного університету
(протокол № 00 від 0.00.00)

Житомир
2008

УДК 004.423.2 (075)
ББК 32.973-018.1я75
В48

В48 **Вінник В.Ю.** Основи об'єктно-орієнтованого програмування мовою Сі++. — Житомир: ЖДТУ, 2007. — 000 с.

ISBN 000–000–000–000-0

Викладено основи надзвичайно важливого для практичних застосувань та загальновизнаного об'єктно-орієнтованого підходу до програмування разом з основами широко розповсюдженої мови Сі++. Від читача вимагається попереднє володіння основами програмування мовою Сі. Побудова посібника в цілому відповідає структурі лекційного курсу, що читає автор в Житомирському державному технологічному університеті. Виклад теоретичного матеріалу супроводжується прикладами програмного коду. Посібник призначений для студентів напряму підготовки ≪Комп'ютерні науки≫, які вивчають дисципліну ≪Основи програмування та алгоритмічні мови≫, а також для всіх, хто вивчає програмування самостійно.

УДК 004.423.2 (075)
ББК 32.973-018.1я75

Рецензенти:
академік НАН України В.Н.Редько,
д.т.н., професор А.В. Панішев

ISBN 000–000–000–000-0

Зміст

Вступ

Студенти, які підійшли до вивчення мови Сі++ та об'єктно-орієнтованого стилю програмування, вже мають за плечима багаж знань з мови Сі та початкове знайомство з програмуванням у структурно-модульному стилі за книгою [8]. З одного боку, даний навчальний курс і цей посібник є прямими продовженнями курсу основ програмування та попередньої книги, та й мова Сі++, що вивчається в даному курсі, виникла як вдосконалена версія мови Сі. Та з іншого боку, об'єктно-орієнтований стиль по своїй сутності надто сильно відрізняється від раніше вивченого структурно-модульного. Зокрема, сильна подібність між мовами Сі++ та Сі не повинна вводити в оману: за схожими синтаксичними формами ховається кардинальна відмінність принципів і задумів, покладених в основу цих мов. На перший погляд може здатися, що С++ — це та ж мова Сі, до якої додано кілька нових конструкцій. Однак саме ці, нові мовні конструкції і становлять саму сутність мови Сі++ і, в той же час, за своєю глибинною ідеєю докорінно не вкладаються в притаманну мові Сі парадигму. Навіть якщо дві програми, процедурна та об'єктно-орієнтована, розв'язують одну задачу, вони ґрунтуються на двох мало не протилежних світоглядних платформах, тому мають різну логіку побудови і принцип дії. В певному розумінні, починати вивчення даного курсу треба з того, щоб забути вивчений в попередньому курсі підхід до програмування. Використовуючи термінологію Гегеля, об'єктно-орієнтоване програмування є *діалектичним запереченням* структурно-модульного.

Згадка про великого філософа тут не випадкова. Можна сміливо стверджувати, що об'єктно-орієнтоване програмування немислиме без філософії і являє собою прикладну філософію. В попередньому курсі мова Сі вивчалася саме як мова, а питання глибинних першооснов, фундаментальних принципів побудови мови не піднімалися — справді, сенс та призначення основних понять та конструкцій мови Сі в тій галузі, для якої вона призначена, настільки очевидний, що потреби виносити зазначені питання на спеціальний розгляд не було. З мовою Сі++ ситуація прямо протилежна. В центрі уваги в цьому

курсі будуть не суто технічні питання (як пишеться і що робить той чи інший оператор), а питання філософські, наприклад: існують абстрактні поняття в реальності, чи вони, навпаки, є лише витворами нашої уяви? Що таке загальне та окреме, абстрактне та конкретне? Що таке поняття і як воно співвідноситься з реальними речами? Справа в тому, що конкретні конструкції мови Сі++ призначені саме для того, щоб бути програмною моделлю, машинною реалізацією цих, надзвичайно абстрактних і дуже непростих категорій. Тому без чіткого розуміння питань онтології, гносеології, методології, логіки годі й думати про написання програм мовою Сі++. Тому виклад матеріалу часто буде супроводжуватися відступами, що на перший погляд не мають прямого відношення до програмування. Студенту не слід ставитися до цих відступів легковажно — вони насправді навіть важливіші за виклад конкретних конструкцій мови Сі++, бо пояснюють сенс на призначення останніх.

При вивченні даного курсу студентам треба розрізняти загальнозначущі принципи, притаманні об'єктно-орієнтованій парадигмі як такій, та синтаксичні особливості конкретної мови Сі++. Перші не залежать від конкретної мови та знаходять своє втілення в широкому сімействі мов: Object Pascal, Eiffel, Java, CLOS (Common Lisp Object System), Smalltalk. В подальшій професійній діяльності студент може програмувати будь-якою мовою — якщо добре засвоєно загальні принципи, то прилучити до них синтаксичні правила тієї чи іншої мови буде неважко.

Студенти-початківці нерідко скаржаться, що об'єктно-орієнтоване програмування надто складне. Та насправді воно винайдене саме для того, щоб, навпаки, спростити роботу практика-програміста. Може виявитися складно глибоко зрозуміти та навчитися застосовувати методологію ООП, але коли цей рубіж перейдено, писати прогами для конкретних практичних задач стає значно простіше, ніж розв'язувати ті ж задачі в структурно-модульному стилі. Можна припустити, що загальна складність структурно-модульного та об'єктно-орієнтованого стилю приблизно однакові. Але якщо у структурно-модульному програмуванні велика частина складності переноситься на кожну окрему задачу, то в ООП значна частина складності припадає на початкове вивчення загальної методології, тоді як подальше застосування її до сотень реальних

задач відповідно спрощується. Тому зусилля, які треба витратити на вивчення ООП, — це інвестиції, які швидко окуповуються.

Таким чином, цей посібник вирішує дві задачі: основну — навчити загальним принципам об'єктно-орієнтованого програмування та підпорядковану їй — навчити мові Сі++ (одній з багатьох об'єктно-орієнтованих мов), яка є конкретним втіленням цих принципів. Оскільки обидва зазначені предмети доволі складні і дуже об'ємні, для посібника обрано певний фрагмент ООП та Сі++, з одного боку достатній для широкого кола практичний задач, з другого боку достатньо показовий, щоб дати студентам цілісне уявлення про предмет, а з третього — по простоті наближений до можливостей студентів першого року навчання.

Прийнятий в цьому посібнику метод викладу матеріалу часто спирається на згадані вище діалектичні заперечення. Зокрема, нові засоби об'єктно-орієнтованої мови Сі++ будемо розглядати у протиставленні з відомими засобами мови Сі. Для того чи іншого відомого з мови Сі механізму наводяться приклади, що показують йоно недоліки (незручність, громіздкість, ненадійність). Це дозволяє побачити в загостреному вигляді проблему — брак у мові Сі належної підтримки для тих чи інших практично значущих задач. Після цього засоби мови Сі++ зручно описувати як природне вирішення виявленої проблеми.

Посібник умовно ділиться на дві частини. В перших двох главах висвітлюються насамперед фундаментальні питання об'єктно-орієнтованої парадигми, по відношенню до яких конкретні засоби мови Сі++ відіграють підпорядковану, ілюстративну роль. В решті глав розповідається про засоби мови Сі++, які вже не мають парадигмального, світоглядного значення, але є важливими для практичних застосувань інструментами. Далі у цьому вступі описано порівняно дрібні особливості мови Сі++, які не становлять докорінного зламу парадигми, а лише додають маленькі зручності до відомих з попереднього курсу засобів мови Сі і необхідні для розуміння матеріалу перших двох глав.

0.1. Однорядкові коментарі

В мові Сі є лише одна форма написання коментарів: в дужках /* та */. Це зручно для великих за обсягом коментарів, що простягаються на кілька рідків. Але, якщо потрібно записати короткий коментар з одного-двох слів, цей спосіб позначення виявляється надто громіздким. Тому в мову Сі++ запроваджено однорядкові коментарі. Однорядковий коментар починається з символів // та простягається до кінця рядка. Звичайно ж, коментарі у «старому» стилі Сі також можна використовувати в програмах мовою Сі++. Нижче наведено програмний текст, в якому використовуються обидва стилі коментарів.

```
/* приклад програми на Сі++
з коментарями різних стилів */
int main( void ) {
  int x = 0; // код повернення
  return x; // завершення нормальне
}
```

0.2. Використання імен структурних типів

Нехай дано оголошення структурного типу даних, наприклад:

```
struct tPoint {
    double x;
    double y;
};
```

В мові Сі, як вивчалося у попередньому матеріалі, повне ім'я структурного типу складається з двох слів: ключового слова **struct** та власне імені. Слово **struct**, згідно правил мови Сі, необхідно вписувати при кожному використанні структурного типу: оголошення змінної s наведеного структурного типу повинно мати вигляд

```
struct tPoint s;
```

Мова Сі++ дозволяє не вказувати слово **struct** при використанні структурного типу[1]. Оголошення змінної s мовою Сі++ набуває ви-

[1] Це правило справедливе і для типів об'єднань **union** та перерахувань **enum**, які в курсі основ програмування не вивчалися

гляду

```
tPoint s;
```

0.3. Оголошення змінних посеред операторів

В мові Сі оголошення всіх локальних змінних повинні стояти лише перед операторами. Якщо функція велика і складна, то між оголошенням змінної та її використанням в тілі функції може бути десяток чи більше рядків, що незручно для сприйняття тексту людиною. Тому іноді буває зручніше оголошувати змінну по місцю потреби. Мова Сі++ це дозволяє: оголошення змінних можуть довільно чергуватися з операторами, або навіть і стояти всередині оператору. Наприклад, наведений нижче рядок був би неправильним з точки зору мови Сі, але цілком нормальний для мови Сі++:

```
for( int i = 0; i < 10; i++ )
  printf( "*" );
```

0.4. Перевантаження імен функцій

В мові Сі кожна функція повинна мати неповторне ім'я: програма не може містити двох різних функцій з однаковими іменами.

Але ж на практиці часто трапляються сімейства функцій, дуже схожих між собою за призначенням, які виконують однакову за сенсом роботу з даними різних типів. Наприклад, програмісту може знадобитися сімейство функцій виведення, до якого увійдуть функція друку цілого числа, функція друку дійсного числа, функція друку рядка, функція друку масиву цілих чисел, тощо. Оскільки ці функції дуже схожі за призначенням, хотілося б, щоб їх імена теж були схожі.

Єдиним можливим рішенням у мові Сі було б дати цим фнкціям імена, що складаються з однакового префіксу, наприклад `print`, та суфіксу, свого для кожної функції:

```
void print_int( int x );
void print_double( double x );
void print_string( char *x );
```

```
void print_intarray( int n, int *x );
```

Мова Сі++ пропонує зручніший спосіб. Наведені функції мовою Сі++ можна переписати так:

```
void print( int x ) { // друк цілого числа
printf( "%d", x );
}
void print( double x ) { // друк дійсного числа
printf( "%lf", x );
}
void print( char *x ) { // друк рядка
puts( x );
}
void print( int n, int *x ) { // друк масиву
for( int i = 0; i < n; i++ )
    printf( "%5d", x[i] );
}
```

Функції в мові Сі++ можуть мати однакові імена, якщо вони відрізняються кількістю або типами аргументів. Тоді одне ім'я може позначати не одну, а кілька функцій, такі імена називаються *перевантаженими*[1].

В наведеному прикладі перевантаженим є ім'я `print`: воно позначає і функцію з одним аргументом типу **int**, і функції з аргументом типу **double** та **char***, і функцію з двома аргументами — цілим та покажчиком на ціле.

Хоча ці функції є «тезками», компілятор завжди може без плутанини розібратися, яку функцію треба викликати — для цього достатньо подивитися на типи аргументів, які передаються у функцію. Наприклад, нехай дано такий текст:

```
1 int t = 8;
2 int arr[4] = { 32, 45, 9, 27 };
3 print( (t + 23) * 3 );
4 print( sin(3.14 / 6) );
5 print( "Слов'яни" );
6 print( 4, arr );
```

[1] В деяких книгах можна побачити словосполучення «перевантаження функцій». На переконання автора, воно безграмотне: можна говорити лише про перевантаження *імен* функцій

```
7 print( "Hello", 3.14 );
```

В рядку, відміченому цифрою 3, компілятор спершу визначить, що вираз `(t+23)*3` дає значення типу **int**, а тоді серед усіх функцій з ім'ям `print` знайде ту, що найбільш підходить — це буде функція `print(int x)` друку цілого числа. В рядку, позначеному цифрою 4, компілятор спершу дізнається тип значення, яке повертає функція `sin`, зрозуміє, що функції `print` передається аргумент типу **double**, і отже з усіх функцій з іменем `print` обере таку: `print(double x)`. Так само розпізнається виклик в рядках з цифрами 5 і 6. Нарешті, в рядку 5 компілятор не зможе знайти в сімействі функцій `print` таку, яка б мала аргументи типів **char*** та **double**, та повідомить про помилку.

Треба звернути особливу увагу на те, що функції з однаковими іменами компілятор розрізняє лише за типами і кількістю аргументів. Типи значень, які повертають функції з однаковими іменами, можуть відрізнятися, але не можуть бути *єдиною* відмінністю між цими функціями — обов'язково повинні відрізнятися аргументи. Наприклад, у наведеному нижче тексті компілятор зафіксує помилку:

```
int func( char *s );
double func( char *s );
void* func( char *s );
```

Сенс та причину цього правила легко зрозуміти, якщо уявити собі виклик такої функції

```
func( "Дніпро" );
```

та поставити себе на місце компілятора: з наведеного рядка жодним способом неможливо з'ясувати, яка з трьох функцій з іменем `func` мається на увазі. Отже, щоб уникнути таких ситуацій, в мові Сі++ забороняються функції, які мають однакові імена та однакові аргументи, навіть якщо в них відрізняються типи значень.

0.5. Значення по замовчуванню для аргументів

Часто на практиці виникає ситуація, коли деяка функція, що має доволі багато аргументів, викликається з різних місць програми десятки разів і при цьому в абсолютній більшості викликів частині аргументів передаються щоразу ті самі значення. В мові Сі немає іншого

виходу, як щоразу вписувати при виклику значення всіх аргументів. Мова Сі++ дозволяє зробити текст програми лаконічнішим, а працю програміста менш рутинною, приписавши частині аргументів чи навіть всім аргументам функції *значення по замовчуванню*.

Якщо аргументу функції один раз, при її оголошенні, призначити значення по замовчуванню, то всюди далі при виклику функції цей аргумент можна не вказувати — в такому випадку компілятор автоматично підставить значення по замовчуванню. Лише в тих випадках, коли програміст бажає передати аргументу інше значення, відмінне від значення по замовчуванню, потрібно вказати це значення в явному вгляді.

Оголошення функції, у якої частина аргументів має значення по замовчуванню, має вигляд

```
тип функ(тип1 арг1,..., типN аргN=значN,...);
```

Тут `тип`, як звичайно, це тип значення, яке функція повертає, `функ` — ім'я функції, `тип1 арг1` — оголошення звичайного аргументу (без значення по замовчуванню), `типN аргN = значN` — оголошення аргументу з ім'ям `аргN` типу `типN`, якому призначається значення по замовчуванню `значN`. Функція може мати скільки завгодно звичайних аргументів та скільки завгодно аргументів зі значеннями по замовчуванню.

Всі аргументи зі значеннями по замовчуванню можуть оголошуватися лише після усіх звичайних аргументів: інакше компілятор не зможе розібратися при виклику, яке значення якому аргументу належить. Значеннями по замовчуванню можуть бути лише константи (або вирази, які компілятор може звести до констант). Це правило легко зрозуміти: оголошення функції, а отже оголошення аргументів зі значеннями по замовчуванню, обробляється не під час виконання програми, а під час її компіляції. Тому значення аргументів по замовчуванню повинні бути відомі заздалегідь, їх обчислення не повинно залежати від виконання програми.

Проілюструємо сказане прикладом. Нехай програміст пише функцію `log_message`, у якої першим аргументом є текстовий рядок, а другим — ціле число: якщо воно дорівнює 0, то повідомлення треба записати у файл, а якщо 1, то вивести на екран:

```
void log_message( char *text, int where );
```

Нехай програміст хоче направляти майже всі повідомлення до файлу і лише деякі (можливо, повідомлення про найкритичніші, найнебезпечніші помилки) одразу показувати на екран. Тоді в абсолютній більшості викликів цієї функції значенням другого аргументу буде 0, наприклад

```
log_message( "Система␣рівнянь␣вироджена", 0 );
```

і лише в декількох викликах другий аргумент матиме значення 1:

```
log_message( "Нестача␣пам'яті", 1 );
```

Звичайно ж, було б зручно другому аргументу **int** `where` присвоїти значення по замовчуванню 0:

```
void log_message( char *text, int where = 0 );
```

Тепер з двох наведених вище випадків виклику цієї функції перший перетворюється таким чином:

```
log_message( "Система␣рівнянь␣вироджена" );
```

Оскільки вказано значення лише першого аргументу, то компілятор зрозуміє, що для другого потрібно використати значення по замовчуванню 0.

Таким чином, аргумент зі значенням по замовчуванню працює за принципом «якщо явно не вказано інше, то вважати, що значенням аргументу є те, що було вказано в оголошенні».

0.6. Введення та виведення

Мова Сі має потужні засоби введення з клавіатури та виведення на екран, передусім функції `scanf` та `printf`. Ці функції дозволяють виводити дані в як завгодно складному оформленні і так само накладати будь-які умови на формат введених даних. Разом з тим, така надзвичайна гнучкість засобів введення та виведення має оборотною стороною їх високу складність — повний опис синтаксису та семантики форматних рядків займає близько 10 книжкових сторінок.

Ще одним принциповим недоліком вивчених раніше функцій введення та виведення є неможливість контролювати при трансляції типи та кількість аргументів. Ще раз нагадаємо типові серйозні помилки, яких легко може припуститися людина через неуважність, але які не помічає транслятор:

```
double x;
scanf( "%lf", x ); // пропущено & перед x
scanf( "%lf␣%d", &x ); // зайвий специфікатор
printf( "%d", x ); // %d не відповідає double
```

Мова Сі++ містить більш досконалі засоби введення та виведення, які, з одного боку, не менш потужні, ніж функції `scanf` та `printf` (будь-який формат введенни чи виведення, який можна виразити за допомогою форматних рядків, можливо реализувати і за допомогою нових засобів), з другого — набагато простіші у вивченні та застосуванні, а з третього — вільні від згаданих «лазійок для помилок», оскільки весь контроль типів аргументів покладається на транслятор.

В цьому ознайомчому розділі неможливо вичерпно описати засоби введення-виведення мови Сі++ (для цього потрібно спершу вивчити класи, об'єкти та перевантажені операції), тому обмежимося низкою найпростіших часткових випадків.

Стандартні засоби введення-виведення мови Сі++ зібрані в заголовочному файлі `iostream`[1]. Введення та виведення здійснюються через стандартні потоки, відповідно `cin` та `cout`. Щоб зробити ці об'єкти-потоки доступними, напочатку програми (як правило, після директив включення) треба написати рядок[2]

```
using namespace std;
```

Семантику цього рядка залишимо без пояснень.

Нижче наведено приклад введення значення однієї змінної та кількох змінних:

```
int x, w;
double a, b;
cin >> x; // введення однієї змінної
cin >> a >> w >> b; // одразу трьох
```

Наступний приклад показує, як використовувати потік виведення:

[1] Згідно чинного міжнародного стандарту мови Сі++, заголовочні файли не мають розширення `.h`; його вимагають лише старі компілятори, розроблені до введення цього стандарту.

[2] Це стосується лише компіляторів, які підтримують новий стандарт, коли заголовочні файли не мають розширення.

```
int zt = 884;
double a = 3.14 / 6;
cout << "Житомир␣" << zt << "␣р." << endl;
cout << "cos(30deg)=" << cos(a) << endl;
```

Після слова `cin`, яке позначає стандартний потік введення, можна скільки завгодно разів використовувати операцію введення з потоку `>>`, після якої має стояти ім'я змінної (або вираз, який до нього зводиться). Поки що будемо вводити таким чином лише змінні стандартних типів (**`int`**, **`double`**, **`char*`**, тощо), в розділі 5 буде пояснено, як «навчити» потік `cin` обробляти значення типів, створених програмістом (скажімо, структурних). Позначення операції `>>` нагадує стрілку, яка показує напрямок передачі даних — з потоку в змінні.

З потоком виведення `cout` застосовується операція `<<`, після якої може стояти вираз будь-якого стандартного типу (а в розділі 5 покажемо, як розширити цю операцію на власноруч створені типи). Позначення операції, схоже на стрілку, вказує напрямок передачі даних — значення виразів одне за одним йдуть у потік. Словом `endl` в мові Сі++ позначено спеціальний об'єкт «кінець рядка» (англ. end of line): потрапляння цього об'єкту в потік відображається на екрані як переведення курсора на новий рядок (звичайно ж, ніхто не забороняє користуватися й символом `\n` у рядкових константах, проте позначення `endl` видається наочнішим).

Абсолютно неприпустимо говорити, що `cin` та `cout` — це *операції* або, ще гірше, *оператори* введення та виведення. Це груба помилка, якої, на жаль, нерідко припускаються студенти. В главі 5 буде роз'яснено, що `cin` та `cout` це *об'єкти* певних класів, операціями тут є `>>` та `<<`. Більше того, буде роз'яснено, як власноруч створювати класи з подібними властивостями.

0.7. Динамічна пам'ять

В мові Сі є досить потужні засоби управління динамічною пам'яттю — це передусім функції `malloc` та `free`. Але вони надто громіздкі та мають низку недоліків. Перш ніж викладати нові, більш досконалі засоби управління пам'яттю у мові Сі++, згадаємо ще раз виділення пам'яті у мові Сі, щоб виявити ті недоліки, для боротьби з якими і запроваджуються нові механізми. Типовий приклад роботи

з динамічною пам'яттю у мові Сі наведено нижче.

```
int n = 10; /* кількість елементів */
double *p;
p = (double*) malloc( n * sizeof(double) );
```

Перше, що впадає в око — це необхідність щоразу приводити тип значення, яке повертає функція `malloc`, до конкретного типу покажчика — в нашому прикладі це запис `(double*)` перед іменем функції. Це ще не проблема, але вже технічна незручність, яка обтяжує програміста.

Суттєвіше те, що функція `malloc`, нічого не знаючи про тип даних, які будуть зберігатися в пам'яті, що виділяється, працює в масштабі байта. Аргументом функції є не кількість елементів масиву певного типу, а кількість байтів, які треба виділити. Це вже може спонукати програміста до помилок. Справді, уявімо, що програміст через неуважність напише

```
p = (double*) malloc( n * sizeof(int) );
```

Число типу **int** займає в пам'яті менше місця, ніж число типу **double**. Очевидно, програміст в попередньому рядку мав на меті виділити пам'ять для масиву з `n` дійсних чисел, і в подальшому тексті програми буде працювати з покажчиком `p` так, ніби він вказує саме на такий масив, але насправді пам'яті виділяється значно менше.

Більш того, оскільки функція `malloc` вимагає лише одного аргументу цілого типу, та оскільки її не цікавить, звідки береться його значення, можна уявити і такий, зовсім потворний спосіб використання функції:

```
p = (double*) malloc( 11 );
```

Для правильного створення масиву значення аргумента функції `malloc` повинно обчислюватися саме як добуток розміру одного елемента на кількість елементів. В даному ж випадку передається фіксоване число. Число типу **double** займає (принаймні на процесорі автора) 8 байтів, отже виходить, що наведений оператор виділяє пам'ять під масив з $1\frac{3}{8}$ числа!

Узагальнюючи, можна стверджувати, що причини цих та інших незручностей функції `malloc` такі:

1. Функція нічого не знає про тип даних, для яких виділяється пам'ять. Фнкція працює з «просто пам'яттю», а не з пам'яттю, призначеною для даних того чи іншого типу.
2. Функція працює в масштабі байта, а не елемента.
3. Функція не захищає програміста від таких способів використання функції, які можуть призвести до помилок. Надто багато свободи означає і максимум можливостей для прикрих випадковостей.

Щоб подолати вказані труднощі, в мові Сі++ для управління динамічною пам'яттю існують операції[1] **new** (виділення пам'яті, створення нового об'єкту) та **delete** (звільнення пам'яті, знищення динамічного об'єкту).

У операції **new** є кілька різних способів застосування. Зараз вивчимо два найпростіших — виділення пам'яті під одинокий об'єкт та під масив об'єктів, де об'єкти розуміються в сенсі мови Сі, не зачіпаючи питань об'єктно-орієнтованого програмування. Коли будуть вивчені поняття класу, конструктора та деструктора, можна буде викласти застосування операції **new** до створення об'єктів класів з одночасним викликом конструкторів, див. розділ 1.7.

Нехай `тип` — деякий тип даних, допустимий в мові Сі (тобто не клас у сенсі об'єктно-орієнтованого програмування), такий як **int**, **char** чи структурний. Тоді типовий спосіб застосування операції **new** для динамічного створення одного об'єкту цього типу виглядає так:

```
тип *p; // оголосити змінну-покажчик
p = new тип; // створити об'єкт
```

Розберемо останній рядок детально. Вираз **new** `тип` власне створює в динамічній пам'яті один об'єкт типу `тип`, тобто виділяє в динамічній пам'яті область такого розміру, як потрібно для даного типу, та повертає адресу «новонародженого» об'єкту. В в більшості випадків цей вираз стоїть у правій частині оператору присвоювання, але в принципі він може стояти в програмі будь-де, де взагалі допускається вираз типу `тип*` (наприклад, як значення аргументу при виклику функції).

[1] Звертаємо увагу: це не функції, а саме операції

Після операції присвоювання адреса створеного об'єкту, яку повернула операція **new**, потрапляє у змінну `p`, отже надалі з об'єктом можна працювати через цей покажчик.

Випишемо ще раз синтаксичні та семантичні правила найпростішого способу застосування операції **new**:

- слугує для динамічного створення одного об'єкту певного типу;
- пишеться: слово **new**, одразу за яким ім'я типу даних;
- виділяє в динамічній пам'яті область потрібного розміру для зберігання об'єкту цього типу;
- повертає покажчик на цей об'єкт, причому покажчик вже приведений до потрібного типу.

Для того, щоб динамічно створити масив з певної кількості елементів деякого типу, треба скористатися такою формою операції **new**:

```
new тип[ кількість ]
```

де `кількість` може бути задана довільним виразом цілого типу. Операція **new** сама, знаючи тип елементу та їх кількість, обчислює розмір потрібної області пам'яті, виділяє область та повертає покажчик на неї, приведений до типу `тип*`.

Для видалення динамічно створених об'єктів і звільнення пам'яті, яку вони займають, слугує операція **delete**. Якщо `покажчик` — це деякий вираз, значенням якого є покажчик на один динамічно створений об'єкт, то операція знищення цього об'єкту має вигляд

```
delete покажчик;
```

Якщо `покажчик` — це вираз, значенням якого є покажчик на динамічно створений масив об'єктів, то операція знищення цього масиву має вигляд[1].

```
delete[] покажчик;
```

Наприклад, наведений нижче фрагмент створює масив дійсних чисел, розмір якого програмі заздалегідь невідомий та вводиться користувачем, а після обробки знищує його:

[1] В старих версіях мови Сі++ вимагалося, щоб в квадратних дужках вказувалася ще й кількість елементів масиву. В новому стандарті вважається, що менеджер пам'яті сам може розібратися, скільки елементів в масиві

```
int n;
double *p;
cout << "Введіть␣розмірність␣масиву␣" << endl;
cin >> n;
p = new double[n];
// далі звичайна робота з масивом
. . .
// обробку закінчено, звільнити пам'ять
delete[] p;
```

Як видно, записати динамічне створення масиву мовою Сі++ набагато коротше та простіше, ніж мовою Сі. До речі, чим коротший запис конструкції, яка розв'язує ту чи іншу задачу, тим менше в ньому можливостей зробити помилку.

Наостанок нагадаємо, що в розділах 1.7 та 1.8 будуть додатково описані особливості застосування операцій **new** та **delete** до об'єктів класів.

0.8. Посилання

Нагадаємо важливі факти, добре відомі з курсу програмування мовою Сі. В мові Сі аргументи передаються до функцій по значенню, і повертає функція також деякі значення. Завдяки операції взяття адреси таким значенням може виступати й покажчик на деяку змінну, що дозволяє функціям присвоювати значення локальним змінним іншої функції: наприклад, функція `main`, яка має локальну змінну `x`, може викликати функцію `f` та в якості аргументу передати їй покажчик на змінну `x`; функція `f`, знаючи покажчик, може занести по цій адресі нове значення:

```
1 void f( int *p ) { // аргумент-покажчик
2   *p = 0; // нелокальне присвоювання
3 }
4
5 int main( void ) {
6   int x = 1;
7   cout << "до:␣" << x << endl;
8   f( &x ); // передати адресу
9   cout << "після:␣" << x << endl;
```

```
10 }
```

Проте згадані засоби мови Сі дещо громіздкі та незручні: при виклику функції `f` треба при її аргументі писати операцію взяття адреси `&`, а в тілі функції `f` для роботи зі змінною, покажчик на яку передано, потрібна операція розіменування `*`.

Мова Сі++ містить більш досконалий спосіб передачі аргументів до функцій та результатів з функцій — по *посиланню*. З точки зору реалізації він такий само, як і щойно описаний спосіб передачі адрес, але за формою написання набагато зручніший, бо ні при виклику функції для передачі змінної в якості аргументу (рядок 8), ні в її тілі для роботи зі змінною-аргументом (рядок 2) не потрібно писати жодних операцій. Покажемо, як перетворюється попередній програмний текст, якщо передачу аргументу по покажчику замінити на передачу по посиланню.

```
1  void f( int &p ) { // аргумент-посилання
2    p = 0; // нове значння присвоюється посиланню
3  }
4
5  int main( void ) {
6    int x = 1;
7    cout << "до:␣" << x << endl;
8    f( x ); // насправді передається адреса!
9    cout << "після:␣" << x << endl;
10   return 0;
11 }
```

Нова форма оголошення аргументу, показана в першому рядку цього прикладу, це і є оголошення аргумента-посилання. Як видно з рядку 2, з аргументом-посиланням можна працювати так само, як і зі звичайною змінною. Але насправді змінна-посилання `p` містить в собі не копію значення змінної `x`, а її адресу, тому значення присвоюється саме в змінну `x` функції `main`.

Таким чином, посилання поводять себе як покажчики, до яких компеілятор сам «дописує» операцію розіменування.

Посилання може виступати не лише аргументом, але й значенням функції. Розглянемо приклад. У наступній програмі оголошено дві глобальні змінні. Функція `h` повертає посилання на одну з них в залежності від того, чи парний її аргумент.

```
1  int a = 0, b = 0; // глобальні
2
3  int &h( int k ) {
4    if( k % 2 ) return a;
5    return b;
6  }
7
8  int main( void ) {
9    int x;
10   cin >> x;
11   h(x) = 1;
12   cout << a << "\t" << b << endl;
13   return 0;
14 }
```

Оскільки функція повертає не ціле число, а посилання на цілочисельну змінну, то вираз `f(x)` може стояти в лівій частині оператора присвоювання, як в рядку 11 цього прикладу. Якщо користувач вводить непарне число, то функція `f` повертає посилання на змінну `a`, а якщо неперне — посилання на змінну `b`. В ту змінну, посилання на яке повертає функція, і присвоюється нове значення 1. Операції виведення в наступному рядку дозволяють переконатися, що одній з глобальних змінних, та якій саме, присвоїлося нове значення.

Так само, як і аргументи чи значення типу покажчика в мові Сі, посилання можна використовувати не лише для нелокальних присвоювань, але й для того, щоб швидко і без зайвих витрат пам'яті передавати великі об'єкти: для передачі такого об'єкта по значенню потрібно зробити тимчасову копію всього його вмісту, а для передачі по посиланню достатньо передати лише адресу, яка займає всього кілька байтів. У тих випадках, коли функція не має наміру змінювати значення об'єкту, посилання на який отримала через аргумент (тобто не збирається робити з ним нелокальне присвоювання), варто позначити такий аргумент як константне посилання: скажімо, якщо дано структурний тип `TPoint` (моделює точку на площині, задану координатами x, y), функція, що вираховує відстань між двома точками, могла б мати прототип

```
double distance( const TPoint&, const TPoint& );
```

Насамкінець треба звернути увагу на форму запису. Як видно з наведених вище прикладів, посилання позначається знаком `&`, що ставиться перед іменем аргумента, який має приймати посилання, чи перед іменем функції, яка повертає посилання. Іноді студентів вводить в оману те, що той же знак перед іменем змінної чи функції позначає також і операцію взяття адреси. Та насправді легко зрозуміти різницю між цими двома ролями, в яких виступає знак `&`, і ніколи їх не плутати між собою: він означає взяття адреси, коли стоїть у *виразі*, коли в контексті від нього очікується певне *значення*. Наприклад, в операторі присвоювання `q=&y` очевидно, що результатом обчислення правої частини стає конкретне значення — адреса змінної `y`. Натомість контекст `int &g(int &x)` є не виразом, від якого очікується обчислення якогось результату, а оголошенням функції `g`, а його складовою частиною є оголошення аргументу `x`, тому тут знак `&` обидва рази означає посилання.

Розділ 1

Об'єкти та класи

1.1. Найпростіший приклад

Розглянемо, як типова задача розв'язується мовою Сі та притаманним їй стилем, а потім порівняємо з розв'язком мовою Сі++ в об'єктно-орієнтованому стилі. Нехай потрібно в програмі обробляти дані про книги. Кожна книга характеризується набором параметрів (назва і автор). Треба забезпечити можливість вводити та виводити дані про певну книгу. Нижче наведено текст у структурно-модульному стилі мовою Сі.

```
1  typedef struct tagBook {
2    char title[40];
3    char author[40];
4  } TBook;
5  void inputBook( TBook* );
6  void printBook( TBook* );
7  void inputBook( TBook *p ) {
8    printf( "Назва, автор, рік, кількість стор.?\n" );
9    gets( p->title );
10   gets( p->author );
11 }
12 void printBook( TBook *p ) {
13   printf( "%40s %40s\n", p->title, p->author );
14 }
15 int main( void ) {
16   TBook b1;
17   inputBook( &b1 );
18   printBook( &b1 );
19   return 0;
20 }
```

Немає потреби детально пояснювати цей текст. Параметри кожної книги (назва і автор) зберігаються в структурних об'єктах. Відповідний структурний тип означений в рядках 1–4. Функція `inputBook` читає з клавіатури (чи іншого пристрою введення) всі дані про книгу та заносить їх в структурний об'єкт, покажчик на який передано функції в якості аргументу. Функція `printBook` виводить

дані про книгу зі структурного об'єкту, переданого знову ж через покажчик. Функції `setBookTitle` та `setBookAuthor` дозволяють встановити структурному об'єкту певні значення параметрів "назва"та "автор". Наведена наприкінці програми функція `main` ілюструє застосування всіх цих функцій. Для цього оголошуються дві змінні, `b1` та `b2`, в яких містяться об'єкти структурного типу. До цих об'єктів застосовуються описані вище функції, для чого адреси цих об'єктів передаються до функцій в якості аргументів.

Тепер подивимось, як цю ж задачу розв'язують в об'єктно-орієнтованому стилі мовою Сі++.

```
1  class CBook {
2  public:labellst:public
3    void input( void );
4    void print( void );
5  private:
6    char m_author[40];
7    char m_title[40];
8  };
9  void CBook::input( void ) {
10   printf( "Назва, автор, рік, кількість стор.?\n" );
11   gets( title );
12   gets( author );
13 }
14 void CBook::print( void ) {
15   printf( "%40s %40s\n", title, author );
16 }
17 int main( void ) {
18   CBook b1;
19   b1.input();
20   b1.print();
21   return 0;
22 }
```

Оголошення в рядках 1–8 — це оголошення *класу*, який моделює в програмі поняття «книга». Так само, як і в попередній програмі, клас це створений програмістом власний тип даних. Подібно до структурного типу з попереднього лістингу, членами цього класу є дані «назва» та «автор». Але членами класу, на відміну від структурного типу, є також і функції. Функції, що є членами класу, прийнято називати *методами*. Щодо слів **public** та **private** в оголошенні класу, то їх роз'яснення відкладемо на потім (розділ 1.3).

Якщо порівняти реалізацію функцій (наприклад, функції введення, рядки 7–11 першого лістингу та 9–13 другого лістингу), помітна відмінність форми запису. Як видно з рядку 9 другого лістингу, при означенні тіла методу треба вказати не лише власне ім'я методу, але й «по батькові», тобто ім'я класу, до якого він належить:

```
ім'я_класу::ім'я_методу
```

Оголошення змінних, які відповідають конкретним книгам в об'єктно-орієнтованому стилі (рядок 18 другого лістингу) та в структурно-модульному (рядок 16 першого прикладу) на вигляд не відрізняються. Проте в першій програмі значеннями змінних `b1` та `b2` стають об'єкти даних структурного типу, а в другій — об'єкти класу. Об'єкт структурного типу містить в собі члени-дані (як їх ще називають, *поля*), описані в оголошенні типу, а членами об'єкта класу є також і описані у класі методи.

Уважний читач може помітити, що якщо в структурно-модульному стилі кожна функція для обробки книги мала аргумент типу покажчика на структуру даних то у відповідних методів такого аргументу немає. Змін зазнав і спосіб виклику функцій (рядки 17 та 19 відповідно): в першій програмі покажчик на структуру даних передається до функції в якості аргументу, а в другій натомість застосовано позначення вигляду

```
об'єкт.метод(аргументи)
```

Цей запис підкреслює, що викликається функція, яка є складовою частиною об'єкту (схожим чином в мові Сі позначається звертання до члену структурного об'єкту).

1.2. Об'єкти як відображення речей

Від розбору одиничного прикладу перейдемо до формулювання загальних правил і принципів ООП. При цьому варто не лише вивчити самі по собі означення і правила, але й зрозуміти їх сутність, походження і світоглядні підвалини. Вивчаючи той чи інший стиль програмування, треба завжди виходити з того, що програмування — це моделювання дійсності. Тому гарна мова прогармування повинна не затьмарювати, а якнайкраще відображати сутність задачі. В ідеалі мова програмування повинна підтримувати розмірковування

людини над задачею, щоб написання програмного тексту було прямим перекладом роздумів програміста, його внутрішнього монологу про задачу. Відповідно, і при вивченні ООП треба виходити з того, що сама дійсність має деякі риси і властивості, для яких «об'єкти», «класи», «поліморфізм» становлять якнайкращу програмну модель і є їх прямим відображенням[1].

Людині притаманно бачити світ як сукупність речей, предметів. Предмети, які ми бачимо навколо себе, добре відокремлені один від одного, мають чіткі границі. Іншими словами, ми можемо впевнено розрізняти, де той, а де інший предмет, а також виділити певний предмет серед інших. Ця винятково важлива риса дійсності відображається у стилі програмування: програма в ООП складається з *об'єктів* — окремих програмних одиниць, які так само можна впевнено відокремити одну від одної. В теорії ООП про це прийнято говорити, що *об'єкт наділений індивідуальністю*. Ймовірно, ООП не змогли б винайти істоти, які живуть в світі примарних розпливчастих хмар, що плавно, без виражених границь, перетікають одна в одну.

Для порівняння: програма в структурно-модульному стилі складається з підпрограм (у мові Сі — функцій), що реалізують деякі алгоритми, тобто процеси обробки даних. Але ж наша дійсність складається не з процесів, а з речей! Тому моделювати світ структурно-модульною програмою неодмінно означає спотворювати його сутність. Між іншим, такий стиль програмування був би найзручнішим для згаданих вище уявних істот, для яких немає предметів, а є лише безперервні процеси у суцільному розпливчастому середовищі.

Продовжимо з'ясовувати важливі риси дійсності і пов'язувати їх з принципами ООП. В кожен момент часу кожна річ в звичному нам світі має певний *стан*, що характеризується набором *параметрів*: наприклад, для м'яча це три просторові координати, швидкість, маса тощо. Далі, кожна річ час від часу зазнає впливів навколишнього середовища, тобто середовище може здійснювати над річчю деякі *операції*, і на будь-який вплив річ певним чином реагує (зокрема,

[1] Попри те, що об'єктно-орієнтований підхід іноді здається студентам надто складним, насправді він винайдений для того, щоб *спростити* роботу програміста. Це гарний критерій для перевірки глибини своїх знань: вивчення ООП тоді можна вважати вдалим, коли його поняття та принципи починають сприйматися як самоочевидні, або коли стає незрозуміло, як можна було програмувати в інший спосіб.

змінює свій стан) — в цьому полягає її *поведінка*. Скажімо, над м'ячем можна виконувати операцію удару з певною силою і з певним кутом, результатом якої стає зміна швидкості та координат м'яча.

Важливо, що стан та поведінка речі утворюють єдине ціле, в реальному світі вони не існують одна без одної. Так, з одного боку, будь-яка поведінка ніби надбудована над станом: реакція речі на зовнішній вплив полягає у зміні стану (наприклад, удар змінює швидкість та координати м'яча). З іншого боку, поточний стан речі може проявитися назовні лише через деяку поведінку —коли річ взаємодіє з навколишнім середовищем: скажімо, щоб виміряти координати чи температуру речі, треба прикласти лінійку чи термометр[1]

Об'єкт в ООП — це програмна одиниця, яка поєднує в собі дані та функції по обробці цих даних. Зрозуміло, що дані об'єкта відображають стан якоїсь речі, а функції (у традиційній для ООП термінології — методи) це дії, яких зазнає об'єкт з боку навколишнього середовища. Таким чином, об'єкт є очевидною програмною моделлю реальної речі з її станом та поведінкою. Наприклад, у попередньому розділі розглянуто об'єкт, що моделює книгу: він містить дані про автора та назву і операції введення та друку. Об'єкт, що моделює прямокутник, містить дані — координати лівого нижнього кута, ширина, довжина, і операції — переміщення, збільшення чи зменшення розміру тощо. Об'єкт, що моделює вектор, містить дані про розмірність і кожен компонент, і операції обчислення довжини, скалярного добутку та ін.

Спосіб організації програми, при якому дані та функції по їх обробці об'єднуються в спільну програмну одиницю, називається *інкапсуляцією*. Цей термін дослівно означає «вкладання у капсулу» і підкреслює, що дані і функції розташовані в об'єкті ніби в спільній оболонці. Інкапсуляція добре підтримує зазначену вище нерозривну єдність стану та поведінки речей — на відміну від структурно-модульного стилю, де структури даних (модель стану) і функції (модель поведінки) сильно відокремлені одне від одного (див. приклад з

[1] Якщо річ зовсім не взаємодіє з середовищем, неможливо виміряти жоден з параметрів її стану. За термінологією І.Канта це *річ у собі*. Навіть якщо така річ і має якийсь стан, про цей стан нічого неможливо сказати чи дізнатися, а це все одно, що стану взагалі немає — а отже немає і самої речі. Справді, чи багато користі від сейфу з золотом, якщо його неможливо ні побачити, ні намацати, ні відкрити? Напевно, не більше, ніж коли сейфу немає взагалі.

попереднього розділу, де означення структурного типу `TBook` та оголошення функцій для роботи з ним не пов'язані між собою у тексті програми).

1.3. Приховування реалізації

Продовжимо розглядати реальну річ, яка має власний стан і взаємодіє з іншими речами. Дуже важлива риса дійсності полягає в тому, що зовнішній світ впливає на параметри стану речі не безпосередньо, а за допомогою деякої проміжної ланки. Зовнішній світ діє на проміжну ланку, а та, в свою чергу, розподіляє цей вплив на параметри стану речі. Наприклад, у мобільного телефона параметрами стану є значення напруги на його окремих деталях. Але людина, звісно ж, думає не про те, що хоче подати +3В на такий-то резистор, а про те, що натискає кнопку виклику. Людина натискає кнопку, а внутрішні механізми телефона самі вирішують, як перетворити цей зовнішній вплив на зміну напруг та інших параметрів стану телефона.

Така сама проміжна, посередницька ланка діє і в протилежному напрямку: річ не показує зовнішньому світу безпосередньо всі параметри свого внутрішнього стану, а подає ці параметри на посередницьку ланку. Та, в свою чергу, перетворює значення параметрів до вигляду, який можна показати світові. Наприклад, телефон не повідомляє людині про напругу на кожному своєму резисторі — натомість він перетворює ці напруги у картинки та написи, звуки та вібрації.

В підсумку маємо, що зовнішній світ не має прямого доступу до внутрішніх параметрів стану речі: не може ні сприймати їх значень безпосередньо, ні тим більше змінювати їх на власний розсуд[1]. Більше того, зовнішній світ може взагалі не знати про існування у речі тих чи інших параметрів стану — наприклад, для користування мобільним телефоном непотрібно знати про резистори та напруги, достатньо кнопок та дисплею. Річ має ніби дві сторони: внутрішню, приховану, яку бачить лише вона сама, та зовнішню, відкриту до світу та до інших речей. Це відображено і в об'єктно-орієнтованих

[1] Студентам, які бажають поглиблено вивчати матеріал, треба згадати вчення про річ у собі, сутність та явище за І.Кантом, а також поняття свабхава з індійської філософської традиції локаята.

мовах програмування. Будь-які члени-дані та методи можна зробити прихованими — вони не будуть доступні для сторонніх функцій. В мові Сі++ звертатися до прихованих даних і методів можуть лише методи цього ж класу. Саме для цього призначене слово **private** в рядку 5 другого лістинга з прикладу. Дані і методи, оголошені зі словом **public**, навпаки, загальнодоступні, до них може звертатися будь-яка стороння функція.

Поділ членів об'єкта на загальнодоступні та приховані називають *приховуванням реалізації*, вона становить неодмінну та важливу складову сучасних технологій програмування. Реалізація об'єкта складається з усіх його прихованих членів. Про загальнодоступні члени об'єкта кажуть, що вони разом складають його *інтерфейс*. Згідно норм «чистого» ООП всі члени-дані мають бути приховані, інтерфейс має складатися лише з методів. Однак на практиці іноді буває доцільно дещо відступити від цієї строгої норми, зробивши деяки члени-дані загальнодоступними.

Крім наведених вище міркувань про внутрішню сутність та зовнішні прояви речей, потребу у приховуванні реалізації можна пояснити ще й в інший спосіб. Уявімо собі об'єктно-орієнтовану програму як суспільство, членами якого є об'єкти. Кожен об'єкт має якісь свої приватні справи, якими не бажає ділитися з іншими. Наприклад, сторонній об'єкт не має прямого доступу до гаманця чи особистого щоденника іншого об'єкта. Але об'єкт може звернутися до іншого об'єкта з проханням. Той, в свою чергу, сам вирішує, як виконати це прохання і при цьому, можливо, сам звернеться до свого гаманця чи щоденника. В цій метафоричній моделі гаманець, щоденник та дії безпосередньо з ними це реалізація об'єкта, а ті прохання від інших об'єктів, які він охоче береться виконувати — його інтерфейс.

Приховування реалізації приносить великі переваги для програмування. По-перше, воно спонукає програміста до створення чітких та логічно структурованих програм. Справді, кожен об'єкт при такому підході відповідає за свою конкретну задачу. Оскільки деталі реалізації приховані від інших частин програми, об'єкт змушений вирішувати свою задачу вичерпно та цілковито (внші частини програми просто не матимуть змоги зробити щось за нього). По-друге, розробка програм стає надійнішою: якби, навпаки, деталі реалізації об'єктів були відкриті, помилка програміста в якійсь функції могла

б спотворити члени-дані зовсім стороннього об'єкту (якщо людині дозволити не лише натискати кнопки, але й напряму змінювати напругу на кожному резисторі мобільного телефону, можна було б при спробі набрати номер або спалити апарат, або зтерти всю його пам'ять). А приховування реалізації гарантує, що якщо стан об'єкту виявився недопустимим, то причину слід шукати в його ж методах (а не в усій решті програми) — це сильно звужує коло пошуку джерела помилки.

По-третє, приховування зайвих, допоміжних деталей будови об'єкта допомогає вправлятися з об'єктом. Приховуючи реалізацію, ми нібито позбавляємо зовнішнього користувача якоїсь інформації про об'єкт. Але позбавляємо інформації зайвої, непотрібної, дозволяючи краще зосередити увагу на справді важливій. Іншими словами, при такому підході програма, яка використовує об'єкт, зовсім не залежить від його реализації. Щоб використати об'єкт, програміст має знати лише його інтерфейс і може не навантажувати свою пам'ять відомостями про те, якими засобами цей інтерфейс реалізовано. Уявімо для прикладу мобільний телефон, який показує людині значення напруги на міліонах деталей своєї внутрішньої будови. Користування таким телефоном було б майже неможливим — замість того, щоб показати на дисплеї усього лише ім'я абонента, він показував би безліч подробиць, які лише заважають відшукати потрібне ім'я.

Четверта перевага логічно випливає з третьої. Якщо використання об'єкту в програмі залежить лише від його інтерфейсу і не залежить від реалізації, то програміст може замінити реалізацію будь-якого об'єкта — за умови, що інтерфейс залишається незмінним, зміни всередині того чи іншого об'єкта не потягнуть за собою змін у програмі, яка його використовує (навколишня програма просто не помітить, що під інтерфейсом об'єкту щось змінилося). Іншими словами, усілякі зміни реалізації локалізуються, замикаються в межах об'єкту.

Як приклад, розглянемо мобільний телефон. Нехай фірма-виробник винайшла більш досконалу схему та технологію виготовлення, яка робить телефон надійнішим та дешевшим. При цьому ні функціональність, ні зовнішній вигляд, ні розмір та розташування кнопок нової моделі телефону не відрізняються від старої моделі. Це саме й означає, що в телефону замінено реалізацію та залишено

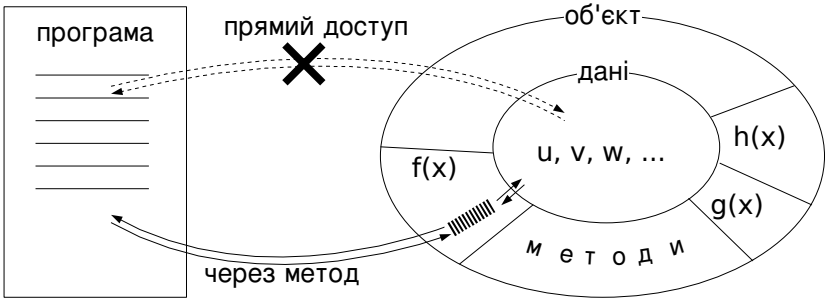

Рис. 1.1. Об'єкт: дані приховані під оболонкою методів

незмінним інтерфейс. Тоді користувач, який звик до старої моделі, просто не помітить заміни нутрощів телефону, зокрема не буде змушений перечитувати інструкцію, забувати старі навички на звикати до нових.

Це, в свою чергу, сприяє принципу *divide et impera* (лат. розділяй та володарюй), що є гаслом технологій програмування протягом кількох останніх десятиліть. Справді, якщо об'єктно-орієнтовану програму розробляють одночасно кілька програмістів, то їм достатньо домовитися між собою лише про інтерфейси своїх об'єктів; реалізацію свого об'єкту кожен може розробляти на власний розсуд, не узгоджуючи її з колегами. Таким чином, значно спрощується спільна робота над великими та складними програмами.

Два вивчені принципи ООП — інкапсуляцію та приховування реалізації — ілюструє рис. 1.1. Дані та методи містяться всередині об'єкта, в одній спільній програмній одиниці. Але дані сховані глибше: зовнішній програмний текст не має права звертатися до них, що показано перекресленою стрілкою. Проте він може звертатися до методів (показано стрілкою), методи, в свою чергу, звертаються до даних та повертають до програми відповідь.

1.4. Класи як відображення понять

Все сказане вище цілком пояснює, навіщо використовувати об'єкти: за допомогою об'єктів у програмі можна відобразити ті чи інші речі,

з яких складається дійсність. Та чи достатньо цього, щоб змоделювати дійсність? Ні, оскільки дійсність, хоча й складається з окремих речей, не вичерпується лише ними. Спостерігаючи дійсність, ми помічаємо в ній не лише окремі речі, але й *різновиди*, сімейства подібних між собою речей. Потреба *класифікувати* речі, тобто знаходити між різними речами спільні риси та поєднувати ці речі у сімейства — неодмінна властивість людського мислення. Її обов'язково має підтримувати гарний стиль програмування, який претендує на зручність та природність для людини.

Об'єднуючи одиничні речі в сімейство, ми тим самим формуємо *загальне поняття*. Кожна одинична річ є представником деякого різновиду, підпадає під те чи інше загальне поняття. Так, «книга» або «вектор» це загальні поняття, а їх представниками є книга «Война і мир» Л.М.Толстого та вектор $\vec{u} = (3, 0, 4)$. Загальне поняття містить в собі визначальні, важливі ознаки та риси, які неодмінно переносяться на кожну одиничну річ, що підпадає під поняття. Наприклад, в загальному понятті «вектор» закладено, що кожен конкретний вектор має певну розмірність, значення компонентів, абсолютну величину (довжину) і т.д. Представник цього поняття, вектор $\vec{u} = (3, 0, 4)$ має розмірність 3, значення компонентів 3, 0 та 4 і довжину 5.

Отже, описавши загальне поняття, ми тим самим задаємо спільну структуру безлічі одиничних речей, які підпадають під це поняття: якими параметрами характеризується їх стан і з яких операцій складаєтья поведінка. Кожна окрема річ при цьому має свої конкретні значення параметрів (скажімо, є вектор $\vec{u} = (3, 0, 4)$ у тривимірному просторі і вектор $\vec{v} = (1, 0)$ у двовимірному), але спільним для всіх речей є сам перелік притаманних їм параметрів і операцій (так, для обох векторів має сенс параметр «розмірність»).

Таким чином, загальне поняття це ніби спільний шаблон для багатьох одиничних речей; кожна одинична річ наповнює цей шаблон власним змістом. Щоб описати ту чи іншу одиничну річ, людині зручно спочатку віднести її до якогось загального поняття, а потім уточнити значення параметрів, наприклад: «нехай $\vec{v}$ це вектор, у якого розмірність 2, а компоненти 1 та -1» або «в мене є книга, явтор якої Л.М.Толстой, а назва "Війна і мир"». Тут перша частина фрази («нехай $\vec{v}$ це *вектор*» та «в мене є *книга*») відносять об'єкт

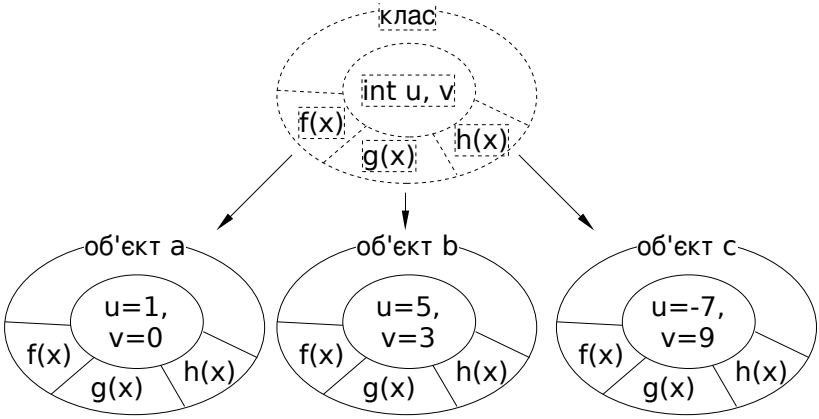

Рис. 1.2. Клас та його екземпляри

до певного різновиду (різновиду векторів та книг), а решта — описує конкретні значення параметрів об'єкту, виокремлює його серед інших об'єктів того ж різновиду.

Прямим втіленням цих філософських засад є наступне правило, що діє майже в усіх в об'єктно-орієнтованих мовах: кожен об'єкт існує не сам по собі, а обов'язково є екземпляром деякого класу. *Клас* — це програмний опис того, які члени-дані та методи будуть наявні в кожному об'єкті цього класу. Це показано на рис. 1.2: клас (показаний пунктиром та розташований зверху) ніби залишає на нижньому рівні кілька проекцій — об'єктів. Кожна проекція має внутрішню будову, запозичену у класу, але наповнює її своїми конкретними значеннями параметрів.

Тут доречно згадати ідеалістичну філософську систему Платона, в основу якої покладено вчення про *ейдоси* — різновиди або ідеї. В дуже спрощеному викладі платонівське вчення стверджує, що в основі світу лежать різновиди, зразки, ідеальні прообрази речей. Конкретні речі, які ми бачимо навколо себе, другорядні та вторинні, вони виникли лише як недосконалий відбиток тих ідеальних прообразів (ніби безліч примарних тіней, відкинутих на наш нижній світ). За Платоном, скажімо, паперовий квадрат є втіленням ідеї

квадратності. Якщо такий квадрат можна порізати, зім'яти (він перестане бути квадратним) чи спалити (він перестане взагалі бути), то ідею квадратності (загальне поняття про квадрат) ніхто не може зіпсувати: на відміну від речей, ідеї вічні, незмінні та досконалі.

Тепер неважко побачити глибоку подібність між вченням Платона та основними засадами об'єктно-орієнтованого програмування. Клас у сенсі ООП відповідає платонівській ідеї (різновиду), а об'єкт, екземпляр класу, відповідає речі, яка є втіленням ідеї. Далі, у ході виконання програми об'єкт може змінювати свій стан, програма може створювати та знищувати об'єкти, тоді як клас жодних змін не зазнає, не виникає та не зникає (справді, клас створює програміст *до того*, як програма почне працювати, і її робота жодним чином не змінює сам клас).

Від філософських засад перейдемо до технічних питань. В мові Сі++ клас являє собою створений програмістом власноруч тип даних. Як відомо з попереднього курсу, тип даних характеризується множиною значень та множиною допустимих операцій. У класа множина операцій — це множина визначених в ньому методів, а множина значень визначається множиною усіх можливих комбінацій значень членів-даних.

Згадаємо, що мова Сі також дозволяє програмісту створювати власні типи даних — структурні типи (**struct**), їх множина значень також складається з усіляких комбінацій значень членів даних. Але ж множина операцій для таких типів жорстко обмежена: це лише звертання до члена структури (взяття поточного значення члена та присвоювання йому нового значення). Натомість в ООП та мові Сі++ можна при створенні свого типу власноруч визначати не лише множину його значень, але й множину операцій.

1.5. Синтаксис та семантика оголошення класу

Оголошення класу в загальному випадку має вигляд:

```
class ім'я_класу {
специфікатор_доступу:
  оголошення членів;
. . .
специфікатор_доступу:
```

```
  оголошення членів;
};
```

Членами класу можуть бути дані (змінні) та методи (функції). Члени-дані оголошуються за загальними правилами мови Сі, як звичайні змінні, в тому числі покажчики і масиви (згодом побачимо, що перед ними можна застосовувати також модифікатори **const**, **static**):

```
тип ім'я; // просто змінна
тип *ім'я; // покажчик
тип ім'я[розмір]; // масив
```

Оголошення методів в простих випадках має такий же вигляд, як і оголошення функцій в мові Сі:

```
тип ім'я( аргументи );
```

Єдина відмінність полягає в тому, що оголошення методу розташовується всередині оголошення класу. В наступних розділах також пояснимо, що при оголошенні методів використовуються такі модифікатори, як **const**, **virtual**, **static**.

В мові Сі++ `специфікатор_доступу` — це одне з трьох службових слів: **public**, **protected** та **private**. Сенс та призначення специфікатору **protected** можна пояснити лише коли вивчено наслідування класів, тому відкладемо його до розділу 2.2, поки що будемо користуватися лише специфікаторами **public** та **private**. Члени класу, оголошені зі специфікатором **public**, є *загальнодоступними*: їх може викликати будь-яка функція (в тому числі метод будь-якого іншого класу). Члени класу, оголошені зі специфікатором **private**, є *прихованими*, до них можуть звертатися лише методи цього ж класу. Члени класу з тими чи іншими специфікаторами можна оголошувати в будь-якому порядку (наприклад, можна спочатку вказати специфікатор **private** і оголосити три члени класу, потім зі специфікатором **public** ще два, а далі знов кілька членів зі специфікатором **private** і під кінець ще кілька членів **public**). Але практика показує, що зручніше за все спочатку оголосити всі **public**-члени, далі, якщо є, **protected**, а потім всі **private**. Отже, оголошення класу набуває такого вигляду:

```
class ім'я_класу {
public:
```

```
  оголошення загальнодоступних членів;
private:
  оголошення прихованих членів;
};
```

Проілюструємо сказане прикладом. Нехай у деякій прикладній прогармі потрібно опрацьовувати прямокутники, у яких сторони паралельні координатним осям. Прямокутники не потрібно малювати на екрані, треба лише зберігати та опрацьовувати дані про них. Кожен прямокутник характеризується координатами центру (x_0, y_0) і довжинами сторін a та b. Щодо будь-якого прямокутника має сенс запитання про його площу, а також запитання, чи лежить точка з заданими координатами (x, y) всередині прямокутника. Крім того, прямокутнику можна призначити певні координати центру та розміри, перемістити на певний вектор $(\Delta x, \Delta y)$ та пропорційно збільшити його розміри в k разів (якщо $k < 1$, вийде зменшення). Звідси прямо слудує, що потрібен клас, який моделює поняття «прямокутник», його членами-даними мають бути координати центру та два розміри, а перераховані вище операції над прямокутниками постають методами класу. Оголошення класу показане в наступному лістингу.

```
class CRect { // прямокутник
// загальнодоступні операції над прямокутником
public:
  // задати координати центра
  void setCenter( double x, double y );
  // задати розміри
  void setSize( double w, double h );
  // змістити
  void move( double dx, double dy );
  // збільшити
  void resize( double k );
  // повертає площу
  double square( void );
  /* повертає:
  1, якщо точка (x,y) лежить всередині,
  0  в іншому випадку */
  int isIn( double x, double y );
private:
  // приховані параметри стану прямокутника
```

```
  double m_x, m_y; // координати центру
  double m_w, m_h; // ширина та висота
};
```

Як було раніше розказано і щойно проілюстровано, оголошення класу містить в собі оголошення (прототипи) методів. Прототип, як відомо з курсу мови Сі, задає ім'я методу, тип значення та типи аргументів. Призначення прототипу — повідомити компілятору, як треба поводитися з функцією (за яким ім'ям її викликати, якого типу аргументи передавати на вхід та якого типу значення чекати з виходу). Окремо від прототипів описують реалізацію функцій. Якщо прототипи методів розміщують всередині оголошення класу, то їх реалізації прийнято виносити за його межі. Реалізація методу оформлюється так:

```
тип ім'я_класу::ім'я_методу( аргументи ) {
  тіло методу
}
```

Тобто якщо при оголошенні метоуд всередині оголошення класу достатньо вказувати лише ім'я власне методу, то в окремо розташованій його реалізації потрібно вказати ім'я класу, спеціальний розділовий знак дві двокрапки (*кваліфікатор області видимості*) та власне ім'я методу; все це разом утворює повне ім'я методу. Справа в тому, що кілька різних класів (наприклад, класи `C1` та `C2` можуть мати методи з однаковим ім'ям, наприклад `f`. Звичайно ж, метод `f` з класу `C1` та метод `f` з класу `C2` — це два різні методи. Саме для того, щоб їх розрізняти, потрібне повне ім'я, відповідно `C1::f` та `C2::f`.

Оголошені в класі члени-змінні стають членами кожного об'єкту даного класу; так само, оголошені в класі методи стають тими операціями, які можна застосовувати до об'єкту класу. Таким чином, метод завжди викликається для конкретного об'єкту і працює над змінними-членами цього об'єкту. Тому в тілі методу можна користуватися оголошеними в класі змінними-членами так само, як і звичайними локальними чи глобальними змінними (брати їх значення, присвоювати їм нові значення, брати х адреси тощо). Коли в тілі методу згадується ім'я змінної-члену, воно означає відповідний член *того об'єкту, для якого викликано цей метод.*

Наведемо реалізації методів класу `CRect`. Перші два методи просто присвоюють значення, передані через аргументи, у відповідні члени-змінні. Два наступні методи змінюють значення членів-змінних: один метод дає приріст координатам центру, а другий збільшує розміри в задану кількість разів. Реалізацію ще двох методів залишаємо читачу в якості вправи

```
void CRect::setCenter( double x, double y ) {
  m_x = x;
  m_y = y;
}
void CRect::setSize( double w, double h ) {
  m_w = w;
  m_h = h;
}
void CRect::move( double dx, double dy ) {
  m_x += dx;
  m_y += dy;
}

void CRect::resize( double k ) {
  m_w *= k;
  m_h *= k;
}
```

1.6. Робота з об'єктами класу

Коли клас оголошено, з ним можна робити все те, що мова Сі дозволяє робити з будь-яким типом даних:

- оголошувати локальні та глобальні змінні такого типу;
- оголошувати змінні типу покажчиків на об'єкти цього типу (а також покажчики вищих рівнів — покажчики на покажчики);
- оголошувати масиви об'єктів даного типу зі заздалегідь визначеною кількістю елементів (тобто з автоматичним видяленням пам'яті);
- динамічно створювати масиви об'єктів даного типу (визначаючи кількість елементів по ходу виконання програми);

- оголошувати і реалізовувати функції, у яких аргументи та/або значення належать цьому типу або типу покажчика на об'єкти цього типу.

Проілюструємо кожен пункт прикладом:

```
// оголошення двох змінних-об'єктів
CRect r1, r2;
// оголошення змінної-покажчика на об'єкт
CRect *p = &r1; // адреса об'єкта r1
// оголошення покажчика на покажчик
CRect **q = &p;
// оголосити масив фіксованого розміру
CRect m[10];
// динамічне створення масиву об'єктів
int n = 20; // кількість елементів
CRect *r = new CRect[n];
/* оголошення функцій зі значеннями або
аргументами типу покажчиків на об'єкти */
CRect *createRects( int n );
void sortBySquare( CRect *s, int n );
```

Зауважимо, що при початковому огляді матеріалу він подається спрощеному вигляді, деякі важливі подробиці відкладаються на потім. Як буде видно з розділу 1.7, створення об'єктів має важливі особливості.

Коли об'єкти класу оголошено, програма може їх обробляти, звертаючись до їх членів. Нехай `об'єкт` — деякий вираз, значенням якого стає певний об'єкт даного класу. Тоді до члену об'єкта можна звернутися через операцію «крапка»: `об'єкт.член`. Ще раз підкреслимо: функція, яка не є методом даного класу (крім винятків, див. розділ 1.11), може звертатися лише до загальнодоступних членів об'єкта. Наведемо кілька прикладів.

```
CRect r;
r.m_x = 12; // помилка !!!
```

Як видно з оголошення класу, цей об'єкт являє собою немов би оболонку, під якою лежать змінні-члени, а на якій розташовано методи. Програмний код не може проникнути крізь оболонку і напряму працювати зі змінними-членами об'єкта, оскільки вони приховані (мають специфікатор **private**), тому у другому рядку компілятор

повідомить про помилку. Натомість з об'єктом можна працювати, звертаючись до загальнодоступних методів:

```
CRect r;
r.setCenter( 12, 8 );
r.setSize( 4, 2 );
r.move( 1, -3 );
r.resize( 1.5 );
double s = r.square();
double x, y;
cin >> x >> y;
if( r.isIn( x, y ) )
  cout << "Влучно" << endl;
else
  cout << "Мимо" << endl;
```

Виклик методу `setCenter` у другому рядку бере значення 12 та 8 і присвоює їх у члени `m_x` та `m_y` — тобто встановлює кооридинати центра. Так само у наступному рядку метод `setSize` присвоює пеіні значення розмірів прямокутника. Наступні два рядки пересувають прямокутник та збільшують його розміри. Далі обчислюється його площа і отримане значення присвоюється в змінну. Нарешті користувач вводить координати точки, а програма запитує у прямокутника, чи потрапляє точка всередину його.

У щойно розглянутому прикладі об'єкт, до якого зверталася програма, був всюди заданий найпростішим з усіх способів — змінною `r`. В принципі нічого не змінюється, якщо треба працювати з об'єктом – елементом масиву:

```
#define N 10
CRect m[N];
for( int i = 0; i < N; ++i ) {
  m[i].setCenter( 0, 0 );
  m[i].setSize( 2*i + 4, i + 2 );
}
```

Тут об'єкт, для якого викликається метод, заданий виразом `m[i]`. Тобто щоразу метод викликається для поточного елементу масиву об'єктів.

Нехай тепер `покажчик` — деякий вираз, значенням якого є покажчик на певний об'єкт класу. Звертатися до члену об'єкта через по-

кажчик треба за допомогою операції «стрілка»: `покажчик->член`, наприклад:

```
void resetRect( CRect *p ) {
  p->setCenter( 0, 0 );
  p->setSize( 1, 1 );
}
```

Тут функція отримує аргумент типу покажчика на об'єкт класу. Семантика обох рядків тіла функції така: взяти об'єкт, на який вказує покажчик `p` та викликати для цього об'єкту відповідний метод. Щоб завершити приклад, покажемо і можливий виклик щойно описаної функції:

```
CRect r;
resetRect( &r );
```

1.7. Конструктори

Згадаємо ще раз наведений вище клас «прямокутник» (`CRect`) і розглянемо оголошення змінної-об'єкту:

```
CRect r;
```

Коли виконання програми доходить до цього оголошення, створюється новий об'єкт під іменем `r`, всередині якого містяться змінні-члени `m_x`, `m_y`, `m_w`, `m_h`. На момент створення їм ще не присвоєно якогось осмисленого значення; випадково в тих комірках пам'яті, які система виділить під зберігання об'єкту, можуть виявитись будь-які значення, в тому числі й фізично безглузді (скажімо, від'ємне значення ширини). Таким чином, одразу після створення об'єкт у змінній `r` може набути непередбачуваного, безглуздого та навідь недопустимого стану.

Між тим, *надійність* є однією з ключових вимог ООП. Зокрема, вимагається, що об'єкт повинен завжди перебувати в коректному та передбачуваному стані (саме для того й приховуються параметри стану від зовнішнього програмного коду, щоб програма не змогла через помилку зіпсувати стан об'єкту). Щоб втілення цієї вимоги було справді послідовним, потрібен спеціальний механізм, який би

гарантував присвоювання членам-даним об'єкта коректних початкових значень.

В більшості об'єктно-орієнтованих мов це робиться за допомогою *конструкторів* — спеціальних методів, які викликаються автоматично в момент створення об'єктів. В мові Сі++ конструктори мають такі властивості і правила написання:

- Якщо в класі є конструктор, то будь-який об'єкт цього класу створюється з викликом конструктора. Якщо конструктора в класі немає, об'єкт створюється і сам, але залишається неініціалізованим. Конструктору можна доручити присвоювання членам об'єкта певних початкових значень чи інші особливі дії.
- Якщо звичайний метод викликається явним чином, і його можна викликати скільки завгодно разів для певного об'єкту, то конструктор явним чином викликати неможливо — для кожного об'єкту він викликається автоматично і тільки один раз: в момент створення об'єкту.
- Конструктор позначається як метод, ім'я якого збігається з іменем класу.
- На відміну від звичайних методів, при конструкторі не пишеться тип значення, що повертається.
- Клас може мати скільки завгодно конструкторів. Маючи однакове ім'я, вони відрізняються кількістю або типами аргументів (згадати про перевантаження імен функцій).
- Забігаючи наперед, скажемо, що конструктор неможливо оголосити віртуальним (див. розділ 2.6).
- Знов забігаючи наперед, скажемо, що конструктор неможливо оголосити статичним (див. розділ 1.14).

Початкову настройку чогось у програмістів прийнято називати *ініціалізацією*. В тому числі і про конструктор кажуть, що він призначений для ініціалізації об'єктів класу.

Проілюструємо сказане. В оголошення класу `CRect` додамо конструктор з чотирьма аргументами, через які передаються початкові значення координат та розмірів (для економії місця наведемо лише оголошення конструктору — решта членів класу залишаються такі ж, як і в попередніх прикладах):

```
class CRect {
public:
  CRect( double x_, double y_,
    double w_, double h_ );
  // далі без змін ...
};
```

Конструктор оголошено першим серед усіх членів класу. Що це саме конструктор, видно з того, що ім'я методу `CRect` — тобто таке ж, як і у класа. Перед іменем методу не вказано тип значення (порівняти, наприклад, з методами `move` чи `square`).

Що стосується реалізації, то її пояснимо в два кроки. Спершу наведемо не зовсім красивий, але простий для розуміння варіант:

```
CRect::CRect(
  double x_, double y_,
  double w_, double h_
) {
  m_x = x_;
  m_y = y_;
  m_w = w_;
  m_h = h_;
}
```

Чому слово `CRect` повторено двічі, має бути зрозуміло: при реалізації будь-якого методу треба вказати його повне ім'я, що складається з імені класу, двох двокрапок та імені методу; в даному ж випадку, оскільки метод є конструктором, ім'я методу та ім'я класу співпадають. Тіло цього конструктора цілком очевидне: значення, які передано на вхід конструктора в якості аргументів, присвоюються відповідним членам об'єкта.

Тепер дамо більш грамотну форму цього ж конструктора. Надання початкових значень членам об'єкта прийнято оформлювати в особливу конструкцію — *список ініціалізаторів*. Список ініціалізаторів пишеться перед тілом конструктора (тобто перед його відкривальною фігурною дужкою). В тілі конструктора залишають лише ті оператори, які не зводяться до початкових присвоювань (якщо вони є: наприлад, за сенсом задачі конструктор, створюючи об'єкт, може читати дані з файлу, друкувати повідомлення тощо), якщо ж таких операторів немає, тіло конструктора виходить порожнім —

між фігурними дужками немає жодного оператора. Кожен ініціалізатор в списку має вигляд: ім'я змінної-члену і в дужках вираз, значення якого цій змінній присвоюється. Ініціалізатори в списку відокремлюються комами. Між круглою дужкою, що закриває список аргументів конструктора, та списком ініціалізаторів ставиться двокрапка. Наприклад:

```
CRect::CRect(
  double x_, double y_,
  double w_, double h_
): m_x(x_), m_y(y_), m_w(w_), m_h(h_)
{}
```

Цей запис означає, що значення аргументу `x_` присвоюється члену `m_x`, значення аргументу `y_` — члену `m_y` і т. д. Тіло конструктору порожнє, оскільки для створення об'єкта, що моделює прямокутних, інших операцій, окрім ініціалізації змінних-членів, не потрібно.

Аргументи для конструкторів треба вказувати при оголошенні змінної — об'єкта класу:

```
CRect r1( 0, 0, 2, 1 );
double x2, y2, w2, h2;
cin >> x2 >> y2 >> w2 >> h2;
CRect r2( x2, y2, w2, h2 );
```

Тут оголошуються два об'єкти класу, одразу ж при створенні для кожного з них викликається конструктор, причому першому об'єкту передаються початкові значення координат центру $(0, 0)$ та розмір 2×1, а початкові значення для другого об'єкту вводяться користувачем.

Те ж саме стосується й динамічного створення об'єктів. В наступних рядках об'єкт класу створюється динамічно, і в момент створення йому передаються початкові значення:

```
CRect *p;
double x0, y0, w0, h0;
cin >> x0 >> y0 >> w0 >> h0;
p = new CRect( x0, y0, w0, h0 );
```

Як видно з прикладу, при динамічному створенні одного об'єкта за допомогою операції **new**, початкові значення для передачі в конструктор треба записати після імені типу.

Якщо клас взагалі має хоча б один конструктор, то створення об'єкту обов'язково супроводжується викликом конструктора (для порівняння: в попередніх розділах наводилися приклади класів без конструкторів, і саме тому при створенні об'єктів цих класів не відбувалася ініціалізація). Навіть якщо при оголошенні об'єкта навимсно чи по необережності не передати початкові значення для ініціалізації, транстятор подбає про те, щоб заборонити таку спробу. Наприклад, якщо написати

```
CRect r;
CRect *p;
p = new CRect;
```

то компілятор побачить, що в першому і в третьому рядку робиться спроба створити об'єкт, не передаючи аргументи до конструктора. Оскільки конструктор класу `CRect` вимагає передачі чотири аргументи, в обох випадках компілятор зафікскє помилку. Тим самим мова Сі++ гарантує, що жоден об'єкт в програмі не може з'явитися без тих чи інших початкових значень.

У щойно розібраного правила є один очевидний виняток: коли у класі є конструктор, що не вимагає аргументів, так званий *конструктор по замовчуванню*. Він дозволяє створювати об'єкти, не передаючи їм параметрів — цей конструктор створює об'єкти з якимось заздалегідь визначеним станом. Наприклад, для прямокутників можна запровадити конструктор по замовчуванню, який надає прямокутнику початкові координати центру $(0, 0)$ та розміри 1×1. Тепер у класі буде два конструктори: конструктор по замовчуванню дозволяє створити одиничний квадрат з центром у початку координат, а конструктор з чотирьма аргументами — довільний прямокутник. Для економії місця не наводимо повний текст оновленого оголошення класу «прямокутник», бо крім появи нового конструктора в ньому нічого не змінюється

```
class CRect {
public:
  CRect( void ); // по замовчуванню
  CRect( double x_, double y_, // з аргументами
    double w_, double h_ );
  // далі без змін ...
};
```

```
CRect::CRect( void ) :
// початкові значення відомі
m_x(0), m_y(0), m_w(1), m_h(1)
{} // тіло порожнє
```

Тепер, коли такий конструктор в класі є, наступні оголошення та оператори стають цілком правильними:

```
CRect r;
CRect *p;
p = new CRect;
```

В першому і третьому рядку при створенні об'єкту викликається конструктор по замовчуванню, який надасть і об'єкту `r`, і динамічно створеному об'єкту правильного початкового стану.

Тут доречно зробити таку ремарку, яка виходить за межі теми власне конструкторів. В мові Сі будь-яка дія, що виконується програмою, явним чином записана в її тексті. Натомість в мові Сі++, як видно з попереднього крикладу, є такі випадки, коли важливі операції виконуються в тій точці програмного тексту, де начебто про виклик цих операцій нічого не написано. Скажімо, в рядку, де оголошено змінну `r`, явним чином не написано викликати конструктор, але він все одно викликається, бо це визначено правилами мови Сі++. Таким чином, щоб зрозуміти хід виконання програми мовою Сі++, треба читати програмний текст між рядків, тобто добре розуміти не лише те, що в ньому явно написано, але й те, що неявно мається на увазі згідно правил мови. В подальшому викладі подібні тонкощі траплятимуться неодноразово.

Саме завдяки конструкторам по замовчуванню стає можливим створювати масиви об'єктів. Розглянемо оголошення масиву з автоматичним виділенням пам'яті та операцію динамічного створеня масиву

```
CRect m[10];
CRect *q;
q = new CRect[20];
```

Кожен елемент масивів `m` та `q` є об'єктом класу, а це значить, що його існування, за загальним правилом, має починатися з виклику конструктора. В першому та третьому рядках не лише виділяється

(відповідно, автоматично та динамічно) пам'ять для зберігання масивів об'єктів, але й для кожного елементу цих масивів викликається конструктор. Зрозуміло, що це може бути лише конструктор по замовчуванню (для іншого конструктора потрібні були б аргументи).

Ще один важливий різновид конструкторів розберемо пізніше, в розділі 1.9.

1.8. Деструктори

Як щойно пояснювалося, конструктор це спеціальний мтод, який керує створенням об'єктів даного класу. Цілком симетрично, *деструктор* — це спеціальний метод, який керує знищенням об'єктів. Деструктор викликається автоматично (тобто викликати його власними руками в явному вигляді непотрібно та й немає сенсу), і викликається щоразу, коли деякий об'єкт знищується. Таким чином, життя кожного об'єкту починається викликом конструктору і закінчується викликом деструктору. Наведемо основні правила, якими керується написання і використання деструкторів.

- Якщо в класі є деструктор, то будь-який об'єкт цього класу знищується з його викликом. Якщо деструктора в класі немає, то система сама видаляє об'єкт з пам'яті; деструктор потрібен для того, щоб при знищенні об'єкту виконати якісь визначені програмістом особливі дії.
- Якщо звичайний метод викликається явним чином, і його можна викликати скільки завгодно разів для певного об'єкту, то деструктор явним чином викликати не треба — для кожного об'єкту він викликається автоматично і тільки один раз: в момент знищення об'єкту. А саме: якщо об'єкт створено динамічно (за допомогою операції **new**), то знищується він операцією **delete**; якщо об'єкт є значенням локальної змінної, то знищується він тоді, коли виконання програми доходить до кінця блоку програмного коду, де ця змінна оголошена (див. приклади нижче).
- Деструктор позначається як метод, ім'я якого збігається з іменем класу, перед яким стоїть знак `~`, наприклад `~CVector`.

- На відміну від звичайних методів, при деструкторі не пишеться тип значення, що повертається.
- Клас може мати лише один деструктор. Деструктор не має аргументів.
- Забігаючи наперед, скажемо, що деструктор (на відміну від конструктора) може бути віртуальним (див. розділ 2.6).
- Знов забігаючи наперед, скажемо, що деструктор (як і конструктор) неможливо оголосити статичним (див. розділ 1.14).

Покажемо роботу деструктора на найпростішому прикладі. Спочатку означимо клас, в якому є деструктор, а потім розберемо кілька варіантів програми, що його використовує. Наведений нижче клас не вирішує якуть практичну задачу, а призначений лише для демонстраційних цілей.

```
class CDemo {
public:
  CDemo( int ); // з аргументом
  CDemo( void );   // по замовчуванню
  ~CDemo( void );  // деструктор
private:
  int m_x;
};

CDemo::CDemo( int x_ ) : m_x(x_) {
  cout << "Створено з початк.знач. " << x_ << endl;
}

CDemo::CDemo( void ) : m_x(-1) {
  cout << "Створено по замовчуванню" << endl;
}

CDemo::~CDemo( void ) {
  cout << "Знищується, x=" << m_x << endl;
}
```

Для початку нехай програма складається лише з функції `main`, і в ній оголошуються локальні змінні типу об'єкта класу:

```
1 int main( void ) {
```

```
2   cout << "Початок␣програми" << endl;
3   CDemo a, b(7);
4   cout << "Завершення␣програми" << endl;
5   return 0;
6 }
```

Коли виконання програми доходить до рядку 3, створюється спочатку об'єкт `a` (з викликом конструктора по замовчуванню), а потім об'єкт `b` (з викликом конструктора з аргументом і передачею до нього значення 7). Одразу після виконання рядка 4 функція `main` завершує свою роботу. Оскільки `a` та `b` є локальними змінними цієї функції, вони знищуються, а оскільки значеннями цих змінних є об'єкти класу, для цих об'єктів викликаються деструктори. За загальним правилом, процес знищення об'єктів є номов би дзеркальним відображенням процесу їх створення, тому спочатку знищується об'єкт `b`, а потім об'єкт `a`. Тому програма видасть такий текст:

```
Початок програми
Створено по замовчуванню
Створено з початк.знач. 7
Завершення програми
Знищується, x=7
Знищується, x=-1
```

Розглянемо тепер створення і знищення масивів об'єктів з автоматичним розподілом пам'яті:

```
1 int main( void ) {
2   cout << "Початок␣програми" << endl;
3   CDemo m[3];
4   cout << "Завершення␣програми" << endl;
5   return 0;
6 }
```

В рядку 3 для кожного об'єкта–елемента щойно створеного масиву викликається конструктор по замовчуванню. Відповідно, у рядку 5 для кожного елементу маиву `m` викликається деструктор. Програма в процесі виконання дасть такий результат:

```
Початок програми
Створено по замовчуванню
Створено по замовчуванню
Створено по замовчуванню
```

```
Завершення програми
Знищується, x=-1
Знищується, x=-1
Знищується, x=-1
```

Розглянемо трохи складніший випадок, коли у програмі є кілька функцій, що мають локальні змінні типу об'єктів класу:

```
1  void f( void ) {
2    cout << "Функція f почалася" << endl;
3    CDemo b;
4    cout << "Функція f закінчилася" << endl;
5  }
6  int main( void ) {
7    cout << "Початок програми" << endl;
8    CDemo a( 1 );
9    f();
10   cout << "Завершення програми" << endl;
11   return 0;
12 }
```

Виконання програми починається з функції `main`, і першим створюється об'єкт `a` у рядку 8. Потім викликається функція `f`, у якій створюється локальний об'єкт `b` (рядок 3). Цей об'єкт одразу ж знищується при поверненні з функції `f`. Нарешті завершується і функція `main` і при цьому знищується і об'єкт `a`. Виконання програми дає такий результат:

```
Початок програми
Створено з початк.знач. 1
Функція f почалася
Створено по замовчуванню
Функція f закінчилася
Знищується, x=-1
Завершення програми
Знищується, x=1
```

Насамкінець розберемо програму з динамічним створенням та знищенням об'єктів.

```
1  CDemo *f( void ) {
2    cout << "Функція f почалася" << endl;
3    CDemo *q = new CDemo(1);
```

```
4   cout << "Функція␣f␣закінчилася" << endl;
5   return q;
6 }
7 int main( void ) {
8   cout << "Початок␣програми" << endl;
9   CDemo *p = f();
10  cout << "Функція␣main␣знищує" << endl;
11  delete p;
12  cout << "Завершення␣програми" << endl;
13  return 0;
14 }
```

Функція `main` сама не створює об'єкт. Вона викликає функцію `f`, яка створює об'єкт динамічно і повертає покажчик на нього. Об'єкт в динамічній пам'яті продовжує жити і після того, як завершилася функція, що його породила. Тому після повернення у функцію `main` її локальна змінна `p` міститиме покажчик на об'єкт. Динамічно створені об'єкти самі не знищуються — програма повинна сама в певний момент часу застосувати до них операцію видалення **`delete`**. В рядку 11 для об'єкту викликається деструктор і одразу після того, як він відпрацює, об'єкт видаляється з пам'яті. Отже маємо такий результат виконання програми:

```
Початок програми
Функція f почалася
Створено з початк.знач. 1
Функція f закінчилася
Функція main знищує
Знищується, x=1
Завершення програми
```

На с.46 ми вже зазначали, що у мові Сі++ часто бувають випадки, коли важливі операції викликаються там, де в тексті програми їх виклик начебто ніяк не позначено. Деструктори це ще один приклад такого явища: в щойно наведених прикладах програм у відповідних рядках не написано «викликати деструктор для того чи іншого об'єкту». Щоб побачити там виклик деструктора, треба добре знати правила мови і уявляти логіку роботи програми.

Наведені вище приклади ілюструють лише те, як і коли деструктор викликається. Зараз з'ясуємо, *навіщо* включати їх в клас. При програмуванні справді складних, реальних задач часто буває так,

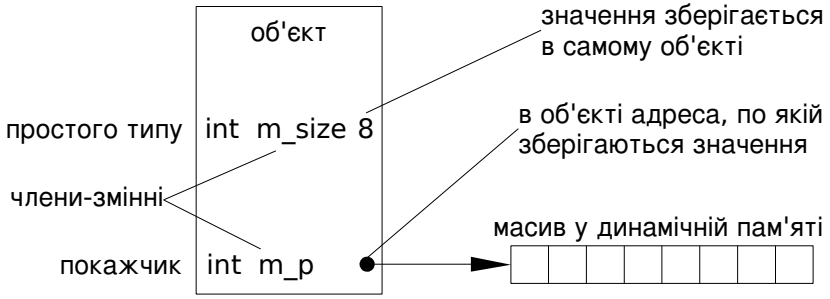

Рис. 1.3. Об'єкт з членом-покажчиком на динамічні дані

що об'єкт має у серед своїх членів-даних не лише значення простих типів (цілих, дійсних тощо), але й покажчики на динамічно виділені області пам'яті (найчастіше масиви). Важливо зрозуміти: покажчик на дані є членом об'єкту (тобто міститься в об'єкті), а самі ці дані знаходяться у динамічній пам'яті, за межами об'єкту, рис. 1.3. Для прикладу можна розглянути клас «вектор цілих чисел», у якого є змінна-член — покажчик на динамічно виділений масив елементів вектору; інший приклад — клас, у якого є член–покажчик на динамічно виділений текстовий рядок. Уявімо, що відбудеться при знищенні такого об'єкту, якщо у класі не описано деструктор. Об'єкт зникає з пам'яті разом з усіма своїми змінними-членами, відтепер їх значення вже не дістати. Але ж масив чи рядок так і залишається в динамічній пам'яті: з точки зору системи ця ділянка динамічної пам'яті все ще відмічена як зайнята, бо системі ніхто не наказував її звільнити.

Тепер до цього масиву неможливо «дотягнутися» жодним чином, бо єдиний «канал доступу до нього» — покажчик — загинув разом з об'єктом. Програма не може не лише користуватися масивом (наприклад, вести обчислення з його елементами), але й остаточно знищити його, бо для цього також потрібно знати покажчик. Так пам'ять засмічується «мертвими» об'єктами, які лише займають місце. Саме для того, щоб цього не відбувалося, і використовуються деструктори. Зауважимо, що в виділяє пам'ять під такі динамічні дані, як правило, конструктор.

Розглянемо приклад — клас «вектор цілих чисел». Змінна-член m_p вказує на масив, пам'ять під який виділяється динамічно. У конструкторі цей динамічний об'єкт створюється і заповнюється початковими значеннями (для цього конструктору передається аргумент — довжина масиву, який треба створити), а у деструкторі знищується. Для наочності конструктор та деструктор друкують діагностичні повідомлення, які допоможуть зрозуміти логіку виконання програми.

```
class CVector {
public:
  CVector( int size_ );
  ~CVector( void );
  // ...інші методи...
private:
  int m_size;
  int *m_p;
};

CVector::CVector( int size_ ) :
  m_size( size_ ),
  m_p( new int[size_] )
{
  for( int i = 0; i < m_size; ++i )
    m_p[i] = 0;
  cout << "Дин.масив␣довжиною␣"
       << m_n << "␣створено" << endl;
}

CVector::~CVector() {
  delete[] m_p;
  cout << "Дин.масив␣довжиною␣"
       << m_n << "␣знищено" << endl;
}

int f( void ) {
  CVector a(8);
  // ..якісь оператори..
  return якесь_значення;
}
```

Коли у функції `f` створюється локальний об'єкт `a`, для нього викликається конструктор, який виділяє динамічну пам'ять для нового масиву замовленої довжини, а покажчик на його початок присвоює члену `m_p` цього об'єкта. Коли функція `f` завершується (передостанній рядок), а її локальні змінні знищуються, для об'єкта `a` буде автоматично викликано деструктор, який звільнить динамічну пам'ять від масиву.

В більш складних програмах об'єкт може володіти не лише покажчиками на області динамічної пам'яті, але й іншими системними ресурсами (скажімо, об'єкт може містити дескриптор відкритого файлу, дескриптор вікна у графічній оболонці тощо). В таких випадках також слід використовувати деструктор, щоб об'єкт перед зникненням встиг звільнити ці ресурси (закрити файл, знищити графічне вікно).

Наостанок ще раз підкреслимо (студенти часто припускаються помилки саме в цьому): деструктор сам по собі не знищує об'єкт; деструктор — це метод, який система автоматично викликає перед тим, як знищити об'єкт.

1.9. Тимчасові копії об'єктів та конструктор копіювання

В цьому розділі розберемо особливий різновид конструктору, призначенням якого є створити новий об'єкт як копію (клон) іншого об'єкта даного класу. Цей різновид дуже важливий на практиці, оскільки під час роботи справді складних програм мало не щомиті автоматично створюються тимчасові копії об'єктів: зокрема, коли об'єкт передається до функції в якості аргументу чи повертається з функції як її значення.

В розділі 1.7 йшлося про конструктори з аргументами, потрібні для того, щоб створювати нові об'єкти даного класу, передаючи через аргументи початкові значення для їх ініціалізації. Конструктор копіювання — це такий конструктор, аргументом якого є об'єкт (більш точно — *посилання* на об'єкт), з якого треба зняти копію. Іншими словами, об'єкт-оригінал і є тими початковими даними, які використовуються для створення нового об'єкту.

Якщо у класі не означити своїми руками конструктор по замов-

чуванню, система й сама може знімати з об'єктів копії побайтно — просто присвоюючи членам-змінним нового об'єкту значення відповідних змінних-членів об'єкта-оригінала. Але іноді (особливо коли клас містить члени – покажчики на динамічно виділені області пам'яті) такого способа клонування виявляється недостатньо: щоб копія була правильною, треба власними руками визначити алгоритм створення об'єкта-копії. Він і оформлюється у вигляді конструктора копіювання. Подальший розгляд побудуємо по такій схемі:

- спочатку розберемо в подробицях, як і коли створюються тимчасові копії при передачі об'єкта до функції в якості аргументу;
- далі на прикладі примітивного класу (для якого, зрештою, вистачило б і побайтного копіювання) розберемо, як писати конструктор копій;
- за допомогою цього конструктора зможемо дослідити в подробицях механізм створення та знищення тимчасових копій;
- потім на прикладі покажемо, що відбувається, коли у класі з членами-покажчиками залишити побайтний механізм створення копій, і чому для таких класів абсолютно необхідний написаний власноруч конструктор копіювання;
- насамкінець покажемо, як треба реалізовувати конструктор копіювання для такого класу.

Копіювання об'єкта при передачі через аргумент Нехай `C` — деякий клас, `f` — функція з аргументом типу `C`, `a` — деяка змінна, в якій міститься об'єкт класу `C`; нехай функція `f` викликається і об'єкт `a` передається їй в якості значення для аргумента `x`:

```
void f( C x ) {
 // якісь оператори
}
int main( void ) {
  C a;
  //...
  f(a);
  //...
}
```

Розглянемо в подробицях, як здійснюється виклик функції `f` (рядок 7) і що при цьому відбувається з об'єктом-аргументом.

За загальним правилом, відомим ще з курсу програмування мовою Сі, значенням аргументу `x` функції `f` стає не сам об'єкт `a`, а його копія. Таким чином, в момент виклику функції `f` створюється новий об'єкт `x`, і в нього копіюється об'єкт `a` функції `main`. При цьому, якщо в класі `C` є конструктор копіювання, то для копіювання об'єкту `a` в об'єкт `x` використовується саме він, а якщо такого конструктора в класі немає, то система сама копіює вміст об'єкта байт за байтом.

Згадаємо також, що аргумент (в даному випадку `x`) є локальною змінною своєї функції (в даному прикладі функції `f`). За загальним правилом, також відомим з курсу програмування мовою Сі, локальні змінні знищуються при виході з функції. Це означає, що якщо у класі `C` означено деструктор, він буде викликаний для об'єкту `x`, коли виконання дійде до рядка 3. Якщо ж деструктора в класі немає, система просто видалить об'єкт з пам'яті.

Перепишемо наведений вище клас `CDemo`, щоб за допомогою його наочно дослідити описаний вище процес створення та знищення тимчасової копії.

```
1  class CDemo {
2  public:
3    CDemo( int );    // звичайний конструктор
4    CDemo( const CDemo& );    // конструктор копіювання
5    ~CDemo( void );  // деструктор
6  private:
7    int m_x;
8  };
9
10 CDemo::CDemo( int x_ ) : m_x( x_ ) {
11   cout << "Об'єкт створено, x=" << m_x << endl;
12 }
13
14 CDemo::CDemo( const CDemo &obj ) : m_x( obj.m_x ) {
15   cout << "Об'єкт скопійовано, x=" << m_x << endl;
16 }
17
18 CDemo::~CDemo( void ) {
19   cout << "Об'єкт знищується, x=" << m_x << endl;
```

```
20 }
```

Конструктор, оголошений у рядку 4, є конструктором копіювання — це видно з того, що конструктор має один аргумент — посилання на об'єкт свого ж класу. Цей конструктор бере значення члену-змінної `m_x` об'єкта-оригінала та присвоює його члену-змінній свого об'єкту, а також друкує повідомлення. Розглянемо тепер програму, де об'єкт цього класу передається до функції в якості аргументу.

```
1  void f( CDemo x ) {
2    cout << "Функцію␣f␣викликано" << endl;
3  }
4
5  int main( void ) {
6    CDemo a( 7 );
7    cout << "Виклик␣функції␣f" << endl;
8    f( a );
9    cout << "Повернення␣до␣main" << endl;
10   return 0;
11 }
```

Результатом її виконання стає такий текст:

```
Об'єкт створено, x=7
Виклик функції f
Об'єкт скопійовано, x=7
Функцію f викликано
Об'єкт знищується, x=7
Повернення до main
Об'єкт знищується, x=7
```

Як видно з цього протоколу, спочатку при виклику функції `f` перш за все об'єкт копіюється в її аргумент (для чого викликається конструктор копіювання), потім виконується тіло функції, далі аргумент функції знищується, і виконання програми повертається у функцію `main`.

Розглянемо тепер клас `CVector`, наведений на с. 53, і з'ясуємо, що відбувається у процесі роботи наступної програми (подібної до щойно розібраної):

```
1  void f( CVector x ) {
2    cout << "Функцію␣f␣викликано" << endl;
```

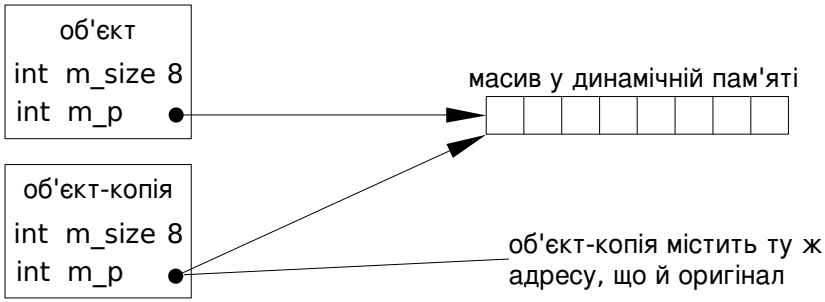

Рис. 1.4. Побайтне копіювання об'єктів з членами-покажчиками: виникнення паразитного покажчика

```
}

int main( void ) {
  CVector a( 7 );
  cout << "Виклик функції f" << endl;
  f( a );
  cout << "Повернення до main" << endl;
  return 0;
}
```

Як вже пояснювалося вище, в процесі виклику функції `f` створюється тимчасова копія об'єкта `a`. Але, на відміну від розібраного вище класу `CDemo`, у класі `CVector` немає конструктора копіювання. Тому в змінній `x` створюється побайтна копія об'єкта `a`. А це означає, що об'єкт `x` міститиме в члені `m_p` ту ж саму адресу масиву, що й об'єкт `a`. Іншими словами, виникає ситуація, коли два різних об'єкти через свої члени-почажчики спільно володіють однією областю пам'яті (рис. 1.4). Це явище (крім нечастих випадків, коли програміст спричиняє його навмисно і добре знає, що робить) дуже небезпечне, бо спричиняє помилки, які спотворюють роботу програми і які буває важко знайти.

Справді, уявімо, що у векторі `x` програма заповнює всі елементи нулями. Але ж, оскільки об'єкти `x` та `a` ділять між собою один і той самий масив, обнулиться і вектор `a` у функції `main`. Ще гір-

ша неприємність трапляється, коли функція `f` завершується і передає управління назад до функції `main`. За загальним правилом для локального об'єкта `x` викликається деструктор. Деструктор знищує масив, покажчик на який зберігається в члені `m_x` об'єкта `x`. Але ж це той самий масив, яким володіє ще й об'єкт `a`. Виходить, що без відома об'єкта `a` хтось знищує його дані! Після повернення з функції `f` будь-які спроби працювати з об'єктом `a` спричинять помилку, бо призведуть до операцій над вже знищеним масивом.

Все це грубо порушує принцип локальності, згідно якого все, що функція `f` робить зі своїми локальними змінними, не повинно жодним чином впливати на локальні змінні інших функцій. Зрештою, значення аргументу при передачі до функції для того і копіюється у тимчасову змінну, щоб унеможливити спотворення об'єкта-оринінала.

Як видно з цього прикладу, побайтне копіювання об'єктів годиться тільки тоді, коли об'єкти містять члени-змінні лише скалярних типів (цілі, дійсні тощо), та зовсім не годиться, коли об'єкти містять члени-покажчики на дані, створені у динамічній пам'яті. В останньому випадку, щоб позбутися зазначених вище помилок, треба самостійно реалізувати конструктор копіювання.

Як видно з попереднього параграфу, при копіюванні об'єкта класу `CVector` треба не переносити в об'єкт-копію адресу того ж масиву, а створювати новий масив, в який копіювати елементи масиву з об'єкта-оригінала. Іншими словами, результатом копіювання об'єкта має стати ситуація, показана на рис. 1.5, замість розібраної вище (рис. 1.4).

Цю задачу вирішує конструктор копіювання, наведений в наступному лістингу. В лістингу показано оголошення класу, до якого додалося оголошення нового конструктора, а також показана реалізація цього конструктора. Реазізації другого конструктора та деструктора залишаються такими ж, як і в лістингу на с. 53.

```
1 class CVector {
2 public:
3   CVector( int n_ );
4   CVector( const CVector& );
5   ~CVector( void );
6   // ...інші методи...
```

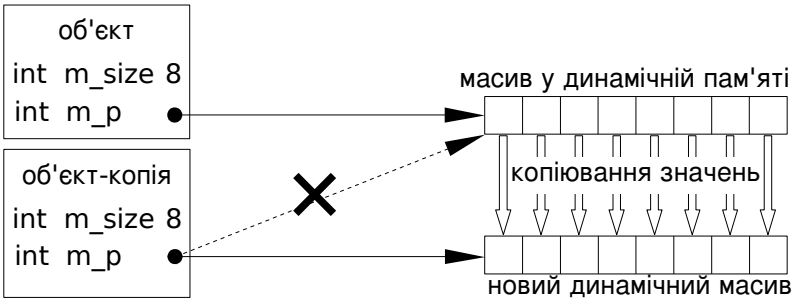

Рис. 1.5. Правильне копіювання об'єктів з членами-покажчиками

```
7  private:
8    int m_n;
9    int *m_p;
10 };
11
12 CVector::CVector( const CVector &obj ) :
13   m_n( obj.m_n ),
14   m_p( new int[obj.m_n] )
15 {
16   for( int i = 0; i < m_n; ++i )
17     m_p[i] = obj.m_p[i];
18   cout << "Вектор␣скопійовано" << endl;
19 }
```

З цим класом програма, наведена на с. 57, працює коректно.

1.10. Підоб'єкти

В розглянутих вище прикладах члени об'єктів належали або простим стандартним типам, або були покажчиками на масиви елементів таких типів. Та ніщо не заважає членам-даним об'єкта в свою чергу бути об'єктами інших класів. Таким чином, об'єкти можуть бути вкладені один в одний. Об'єкт, що є членом іншого об'єкта, тобто входить до складу останнього, називають *підоб'єктом*.

Для прикладу уявімо клас, що моделює поняття «книжкова сторінка» (такий клас може знадобитися, скажімо, в програмному про-

дукті, що готує текст до друку). Сторінка має власні розміри, полями певної ширини обмежується прямокутна область тексту всередині. Таким чином, членами об'єкту «сторінка» мають бути: ширина та довжина в міліметрах і поля (дійсні числа) та область тексту (підоб'єкт класу «прямокутник»). Для простоти вважаємо, що поля у сторінки з усіх боків однакові. Нижче показано оголошення класу, що моделює сторінку:

```
class CPage {
public:
  // аргументи конструктра:
  // ширина, висота сторінки, ширина поля
  CPage( double, double, double );
  // інші методи ...
private:
  double m_width;
  double m_height;
  double m_margin;
  CRect m_text; // підоб'єкт
};
```

Єдина важлива особливість об'єктів, що мають підоб'єкти, полягає в способі їх створення. Коли створюється об'єкт (в даному прикладі — об'єкт класу `CPage`, одночасно мають створитися і всі його підоб'єкти (в даному випадку член `m_text` класу `CRect`). Але ж створення об'єктів завжди має супроводжуватися викликом конструктора, який надасть їм правильних початкових значень. В мові Сі++ діє загальне правило: конструктори для ініціалізації підоб'єктів викликаються з конструктора «великого» об'єкту у списку ініціалізаторів. У класу «сторінка», наприклад, маємо такий конструктор:

```
CPage::CPage( double w_, h_, m_ ) :
  m_width( w_ ),
  m_height( h_ ),
  m_margin( m_ ),
  m_text( m_, m_, w_-2*m_, h_-2*m_ )
{
}
```

Як видно, останньою в списку ініціалізаторів стоїть ініціалізація підоб'єкта `m_text` такими значеннями: прямокутник зсунуто на шири-

ну поля відносно горизонтального та вертикального краю сторінки (перші два аргументи), текстова область має ширину та довжину, меншу за розміри сторінки на подвійну ширину полів.

Якщо в списку ініціалізаторів немає виклика конструктору для підоб'єкта, то компілятор спробує проініціалізувати підоб'єкт викликом конструктора по замовчуванню. Але якщо в класі, до якого належить підоб'єкт, конструктора по замовчуванню немає, компілятор повідомить про помилку. Тим самим гарантується, що підоб'єкт у складі більшого об'єкта неодмінно буде проініціалізовано викликом якогось конструктора і отримає коректні значення членів-даних від самого початку свого існування: спробу залишити підоб'єкт без ініціалізації компілятор не пропустить.

1.11. Дружні функції і класи

Як було викладено вище, в більшості випадків бажано захищати члени-дані об'єктів від втручання сторонніх функцій. Але в іноді випадках все ж буває бажано деяким небагатьом функціям дозволити пряме звертання до членів об'єкта в обхід захисту. Це має сенс тоді, коли функція, хоча й зовнішня щодо класа (тобто не метод цього класу), дуже тісно пов'язана з класом за логікою побудови програми та згідно задуму програміста.

В таких випадках програміст, який пише оголошення класу, може оголосити ті чи інші функції *друзями* цього класу. Дружня функція має повний доступ до всіх членів класу, навіть прихованих за специфікатором **private**, тобто має такі само права доступу до членів, як і метод цього класу. Щоб оголосити функцію дргом класу, всередині оголошення класу треба записати її прототип з ключовим словом **friend**.

Наприклад, розглянемо дві однакові функції, `f` та `g`, які намагаються звернутись до прихованих членів класу «прямокутник». Але перша з них є другом класу, а друга — ні:

```
class CRect {
public:
  CRect( void );
  // далі інші методи ...
private:
```

```
  double m_x, m_y;
  double m_w, m_h;
// оголошення функції другом
  friend void f( CRect &r );
};

// дві функції: дружня та недружня
void f( CRect &r ) {
  r.m_x += 2; // гаразд
}

void g( CRect &r ) {
  r.m_x += 2; // помилка !!
}
```

Хоча два рядки в тілах функцій цілком однакові, але перший з них компілятор вважає правильним, а в другому зафіксує помилку. Справді, член `m_x` в класі `CRect` прихований, тому стороння функція `g` не має права до нього звертатися. Натомість функція `f` на правах друга може це робити.

Ще раз підкреслимо: якщо всередині огоошення класу перед прототипом функції стоїть слово **friend**, то це означає не оголошення методу класу, а те, що зовнішню функцію клас називає своїм другом, тобто дає їй право доступу до своїх прихованих членів. Дружнім до класу можна оголосити не лише функцію, але й інший клас: тоді всі його методи отримують повний доступ до членів даного класу.

Механізм дружніх функцій дещо послаблює механізми захисту даних, що виглядає не зовсім гарно з точки зору «чистої» об'єктноорієнтованої парадигми. На практиці користуватися дружніми функціями слід тоді, коли одночасно справджуються дві умови: (1) функція і клас (чи клас з іншим класом) пов'язані так тісно, що їх неможливо уявити порізно; (2) доступ до членів класу через інтерфейс (як того вимагає «чисте» ООП) відчутно збільшує затрати часу, коли від програмного коду вимагаєтсья висока швидкодія.

1.12. Покажчик this

Як неодноразово зазначалося вище, метод викликається завжди для певного об'єкту. Викликаний метод може обробляти змінні-члени то-

го об'єкту, до якого він викликаний. В деяких випадках буває бажано в коді метода отримати покажчик на той об'єкт, до якого цей метод в даний момент застосовується. Для цього в мові Сі існує спеціальне слово **this**. Воно може використовуватися лише в тілах методів (забігаючи наперед: методів, що не є статичними).

Для прикладу додамо в клас `CVector` метод під назвою `copyFrom`:

```
class CVector {
public:
  // ...
  CVector& copyFrom( const CVector& );
protected:
  int m_size;
  double *m_p;
};

CVector& CVector::copyFrom( const CVector &x ) {
  // якщо спроба копіювати в себе, то вийти
  if( &x == this ) return *this;
  // звичайний алгоритм копіювання
  if( x.m_size != m_size ) {
    delete[] m_p;
    m_p = new double[x.m_size];
    m_size = x.m_size;
  }
  for( int i = 0; i < m_size; ++i )
    m_p[i] = x.m_p[i];
  // повернути посилання на цей же об'єкт
  return *this;
}
```

Цей метод має аргумент типу посилання на вектор та копіює елементи вектора-аргумента до вектору, для якого цей метод застосовано. Іншими словами, якщо `u` та `v` — змінні типу `CVector`, то після операції `u.copyFrom(v)` обидва ці вектори будуть містити однакові значення компонентів.

Зрозуміло, що є один частковий випадок, коли копіювання можна не робити взагалі: це коли той об'єкт, для якого викликано метод, і той, що переданий йому в якості аргумента, співпадають: справді, об'єкт не зміниться, якщо його скопіювати з самого себе. Для пе-

ревірки цієї умови і знадобиться покажчик **this**. Тому в першому ж операторі метода перевіряється, чи не співпадає адреса об'єкта, переданого в якості аргумента, з адресою поточного об'єкта.

Нарешті, метод повертає посилання на той об'єкт, до якого його було застосовано (***this**). Це прийом, який часто використовуєтья в професійному програмування і дозволяє в одному рядку «чіплятиі» один на одний кілька викликів методів. Наириклад, до посилання, яке поверне метод `copyFrom`, можна одазу ж застосувати метод `print` того ж класу `CVector` (припустимо, що такий метод в класі є і що він друкує значення компонентів вектора):

```
u.copyFrom(v).print();
```

1.13. Константні методи

Деякі методи за своїм призначенням змінюють об'єкт, присвоюючи нові значення його змінним-членам. Скажімо, метод `move` класу `CRect` змінює значення членів `m_x` та `m_y`, а метод `resize` — члени `m_w` та `m_h`. Та є і такі методи, які не повинні жодним чином змінювати стан об'єкту — вони лише беруть з нього інформацію. В прикладі з класом `CRect` такими є методи `square` та `isIn`. В теорії програмування для методів першого різновиду закріпилася назва «модифікатори», для другого — «аналізатори» або «селектори».

В гарному професійному програмуванні вважається «хорошим тоном» в явному вигляді відмічати ті методи, які зобов'язуються не змінювати об'єкт, модификатором **const**. Це слово пишеться на кінці, після круглої дужки, що закриває список аргументів; його треба писати як в прототипі, так і в реалізації методу. Наприклад, покажемо переписану частину оголошення класу та оновлені реалізації двох методів:

```
class CRect {
public:
  // . . .
  double square( void ) const;
  int isIn( double x, double y ) const;
private:
  double m_x, m_y;
```

```
  double m_w, m_h;
};

double CRect::square( void ) const {
  return m_w * m_h;
}

int CRect::isIn( double x, double y ) const {
  return (x >= m_x) && (x <= m_x+m_w) &&
         (y >= m_y) && (y <= m_y+m_h);
}
```

З одного боку, чітке позначення методів-аналізаторів бажане з суто технічної точки зору. Та з іншого боку, і це більш важливо, вимога відокремлювати модифікатори від аналізаторів приносить користь самому програмісту, змушуючи його ретельніше продумувати логіку програми, чіткіше уявляти собі сенс та призначення кожного методу.

В мові Сі++ є правило, яке в деяких випадках вимагає, щоб всі методи, що застосовуються до об'єкта, були лише константними. Розглянемо ситуацію, коли об'єкт передається до деякої функції `f` в якості аргументу через покажчик чи посилання. Нехай функція `f` не має наміру змінювати цей об'єкт, тоді аргумент функції — покажчик або посилання повинен оголошуватися з модифікатором **const**, як того вимагає красивий стиль програмування:

```
void f( const CRect *p ) {
  // . . .
}
```

Тоді в тілі функції `f` викликати для об'єкта, на який вказує покажчик `p`, неконстантні методи, заборонено. Справді: оголосивши аргумент константним покажчиком, фукнція взяла на себе зобов'язання жодним чином не «псувати» переданий їй об'єкт. Значить, вона не має права чинити над об'єктом будь-які дії, що хоча б в принципі можуть його змінити. Зокрема, для об'єкта можна викликати лише ті методи, які не змінюють його стан — а це, звичайно ж, методи, позначені модифікатором **const**. Таким чином, якщо тіло функції містить оператори

```
void f( const CRect *p ) {
  cout << p->square() << endl;
  p->move( 1, 2 );
}
```

то перший з них компілятор схвалить, а в другому зафіксує помилку — для об'єкту, який оголошений незмінним, робиться спроба викликати метод, що може його змінити.

Наступне правило цілком очевидне: в реалізації самого константного методу покажчик **this** є константним покажчиком. Наприклад для класу `CRect`: якщо в тілі такого методу, як `move`, покажчик **this** має тип `CRect*`, то у тілі такого методу, як `square` — тип **const** `CRect*`.

1.14. Статичні члени

Матеріал цього розділу зручно пояснити, протиставивши його викладеному в попередніх розділах. Поглянемо ще раз на семантику оголошень класів і об'єктів. Оголошення класу є загальним описом того, які члени (змінні і методи) містяться в кожному об'єкті цього класу. Об'єкт має в собі набір змінних і може виконувати певний набір методів, перелік яких береться з оголошення класу; ці змінні і методи є членами даного об'єкту, його «власністю».

Мова Сі++ дозволяє оголошувати також змінні і методи, які є «власністю» не окремого об'єкту, а класу як цілого — їх прийнято називати *статичними*. Всі об'єкти даного класу «бачать» статичний член-змінну, але це одна й та сама змінна, спільна для всіх об'єктів класу. Так само і до статичного метода може звертатися будь-який об'єкт даного класу, але такий метод, на відміну від звичайного, застосовується не до конкретного об'єкту, а до класу як цілого. Зокрема, у статичному методі не має сенту покажчик **this**, статичний метод не може (як це роблять звичайні методи) звертатися по імені до нестатичних членів-змінних (бо у випадку звичайних методів мається на увазі, що це член-змінна поточного об'єкту, а у статичного методи поточного об'єкту немає).

Іншими словами, статичні змінні зручно використовувати, тоді, коли за сенсом задачі потрібно, щоб всі об'єкти одного класу могли «спілкуватися» між собою через доступ до спільних даних. При цьому механізми обмеження доступу забезпечують, що сторонні функції

не втрутяться без дозволу в ці дані.

Перейдемо до правил синтаксису. Перед оголошенням статичного члену (змінної чи методу) ставиться спеціальне слово **static**. Статичними неможна оголошувати конструктори та деструктори. Забігаючи наперед (розділ 2.6): метод не може бути одночасно статичним і віртуальним (тобто модифікатори **static** та **virtual** один з одним не суміщуються). Статичну змінну-член потрібно не тільки оголосити, але й проініціалізувати. Для цього треба окремим рядком (за межами оголошення класу) записати тип, повне статичної ім'я змінної-члена (складається з імені класу, знаку :: та власле імені), знак присвоювання та початкове значення (останнє повинно бути задано таким виразом, який можливо обчислити на етапі компіляції, не чекаючи запуску програми). Розберемо такий приклад:

```
class A {
public:
  A(void);
  ~A(void);
  // оголошення статичноо методу
  static int objCount(void);
private:
  // оголошення статичної змінної
  static int m_objCount;
};

// ініціалізація статичного члена
int A::m_objCount = 0;

A::A(void) {
  // звичайний метод звертається
  // до статичного члена
  ++m_objCount;
}

A::~A( void ) {
  --m_objCount;
}

int A::objCount(void) {
  return m_objCount;
```

```
}
```

Як видно з програмного тексту, всі об'єкти класу `A` ділять між собою змінну `A::m_objCount`. Її початкове значення 0, щоразу при створенні об'єкта значення нарощується на 1, а при знищенні об'єкту — зменшується. Це означає, що в кожен момент часу значенням цієї змінної є кількість наявних в пам'яті об'єктів класу `A`. Ця змінна прихована від стороннього програмного коду (наприклад, щоб сторонні функції не зіпсували лічильник об'єктів, присвоївши йому інше значення), але метод `objCount` дозволяє дізнатися у класу значення цієї змінної. Ці методи показує в дії наступна програма:

```
int main( void ) {
  cout << A::objCount() << endl;
  A a;
  cout << A::objCount() << endl;
  {
    A m[5];
    cout << A::objCount() << endl;
  }
  cout << A::objCount() << endl;
  return 0;
};
```

Коли спочатку виводиться значення лічильнику об'єктів, жодного об'єкта класу `A` ще не створено, тому надруковано буде число 0. Далі створюється один об'єкт, і наступний ряок програмного коду надрукує значення 1. Після того, як створиться масив, кількість об'єктів досягне 6, що й буде надруковано. Нарешті, оскільки масив є локальним для блоку кода у фігурних дужках, при виході з блоку його буде знищено, і останній оператор повідомить про наявність одного об'єкту.

Наостанок варто висвітлити роль статичних методів у більш широкому контексті. У фундаментальних дослідженнях з теорії програмування ставиться (і різними дослідниками по-різному вирішується) цікаве питання: чи можна кожен наявний в програмі *клас* в свою чергу теж вважати об'єктом? В тій самій мірі, в якій змінна `a` в наведеному прикладі є екземпляром класу `A`, чи не варто сам клас `A` (поруч з класами `CRect`, `CBook` та ін.) розглядати як екземпляр якогось класу `X`? Якщо клас `CRect`, скажімо, моделює поняття

«прямокутник», то клас `X` моделює поняття «клас». Його прийнято називати терміном «метаклас».

Аналогічне питання, тільки сформульоване мовою філософії, турбує найвидатніших мислителів людства протягом століть. Якщо є, скажімо, поняття «книга», представниками якого є «Війна і мир» та цей навчальний посібник з ООП, поняття «прямокутник», «число» тощо, то має сенс і поняття «поняття», представниками якого є і поняття книги, і поняття прямокутника і числа. Те ж саме в іншій інтерпретації: якщо набір схожих між собою речей (скажімо, тисячі різних конкретних книг) можна охопити спільною думкою і об'єднати в поняття (утворивши таким чином загальне поняття «книга"), то чи не є і окремі поняття також своєрідними «речами», які можна так само схопити думкою разом в понятті «поняття»?

Зрозуміло, що таке поняття займає за своєю сутністю сильно відрізняється від усіх інших понять, має цілком особливий логічний статус. Так само і клас, що моделює поняття «клас», якщо запровадити його в мову ООП, буде мати низку особливостей, що сильно відрізняють його від усіх інших класів. Зокрема, він має бути своїм власним екземпляром. Навколо метакласу виникає так багато проблем (і фундаментальних, і суто технічних), що автори багатьох мов ООП вважають за краще не вводити в мову жодних засобів для роботи з метакласами, повністю ігноруючи таку можливість. Статичні члени в мові Сі++ є дуже обережною спробою в дати сильно обмежену підтримку метакласів. Статичні методи вже дозволяють добитися певних практичних переваг метакласового програмування, та водночас ще не втягують в мову безліч пов'язаних з цим проблем.

Зв'язок між метакласами і статичними методами полягає в наступному. Так само як об'єкт звичайного класу має власні члени-змінні, що утворюють його стан, та методи, що утворюють його поведінку, так і клас як екземпляр метакласу повинен мати свої змінні і методи, що стосуються саме класу в цілому. В мові Сі++ немає метакласу як такого, але клас може мати власні (статичні) дані і методи, стан і поведінку.

1.15. Короткий огляд теоретичних основ

- З точки зору логічної структури програми класи є програмними моделями (загальних) понять, а об'єкти — моделями (одиничних) речей.
- На рівні реалізації клас в більшості об'єктно-орієнтованих мов (зокрема, в мові Сі++) — це створений програмістом тип даних, а об'єкт — значення такого типу.
- Клас являє собою опис того, які дані входять в кожен об'єкт цього класу, та які операції (функції) можуть застосовуватися до об'єкта. Ті і інші прийнято називати членами класу (відповідно, об'єкта); члени-функції також називають методами.
- В мові Сі++ і в більшості об'єктно-орієнтованих мов кожен об'єкт належить певному класу, тобто є екземпляром класу.
- Об'єкт-екземпляр певного класу має в своєму складі ті і лише ті члени-дані, що є у цьому класі; до об'єкту можна застосовувати лише ті операції, які визначені в його класі.
- Описуючи в класі перелік членів-даних та методів, можна вказати, які з них будуть доступні (як ще кажуть, видимі) ззовні, з інших функцій програми, а які становлять подробиці внутрішньої будови класу, тобто видимі лише для інших методів цього ж класу. Частіше за все загальнодоступними роблять більшу частину методів, а внутрішніми (прихованими) всі члени-дані та, можливо, деякі методи.
- Якщо об'єкт є програмною моделлю якоїсь речі, то члени-дані об'єкту моделюють параметри її стану, а загальнодоступні методи — її поведінку (тобто реакції речі на зовнішні впливи).
- Образно кажучи, програма спілкується з об'єктом виключно мовою викликів його загальнодоступних методів. Об'єкт сам знає свій поточний стан, сам вміє його змінювати у відповідь на звернення з зовнішнього світу, та нікому більше не довіряє контролювати свій стан.
- Конструктор це спеціальний метод, який викликається в момент створення об'єкту і призначений для його ініціалізації — тобто для присвоювання членам об'єкту початкових значень.

- Якщо в класі немає конструктора, то при створенні об'єкту цього класу лише виділяється ділянка пам'яті потрібного розміру (щоб в ній вмістилися всі змінні-члени об'єкта), а члени-змінні залишаються неініціалізованими.
- Конструктор в загальному випадку має аргументи, через них програма передає початкові значення і тим керує створенням об'єкту. В класі може бути як завгодно багато конструкторів, що відрізняються типами чи кількістю аргументів.
- Конструктор по замовчуванню — це особливий різновид конструктора, конструктор без аргументів. Саме він дозволяє створювати об'єкти, не передаючи для них вихідних значень. Зокрема, конструктор по замовчуванню використовується для ініціалізації об'єктів–елементів масиву.
- Деструктор — це спеціальний метод, який викликаєтья автоматично при знищенні об'єкту, безпосередньо перед тим, як система видалить його з пам'яті. Якщо в класі немає деструктора, то система просто видаляє об'єкт, не викликаючи якихось допоміжних дій.
- Деструктор в класі потрібен тоді, коли об'єкт захоплює динамічну пам'ять (чи якісь інші системні ресурси). Завдяки деструктору об'єкт в момент знищення зможе звільнити ці ресурси, які інакше лишилися б «мертвими» (тобто такими, які програма вже не може використовувати, але система вважає їх зайнятими).
- Конструктор копіювання — це особливий різновид конструктора, конструктор з одним аргументом типу посилання на об'єкт свого класу (наприклад, якщо клас має ім'я `C`, то його конструктор копіювання має аргумент типу **`const`** `C&`). Цей конструктор викликається щоразу, коли програмі треба створити копію об'єкта, наприклад, коли об'єкт передається у функцію в якості аргументу чи повертається як значення функції.
- Якщо конструктор копіювання не визначити власноруч, система намагатиметься копіювати об'єкти побайтно: члени-змінні об'єкта-копії отримають такі ж значення, як і члени-змінні об'єкта-оригіналу. Проте такий спосіб копіювання не підходить,

якщо серед членів об'єкту є покажчики на динамічно виділені області пам'яті — в такому випадку алгоритм копіюваннння об'єктів необхідно реалізувати самостійно.

- Якщо серед членів-даних об'єкта `x` класу `C` є об'єкти інших класів, то створення об'єкту `x` починається зі створення цих підоб'єктів (ціле не може виникнути, поки не створено його частини): перш, ніж відпрацює конструктор для об'єкта `x`, мають відпрацювати конструктори для об'єктів-частин. Якщо ці конструктори вимагають аргументів, то програмісту треба задати їх серед ініціалізаторів у конструкторі класу `C`.
- В оголошенні класу можна назвати деякі сторонні функції чи інші класи друзями даного класу. Дружні функції та методи дружніх класів мають повний доступ до всіх (навіть прихованих) членів даного класу. Іншими словами, дружність — це шлях до обходу механізмів захисту членів класу.
- Спеціальне слово мови Сі++ **`this`** може використовуватися лише в коді методів і означає покажчик на той об'єкт, для якого цей метод викликано. Зокрема, цей покажчик використовують, щоб метод повертав посилання на той об'єкт, до якого він застосовувався. Це, в свою чергу, дозволяє об'єднувати кілька викликів різних методів в ланцюжок.
- Методи в класі можна позначати як константні — такий метод бере на себе зобов'язання не змінювати стан об'єкта, до якого застосовується. Наявність спеціально позначених константних методів дозволяє транслятору відслідковувати коректність роботи з об'єктами, які самі мають модифікатор **`const`**.
- Статичними членами володіє не окремий об'єкт, а клас в цілому. До статичної змінної-члена мають спільний доступ всі об'єкти даного класу. Статичний метод викликаєтьс не у прив'язці до певного об'єкту, тому напряму за іменем може звертатися лише до статичних змінних-членів.

Розділ 2

Наслідування і поліморфізм

2.1. Абстрактні і конкретні поняття

Як зазначалося в розділах 1.2 та 1.4, сила ООП полягає в тому, що воно добре моделює дійсність і відображає зручні для людини способи розмірковувати про неї. Саме тому виклад основ ООП в попередньому розділі починався з філософського аналізу важливих рис дійсності і мислення. Перш ніж просунутися далі у вивченні технічних інструментів ООП, потрібно знов зануритися у філософські першооснови.

Класи в об'єктно-орієнтованих мовах програмування запроваджено для того, щоб моделювати *поняття*. Розглянемо детальніше категорію поняття і ті прийоми, якими користується людина, оперуючи поняттями.

Людина тримає в голові не безліч окремих, відірваних одне від одного понять, а *систему* понять, пронизану сіткою взаємних зв'язків. Зокрема, для того, щоб *зрозуміти*, засвоїти нове поняття, людині потрібно не просто осмислити його саме по собі, а включити, вписати його в систему вже наявних понять, встановити відношення і зв'язки нового поняття з вже відомими. Існує чимало відношень і зв'язків між поняттями, але особливо важливе місце серед них займає відношення *абстрактного та конкретного*, *загального та часткового*[1].

Щоб дати строге означення абстрактності та конкретності, треба спиратися на дві характеристики понять: зміст і об'єм. *Об'ємом поняття* називається сукупність всіх речей (матеріальних чи ідеальних), які підпадають під це поняття. *Змістом поняття* називається набір ознак, відображених в цьому понятті.

Наприклад, візьмемо поняття «трикутник». Його об'єм — це нескінченна множина всіх можливих трикутників. А зміст — набір та-

[1] Строго кажучи, абстрактне не зовсім те саме, що загальне, а конкретне — не те саме, що часткове. Але в рамках даного посібника дозволимо собі таке спрощення і будемо вважати вказані пари термінів синонімами

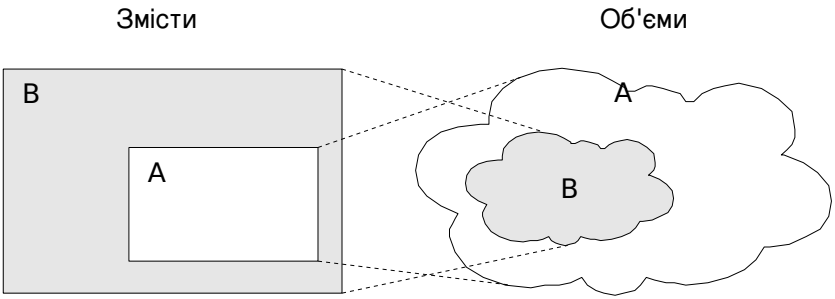

Рис. 2.1. Обернене співвідношення об'єму і змісту

ких ознак, як «бути геометричною фігурою на площині», «складатися з трьох точок (вершин) і трьох відрізків (сторін), що їх з'єднують», «мати суму внутрішніх кутів 2π» тощо. У поняття «парне невід'ємне число» об'єм — це нескінченна множина $\{0, 2, 4, \ldots\}$, а зміст складається з ознак «бути цілим числом», «бути числом, що дорівнює або більше від 0», «бути числом, яке без остачі ділиться на 2» та всіх ознак, які логічно випливають з них. Об'ємом поняття «студент першого курсу з програмної інженерії» є поіменний список першокурсників, а його зміст включає ознаки «бути людиною», «навчатися у вищому навчальному закладі» тощо.

Має місце *закон оберненого співвідношення* змісту і об'єму понять: чим ширший зміст поняття, тим вужчий його об'єм, і навпаки (рис. 2.1). Наприклад, візьмемо поняття «ціле число» і «ціле парне число». У другого поняття ширший зміст, бо порівняно зі змістом першого поняття додалася нова ознака «бути парним>", а об'єм, навпаки, вужчий, бо порівняно з об'ємом першого поняття вилучено числа $1, 3, 5, \ldots$

На цій основі тепер можна дати наступне означення. З двох понять більш абстрактним (або загальним) називається те, у якого ширший об'єм (і, значить, вужчий зміст), а більш конкретним (або частковим) — навпаки, те, у якого вужчий об'єм (і, значить, ширший зміст). Наприклад, поняття «ціле число» абстрактніше за поняття «парне ціле число», а те, в свою чергу, більш абстрактне, ніж поняття «невід'ємне парне ціле число». Звичайно ж, не для всіх пар

понять можна казати, що одне з них абстрактніше за друге. Наприклад, з двох понять «прямокутний трикутник» та «парне невід'ємне число» жодне не є абстрактнішим за друге.

Безпосередньо з означення слідує: якщо деяка річ (предмет) підпадає під деяке більш конкретне поняття, то вона підпадає і під більш абстрактне. Наприклад, кожне невід'ємне ціле число є водночас цілим числом, кожен прямокутний трикутник є трикутником, кожен студент є людиною і т.д.

З іншого боку: всі властивості чи ознаки, притаманні більш абстрактному поняттю, з необхідністю увійдуть до змісту конкретного поняття. Наприклад: відомо, що для поняття «трикутник» має місце властивість: «сума внутрішніх кутів трикутника становить 2π»; оскільки поняття «прямокутний трикутник» конкретніше за поняття «трикутник», то вказана властивість зберігає свою силу і для нього.

Вживаючи метафору це можна скажати і так: конкретне поняття *наслідує* весь зміст абстрактного.

Ця властивість відіграє надзвичайно важливу роль в процесах мислення. Вона дозволяє формулювати *загальні* твердження. Сила загального твердження в тому, що воно висловлюється один раз для абстрактного поняття, але автоматично і з необхідністю переноситься на будь-яке з безлічі більш конкретних понять. Згадаємо попередній приклад: теорему про суму кутів достатньо довести один раз для загального поняття «трикутник», щоб потім більш не обтяжувати себе, повторно доводячи цю теорему для рівнобедренних, прямокутних трикутників, трикутників з тупим кутом тощо.

Знову вдаючись до метафори, висловимо це так: завдяки абстрактним понятям і загальним твердженням ми можемо мислити про речі оптом, а не вроздріб, що, звичайно ж, зручніше і продуктивніше.

В процесі мислення величезну роль відіграють дві операції над поняттями: абстрагування та конкретизація. *Конкретизація* — це операція, яка дозволяє з більш абстрактного поняття побудувати більш конкретне, а *абстрагування* — протилежна операція, яка дозволяє на основі конкретного поняття побудувати більш абстрактне. Конкретизувати понття можна шляхом поповнення його змісту новими ознаками. Наприклад, з поняття «ціле число» можна отри-

Табл. 2.1. Відповідність термінів філософської логіки термінам ООП

Філософська логіка	**ООП і мова Сі++**
Поняття	Клас
Об'єм поняття	Всі об'єкти класу
Зміст поняття	Всі члени класу
Абстрактніше поняття	Батьківський клас
Конкретніше поняття	Породжений клас
Конкретизація поняття	Наслідування класу
Конкретне поняття в своєму змісті має всі ознаки, властивості, характеристики зі змісту абстрактного поняття та, можливо, доповнює їх своїми	Породжений клас має всі члени, оголошені в батьківському класі, а також, можливо, свої власні члени
Річ, яка підпадає під конкретне поняття, автоматично підпадає і під більш абстрактне	Об'єкт породженого класу одночасно є і об'єктом батьківського класу

мати різні конкретизації: «парне ціле число», «ціле число від 10 до 20» тощо. Абстрагуватися, відповідно, можня шляхом відкидання, вилучення деяких властивостей зі змісту поняття. Скажімо, взявши поняття трикутника і відкинувши ту його ознаку, що вершин у нього саме три, отримаємо поняття багатокутника на площині.

Нарешті, ми готові перейти від філософських питань до власне програмування. З того, що гарна технологія програмування прагне якнайкраще відобразити сутність буття і свідомості, дійсності і нашого способу думати про неї, випливає: ООП неодмінно має містити деякі засоби, інструменти, що моделюють зв'язок абстрактних і конкретних понять. В табл. 2.1 показано, які засоби ООП відповідають тим чи іншим філософським категоріям. Детальному опису механізму наслідування класів з технічної точки зору присвячено наступний розділ.

Таким чином, якщо окремо взяті класи є програмними моделями

понять, то наслідування класів за своїм призначенням є програмною моделлю мислительної операції конкретизації понять. Коли є клас, що моделює абстрактне поняття, то за допомогою наслідування з нього можна побудувати другий (породжений) клас, який є моделлю більш конкретного поняття.

Якщо зміст поняття визначається відображеними в ньому ознаками, властивостями, то у класі зміст моделюється сукупністю членівданих і методів. Так само, як конкретне поняття успадковує весь зміст абстрактного (і, можливо, доповнює успадковані ознаки новими, власними), так і породжений клас має всі ті члени (дані і методи), що й батьківський, та, можливо, додає до них свої нові члени.

2.2. Наслідування класів: синтаксис і семантика

З технічної точки зору наслідування класів це засіб, який дозволяє створити новий клас (породжений) на основі іншого, раніше написаного (батьківського) класу. Породжений клас має всі ті члени, що й батьківський клас (успадковує їх), а також деякі свої[1].

Розглянемо правила оголошення породженого класу і роботу з ним на прикладі. Щоб не ускладнювати виклад одночасним розбором усіх тонкощів, почнемо з найпростішого випадку: всі члени обох класів відкриті; у класів немає конструкторів (створення об'єктів не супроводжується додатковими діями).

```
1  class A {
2  public:
3    int u;
4    void f( int );
5  };
6
7  class B : public A {
8  public:
9    int v;
10   int g( void );
11 };
```

[1] В ООП склалися різні термінологічні традиції: наслідування називають також, успадкуванням чи породженням класів; батьківський клас можна називати базовим класом або надкласом; синонімами також є терміни ≪породжений клас≫, ≪успадкований клас≫, ≪дочірній клас≫, ≪підклас≫, ≪похідний клас≫

```
12
13 void A::f( int k ) {
14   u += k;
15 }
16
17 int B::g( void ) {
18   --v;
19   return u+v;
20 }
```

В оголошенні класу `B` показано, що він породжений від класу `A` (рядок 7). Ім'я батьківського класу вказується через двокрапку разом зі специфікатором. Щодо ключа (в даному прикладі **public**), його сенс пояснимо згодом в цьому розділі.

Членами класу `B` є: змінна `u` і метод `f`, успадковані від батьківського класу `A`, та змінна `v` і метод `g`, оголошені в самому класі `B`. Це означає, що при роботі з об'єктом класу `B` можна звертатися і до його власних членів, і до тих, що дісталися йому у спадок. Зокрема, це показано в методі `g`, який звертається до обох членів-змінних, власної та успадкованої. Зі сторонніх функцій також можна звертатися як до власних, так і до успадкованих членів об'єкту (якщо вони відкриті):

```
// оголошується об'єкт породженого класу
B y;
// робота з успадкованими членами
y.u = 0;
y.f(1);
// робота з власними членами
y.v = 8;
cout << y.v();
```

Як вже було пояснено вище, члени класу, оголошені зі специфікатором **public**, доступні для всіх функцій та методів, тобто звертатися до цих членів можна з будь-якого місця програми. Члени, оголошені в класі зі специфікатором **private**, доступні лише для методів цього ж класу і ні для яких інших функцій чи методів. На практиці дуже потрібним виявляється ще один специфікатор **protected** (англ. захищений). Члени класу, оголошені з цим специфікатором, доступні не лише для методів свого класу, але й для методів породжених від

Табл. 2.2. Доступність членів класу для різних функцій. Значення специфікаторів доступу

Різновид функції	Специфікатор		
	public	protected	private
Метод цього ж класу	+	+	+
Метод породженого класу	+	+	—
Всі інші функції	+	—	—

Табл. 2.3. Обмеження доступу до успадкованих членів. Ключі наслідування

Ключ	Специфікатор в базовому класі		
	public	protected	private
public	public	protected	недоступно
protected	protected	protected	недоступно
private	private	private	недоступно

нього класів, та недоступні для інших, сторонніх функцій та методів. Таким чином, всю система специфікаторів доступу в мові Сі++ можна описати, як показано в табл. 2.2.

Пояснимо тепер роль ключа **public** в рядку 7. Як вище було сказано, при наслідуванні члени батьківського класу стають членами породженого класу. Разом з тим, кожен член породженого класу повинен мати той чи інший рівень доступності (**public**, **protected** або **private**). Засоби мови Сі++ дозволяють породженому класу, наслідуючи члени від батьківського, підвищувати ступінь їх захищеності. Наприклад, можна зробити так, щоб всі успадковані члени, які у батьківському класі мали рівень доступності **public** та **protected**, у породженому класі стали прихованими (**private**). Для цього і призначений ключ — одне з трьох слів **public**, **protected** або **private**, яке вказується перед іменем батьківського класу. В табл. 2.3 показано, який рівень доступності матимуть в породженому класі успадковані члени, коли дано їх рівень доступності в батьківському класі та ключ. З таблиці, зокрема, видно, що коли при наслідуванні застосовано ключ

public, як у наведеному прикладі, то **public**-члени батьківського класу стають **public**-членами породженого класу, **protected**-члени базового класу — **protected**-членами породженого класу, а члени, які у батьківському класі мають рівень **private**, для методів породженого класу взагалі недоступні. Останнє важливо: хоча об'єкт породженого класу і містить у собі **private**-члени батьківського класу, але не може до них звернутися зі своїх методів (хоча, звичайно ж, може викликати успадкований від батьківського класу метод, який сам звернеться до тих членів).

Сказане проілюструємо прикладом. Нехай дано оголошення класів:

```
class C {
public:
  int f( int );
protected:
  int g( int );
private:
  int h( int );
};

class D1 : public C {
public:
  int s( int );
};
```

Розберемось, які з наведених нижче рядків відповідають правилам мови Сі++, а які помилкові. Для початку подивимось, що дозволено робити методу породженого класу з успадкованими членами:

```
1 int D1::s( int k ) {
2   int x = f(k);     // гаразд
3   int y = g(k+1);   // гаразд
4   int z = h(k+2);   // помилка !!
5   return x+y+z;
6 }
```

У рядку 2 метод породженого класу звертається до члену `f` батьківського класу. Цей метод оголошений в класі `C` зі специфікатором **public**, отже до нього має право звертатися будь-яка функція, в тому числі і метод `s` породженого класу. В рядку 3 метод звертає-

ться до члену g, який в базовому класі оголошено зі специфікатором **protected**. Згідно табл. 2.2, методи породженого класу мають право звертатися до такого успадкованого члену. Нарешті, в рядку 4 компілятор зафіксує помилку. Хоча метод h і успадкований від батьківського класу, але в тому класі він оголошений зі специфікатором **private**. Це означає, що лише методи самого класу C можуть до нього звертатися.

Оскільки при наслідуванні задано ключ **public**, то й успадкований класом D1 **public**-метод f класу C в породженому класі також набуває рівень доступності **public**. Два інших успадкованих методи для сторонньої функції невидимі:

```
1 int main( void ) {
2   D1 d;
3   cout << d.f(0); // гаразд
4   cout << d.g(1); // помилка !!
5   cout << d.h(2); // помилка !!
6   return 0;
7 }
```

Уявімо тепер замість класу D1 клас D2, єдина відмінність якого полягає в тому, що при наслідуванні задано ключ **protected**:

```
class D2 : protected C {
public:
  int s( int );
};
```

Успадковані від батьківського класу методи f та g в породженому класі набувають рівня доступу **protected**, а успадкований метод h в породженому класі, як і раніше, недоступний. Це означає, що ситуація з викликом успадкованих методів з методу s залишається без змін (перші два викликати можна, третій ні), а доступність успадкованих методів для зовнішньої функції змінюється таким чином:

```
1 int main( void ) {
2   D2 d;
3   cout << d.f(0); // помилка !!
4   cout << d.g(1); // помилка !!
5   cout << d.h(2); // помилка !!
6   return 0;
7 }
```

Вище, коли розглядалися основи філософської логіки, ми переконалися, що річ, яка підпадає під конкретне поняття, тим самим підпадає і під більш абстрактне поняття (наприклад, кожен студент є людиною, кожен прямокутний трикутник є трикутником тощо). Прямим наслідком і втіленням цього принципу є наступне правило ООП: кожен об'єкт породженого класу може водночас розглядатися і як об'єкт батьківського класу. Зокрема це означає, що *змінній, типом якої оголошено батьківський клас, можна в якості значення присвоювати об'єкт породженого класу*, так само і для покажчиків. Іншими словами, *тип об'єкта або покажчика може перетворюватися знизу вгору* (від породженого класу до батьківського). Наприклад, розглянемо такі оголошення і присвоювання:

```
C  x, *p;
D1 y, *q = new D;
x = y; // вгору, гаразд
p = q; // вгору, гаразд
y = x; // вниз, помилка !!!
q = p; // вниз, помилка !!!
```

У перших двох операторах присвоювання відбувається дозволене правилами мови (та законами філософської логіки) перетворення типу об'єкта знизу вгору. В правій частині присвоювання стоїть об'єкт (покажчик на об'єкт) породженого класу, а в лівій — змінна типу об'єкта (покажчика на об'єкт) базового класу. В двох останніх присвоюваннях навпаки, робиться спроба об'єкт батьківського класу (відповідно, покажчик) з правої частини присвоїти у змінну типу об'єкта (покажчика на об'єкт) породженого класу, що заборонено — в цих рядках компілятор повідомить про помилку.

Дане правило можна пояснити не лише з логіко-філософських, але й з суто технічних позицій. Коли присвоювання іде вгору, у об'єкта з правої частини є всі ті члени, що й у об'єкта зліва (та, можливо, додаткові), тому *всі* члени-змінні об'єкту з лівої частини присвоювання набувають певних значень. Якщо ж уявити, що присвоювання іде вниз, тобто в лівій частині стоїть об'єкт породженого класу, то частині його членів значення присвоїтися не можуть: у об'єкта з правої частини відповідних членів немає. Для прикладу розглянемо два дуже простих класи і відповідні об'єкти:

```
class A {
public:
  int u;
};
class B : public A {
public:
  int v;
};
A a; // об'єкт батьківського класу
B b; // об'єкт породженого класу
```

Розглянемо присвоювання

```
a = b; // вгору
```

Об'єкт `b` породженого класу містить в собі змінну-член `u`, оголошену в батьківському класі та свою власну член-змінну `v`. Об'єкт `b` батьківського класу має лише член-змінну `u`. Тому в цьому операторі значення члена `u` з об'єкта`b` буде присвоєно члену `u` в об'єкт `a`; значення члена `v` об'єкта `b` з правої частини просто проігнорується, що є нормальним. Якщо ж взяти присвоювання

```
b = a; // вниз
```

то члену `v` об'єкта `a` ніде взяти своє нове значення, бо в об'єкта у правій частині немає такого члена. Як наслідок, присвоювання такого вигляду мають бути заборонені.

2.3. Простий приклад

Покажемо, як на практиці застосувати наслідування для побудови більш-менш досконалої програмної моделі понять.

Нехай клас `CFigure` моделює поняття «геометрична фігура» (на площині). Довільна геометрична фігура має лише два числові параметри: координати, що характеризують розташування фігури. Інших параметрів у якоїсь (невідомо, якої саме) фігури немає: скажімо, неможна заздалегідь сказати, що у неї є радіус, бо фігура може не бути колом, неможна також сказати, що у довільної фігури є діагональ, гіпотенуза тощо. Тому в класі `CFigure` оголошуються члени-змінні `m_x` та `m_y` дійсного типу.

Клас `CRect` моделює поняття «прямокутник». Поняття прямокутника конкретніше за поняття геометричної фігури — кожен прямокутник є фігурою. Зокрема, до змісту поняття «прямокутник» входять числові параметри — координати місця розташування, запозичені зі змісту поняття «фігура». Відповідно, клас `CRect` наслідує від класу `CFigure` члени-змінні `m_x` та `m_y`. Крім того, прямокутник характеризується довжиною та висотою (як і раніше, вважаємо, що його сторони паралельні координатним осям).

Що стосується методів, то оскільки будь-яка фігура має координати, то для цього абстрактного поняття вже можна визначити операцію «перемістити на вектор $(\Delta x, \Delta y)$». Зрозуміло, її програмною моделлю є метод, який треба оголосити в класі `CFigure`. Тоді цю операцію успадкує і клас `CRect` (та інші успадковані від `CRect` класи, що моделюють такі конкретні поняття, як «коло», «трикутник» тощо. Операцію «змінити розмір в k разів» має сенс саме для того поняття, у якому визначено характеристику «розмір». Таким чином, ця операція програмно моделюється методом, оголошеним у класі `CRect`[1].

Продумаємо специфікатори доступу. Методи «перемістити» та «змінити розмір», зрозуміло, мають бути загальнодоступними, щоб програма могла обробляти фігури, користуватися ними. Щоб програма не могла в обхід методів самовільно маніпулювати змінними-членами, доступ до них слід обмежити специфікаторами **`private`** або **`protected`**. Виходячи зі змісту задачі, логічно припустити, що по мірі подальшої розробки методам породжених класів може знадобитися доступ до координат чи розмірів, тому обираємо специфікатор **`protected`**.

В обох класах запровадимо конструктори. Конструктор по замовчуванню нехай задає фігурі початкові координати $(0, 0)$, а прямокутнику ще й розміри 1×1. Для фігури конструктор з аргументами отримує через аргументи початкові значення координат. Для прямокутника конструктор з чотирьма аргументами отримує початкові значеня координат та розмірів; конструктор з двома аргументами отримує лише початкові значення розмірів, а координатам встанов-

[1] треба зізнатися, що тут ми допустили грубе спрощення; в розділі 2.6 побачимо, що є сенс оголошення такого методу винести в батьківський клас, а вже реалізацію розмістити в породженому класі, в якому визначено розмір.

люються значення $(0, 0)$.

Нагальної потреби в деструкторах в даній задачі немає, бо ні абстрактна фігура, ні прямокутник не захоплюють динамічну пам'ять та інші системні ресурси. Але деструктори все ж оголосимо: по-перше, вони знадобляться в наступному розділі для ілюстрації важливих правил мови, а по-друге їх варто зарезервувати з огляду на можливі подальші розширення даної системи класів.

Нижче наведено оголошення класів, які випливають з наведених вище міркувань. Тіла методів не наводимо за очевидністю. Згодом цей приклад будемо продовжувати та нарощувати.

```
class CFigure {
public:
  CFigure(void);
  CFigure(double x_, double y_);
  ~CFigure(void);
  void move(double dx, double dy);
protected:
  double m_x, m_y;
};

class CRect : public CFigure {
public:
  CRect(void);
  CRect(double w_, double h_);
  CRect(double x_, double y_, double w_, double h_);
  ~CRect(void);
  void resize(double k);
protected:
  double m_w, m_h; // ширина та висота
};
```

Від класу `CFigure` можна породити, крім класу `CRect`, також класи `CCircle` (коло), `CTriangle` (трикутник) і т.д. Всі вони успадковують від батьківського класу оголошення членів-змінних `m_x` та `m_y` і метод `move`. На цьому прикладі видно важливу перевагу ООП: характеристики чи властивості, спільні для кількох класів, можна не переписувати кілька разів, повторюючи схожі оголошення в кожному класі, а один раз описати в батьківському класі та скільки завгодно разів успадковувати ці описи в породжених класах. Так ООП

дозволяє економити зусилля програміста та підтримує багаторазове використання коду (reusing).

2.4. Особливості конструкторів і деструкторів при наслідуванні

Як було вже сказано в розділі 1.7, з метою надійності в «чистому» ООП вимагається, щоб в момент створення об'єкта всі його члени-змінні набували конкретних значень (ініціалізувалися). Для ініціалізації призначений спеціальний метод, конструктор. Ініціалізація об'єкта породженого класу має певні особливості, бо такий об'єкт має як свої власні змінні-члени, так і успадковані від батьківського класу.

Перш за все зазначимо, що конструктори і деструктори це виняток з правила, за яким методи батьківського класу стають методами породженого класу: на відміну від інших методів, конструктор та деструктор, оголошені в батьківському класі, не стають конструктором і деструктором для породженого класу. Це легко зрозуміти, бо успадкований конструктор нічого не може знати про ініціалізацію змінних-членів, оголошених в породженому класі, а деструктор — про їх коректне знищення.

Загальний принцип, з якого випливають всі конкретні правила роботи з конструкторами, формулюється так. Кожен клас турбується про ініціалізацію власних членів (іншими словами, конструктор самого породженого класу має ініціалізувати лише члени-змінні, оголошені в цьому породженому класі; для ініціалізації успадкованих змінних-членів він має викликати конструктор батьківського класу).

Логічність та природність даного правила зрозуміти легко: члени батьківського класу можуть бути взагалі недоступними для методів породженого класу (якщо в базовому класі вони мають рівень доступа **`private`**), тому лише конструктору батьківського класу можна довірити їх ініціалізацію.

Якщо розглянути питання глибше, можна побачити, що в основі цього правила лежить і більш фундаментальна причина — для цього треба взяти до уваги, *яким чином* члени батьківського класу стають членами породженого класу, який спосіб їх фізичного розміщення в пам'яті. З точки зору реалізації *об'єкт породженого класу містить*

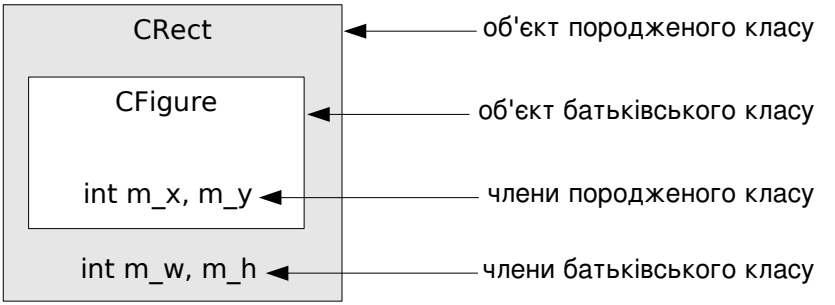

Рис. 2.2. Будова об'єкта породженого класу

в собі об'єкт батьківського класу, рис. 2.2; іншими словами, наслідування реалізується на основі включення об'єкту. За загальним правилом (розділ 1.10) підоб'єкт — частина більшого об'єкта має ініціалізуватися раніше, ніж сам цей об'єкт. Таким чином, при створенні об'єкта породженого класу спочатку викликається конструктор базового класу, а потім конструктор подорженого класу.

Що стосується деструкторів, то тут треба мати на увазі ще одне важливе правило: процес знищення об'єкту є дзеркальним відображенням процесу його створення (іншими словами, порядок виклику деструкторів протилежний до порядка виклику конструкторів). Це означає, що при знищенні об'єкту породженого класа спочатку викликається деструктор породженого класу, а потім деструктор батьківського. Основу цього правила зрозуміти легко: треба згадати, що об'єкт базового класу є підоб'єктом (складовою частиною) об'єкта породженого класу. Знищувати підоб'єкт неможна, доки не демонтовано більший об'єкт, до якого він належить.

Покажемо роботу конструкторів та деструкторів породженого класу на прикладі. В розділі 2.3 оголошено два класи: батьківський моделює поняття фігури, а породжений — поняття прямокутника. В класах оголошено конструктори та деструктори. Дамо цим конструкторам конкретні означення, передбачивши друк повідомлень про створення та знищення кожного об'єкта.

```
1 CFigure::CFigure(void) : m_x(0), m_y(0) {
2   cout << "фігуру␣створено␣по␣замовчуванню" << endl;
```

```
3  }
4
5  CFigure::CFigure(double x_, double y_) :
6  m_x(x_), m_y(y_) {
7    cout << "фігуру␣("
8         << m_x << "," << m_y
9         << ")␣створено" << endl;
10 }
11
12 CFigure::~CFigure(void) {
13   cout << "фігура␣("
14        << m_x << "," << m_y
15        << ")␣знищується" << endl;
16 }
17
18 CRect::CRect(void) :
19 /* увага: тут викликається конструктор
20 по замовчуванню батьківського класу!*/
21 m_w(1), m_h(1) {
22   cout << "прямокутник␣створено␣"
23        << "по␣замовчуванню" << endl;
24 }
25
26 CRect::CRect(double w_, double h_) :
27 /* увага: тут викликається конструктор
28 по замовчуванню батьківського класу!*/
29 m_w(w_), m_h(h_) {
30   cout << "прямокутник␣"
31        << m_w << "x" << m_h
32        << "␣створено" << endl;
33 }
34
35 CRect::CRect(
36   double x_, double y_,
37   double w_, double h_
38 ) :
39 // явний виклик конструктора базового класу
40 CFigure( x_, y_ ),
41 m_w(w_), m_h(h_) {
42   cout << "прямокутник␣"
```

```
43          << m_w << "x" << m_h
44          << "␣створено" << endl;
45  }
46
47  CRect::~CRect(void)
48  {
49    cout << "прямокутник␣"
50          << m_w << "x" << m_h
51          << "␣знищується" << endl;
52  /* увага: тут викликається
53  деструктор батьківського класу! */
54  }
```

Конструктори та деструктор класу `CFigure` коментарів не потребують. Інтерес становлять конструктори і деструктор класу `CRect`. Почнемо з конструктора з чотирьма аргументами (рядок 35 і далі). В процесі створення об'єкту породженого класу створюється і об'єкт батьківського класу. Для цього з конструктора класу `CRect` викликається конструктор класу `CFigure`. В даному прикладі два перших аргументи конструктора породженого класу просто передаються до конструктора батьківського класу. Після того, як конструктор батьківського класу відпрацює (проініціалізує члени-змінні `m_x` та `m_y` та надрукує повідомлення про створення фігури), ініціалізуються власні члени породженого класу — змінні `m_w` та `m_h` — і виконується тіло конструктора (друк повідомлення).

Розглянемо тепер конструктор з двома аргументами (рядок 35 і далі). В ньому не прописано виклик конструктора базового класу. Але в базовому класі є конструктор по замовчуванню. За загальним правилом конструктор по замовчуванню викликається тоді, коли об'єкт створити треба, а жодних початкових даних для цього не задано. Після того, як відпрацює конструктор по замовчуванню класу `CFigure`, ініціалізуються члени-змінні `m_w` та `m_h` класу `CRect` та друкується повідомлення, що прямокутник створено. Схожим чином працює і конструктор класу `CRect` по замовчуванню (рядок 18 і далі).

Візьмемо програму

```
int main( void ) {
  CRect r(10,20,8,4);
  return 0;
```

```
}
```

Згідно викладених вище правил результатом її виконання стане видача таких повідомлень:

```
фігуру (10,20) створено
прямокутник 8x4 створено
прямокутник 8x4 знищується
фігура (10,20) знищується
```

2.5. Множинне наслідування і його проблеми

Вище розглядалося т.зв. *просте* наслідування, тобто таке, коли породжений клас породжується від одного батьківського. Та в принципі клас може наслідуватися одночасно від двох чи кількох батьківських класів. Спершу треба розглянути логіко-філософську основу цього явища: для яких сторін дійсності та мислення множинне наслідування є адекватною програмною моделлю.

Вище вже зазначалося, що наслідування класів є спробою програмно змоделювати відношення «абстрактне-конкретне» між поняттями. Та конкретне поняття може бути конкретизацією одночасно кількох абстрактних. Візьмемо для прикладу поняття «біла річ» (в об'єм входять аркуш паперу, сніг, стеля тощо) та «квадратна річ». Тоді поняття «біла квадратна річ» є конкретизацією їх обох: до його об'єму входять лише ті речі, які входять до об'ємів одночасно обох понять «білої» і «квадратної» речі, а зміст містить як все зі змісту поняття білої речі, так і все зі змісту поняття квадратної речі.

При множинному наслідуванні членами успадкованого класу стають члени всіх батьківських класів; крім того, в породженому класі можуть оголошуватися і свої власні члени. Синтаксис оголошення класу, породженого від кількох батьківських, суттєво не відрізняється від синтаксису простого наслідування. Батьківські класи разом з ключами доступу перераховуються через кому. Для кожного з батьківських класів можна вказувати свій ключ доступу. Наприклад:

```
class A1 {
public:
  A1( int );
  int f1( void );
```

```
protected:
  int x1;
};
class A2 {
public:
  A2( int );
  int f2( void );
protected:
  int x2;
};
class B : public A1, public A2 {
public:
  B( int );
  int g( void );
protected:
  int y;
};
```

Клас `B` породжено одночасно від двох батьківських класів: `A1` та `A2`. Тому членами класу `B` стають і члени класу `A1` (метод `f1` та змінна `x1`), і члени, успадковані від класу `A2` (відповідно, `f2` та `x2`), і до того ж власні члени `g` та `y`. Оскільки обидва батьківських класи успадковано з ключем **public**, то методи `f1` та `f2`, які у своїх класах мали загальнодоступний рівень доступу, у породженому класі також будуть загальнодоступні, а змінні `x1` та `x2` в породженому класі залишаються з рівнем доступу **protected**.

Вище було сказано, що кожен об'єкт породженого класу містить в собі підоб'єкт батьківського класу. Тоді у випадку множинного наслідування об'єкт породженого класу має містити підоб'єкти кожного зі своїх батьківських класів. Так, об'єкт класу `B` з наведеного прикладу має будову, показану на рис. 2.3.

Як пояснювалося раніше, для ініціалізації підоб'єкта, успадкованого від батьківського класу, з конструктора породженого класу треба викликати конструктор батьківського. Тоді при множинному наслідуванні з конструктора породженого класу треба викликати конструктори *всіх* батьківських класів, наприклад:

```
B::B( int k_ ) :
  A1( k_*2 ), A2( k_+10 ), y( k_ )
{}
```

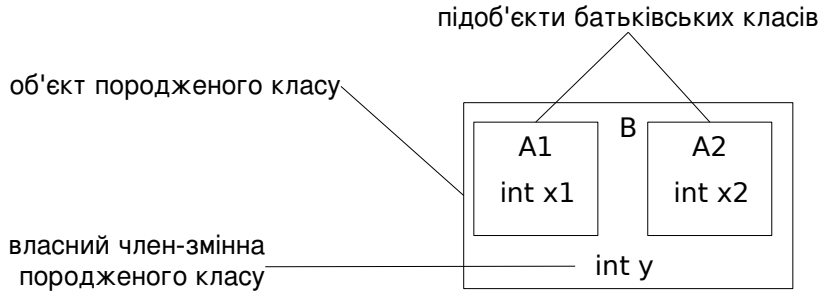

Рис. 2.3. Будова об'єкта при множинному наслідуванні

Множинне наслідування порівняно з простим дозволяє створювати набагато виразніші та більш гнучкі ієрархії класів, які краще відображають дійсність. Але разом з тим множинне наслідування тягне за собою низку серйозних труднощів. В деяких широко розповсюджених об'єктно-орієнтованих мовах (наприклад, Java та Об'єктний Паскаль) навіть взагалі немає множинного наслідування: автори цих мов вирішили, що вигоди, яких можна добитися за допомогою множинного наслідування, не варті пов'язаних з ним проблем.

Перша незручність полягає в тому, що в двох чи кількох батьківських класах можуть міститися члени з однаковими іменами. Наприклад, якщо трохи змінити попередній лістинг:

```
class A1 {
public:
  int f( void );
protected:
  int x;
};
class A2 {
public:
  int f( void );
protected:
  int x;
};
class B : public A1, public A2 {
public:
```

```
  int g( void );
};
```

Породжений клас успадковує від класу `A1` член-змінну з іменем `x`, а від класу `A2` — іншу змінну з таким же іменем. Звичайно ж, це дві різні змінні, але імена у них однакові. Виникає питання, як методам класу `B` розрізнити ці дві змінні: методи породженого класу цілком можуть працювати з успадкованими членами-змінними. Такий само конфлікт імен виникає між методом `f`, успадкованим від класу `A1`, та методом `f`, успадкованим від класу `A2`. Оскільки ці методи загальнодоступні, то не лише методам класу `B`, але й програмі, що його використовує, може знадобитися виклик одного з методів-тезок.

З цією проблемою впоратися нескладно. Достатньо такий член називати не просто по імені (в даному прикладі `x` чи `f`), але також і «по батькові»: наприклад `A1::x` чи `A2::f`, що означає «та змінна `x`, яка означена в класі `A1`» чи відповідно «той з двох методів `f`, який успадковано від класу `A2`».

Друга проблема серйозніша і вирішується не так просто. При множинному наслідуванні породжений клас може кілька разів успадкуватися від одного й того самого класу. Уявімо, що класи `B1` та `B2` породжені (простим наслідуванням) від деякого класу `A`, а клас `C` породжується від двох цих класів. За загальним правилом об'єкт класу `C` містить підоб'єкти своїх батьківських класів `B1` та `B2`. Але ж об'єкти класів `B1` та `B2` так само містять у собі кожен по підоб'єкту батьківського класу `A`. Таким чином, клас `C` двічі непрямо успадковує клас `A` і містить два підоб'єкти цього класу. Розглянемо приклад:

```
1  class A {
2  public:
3    A(int);
4  protected:
5    int u;
6  };
7
8  A::A( int u_ ) : u( u_ ) {
9    cout << "A␣␣створено␣з␣" << u << endl;
10 }
11
```

```
12 class B1 : public A {
13 public:
14   B1( int );
15 protected:
16   int v1;
17 };
18
19 B1::B1( int v1_ ) : A( v1_+20 ), v1( v1_ ) {
20   cout << "B1␣створено␣з␣" << v1 << endl;
21 }
22
23 class B2 : public A {
24 public:
25   B2( int );
26 protected:
27   int v2;
28 };
29
30 B2::B2( int v2_ ) : A( v2_*7 ), v2( v2_ ) {
31   cout << "B2␣створено␣з␣" << v2 << endl;
32 }
33
34 class C : public B1, public B2 {
35 public:
36   C( int );
37 protected:
38   int w;
39 };
40
41 C::C( int w_ ) : B1( w_*2 ), B2( w_+10 ), w( w_ ) {
42   cout << "C␣␣створено␣з␣" << w << endl;
43 }
44
45 int main( void ) {
46   C c(1);
47   return 0;
48 }
```

Об'єкт класу `C` має будову, показану на рис. 2.4. В цьому легко переконатися, якщо запустити програму: конструктори друкують повідомлення про створення кожного об'єкту, що дозволяє прослід-

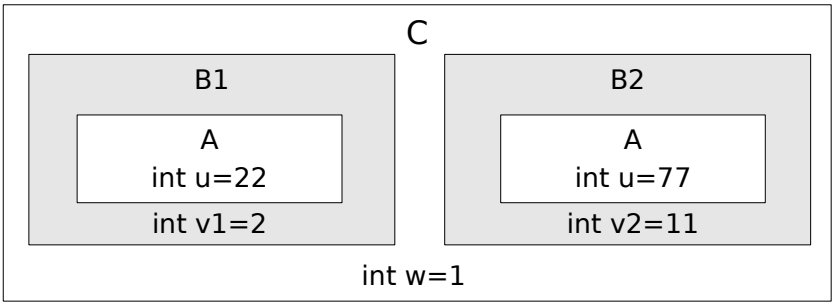

Рис. 2.4. Будова об'єкта при двократному наслідуванні одного класу

кувати за тим, скільки об'єктів створюється, яким класам вони належать і в якому порядку виникають. Виконання програми дає такий результат:

```
A  створено з 22
B1 створено з 2
A  створено з 77
B2 створено з 11
C  створено з 1
```

Звичайно ж, наявність в об'єкті двох різних підоб'єктів одного класу-предка незручна з суто технічної точки зору і безглузда з точки зору логіки. Було б бажано зробити так, щоб спільний предок наслідувався лише один раз. Мова Сі++ містить для цього спеціальний інструмент (хоча його можливості і обмежені, і повністю проблему він не вирішує). Клас, який згодом може «зіткнутися» із своїм двійником у породжених класах, треба при наслідуванні оголосити *віртуальним* батьківським класом. Нариклад, рядки 12 та 23 набувають вигляду

```
class B1 : virtual public A {
```

та

```
class B2 : virtual public A {
```

Якщо батьківський клас оголошено віртуальним, то об'єкт будь-якого породженого класу гарантовано матиме в собі лише один підоб'єкт цього класу — навіть якщо через множинне наслідування

успадкує його кількома шляхами. Так, в даному прикладі об'єкт класу C міститиме лише один підоб'єкт класу A.

Працюючи з віртуальним наслідуванням, треба постійно мати на увазі деякі тонкощі. При створенні об'єкта породженого класу першим завжди створюється підоб'єкт віртуального батьківського класу. В нашому прикладі — при створенні об'єкта класу C першим створюється той єдиний підоб'єкт класу A. На відміну від розглянутої вище ситуації з невіртуальним наслідуванням, коли конструктор для підоб'єкта класу A викликався двічі з конструкторів підоб'єктів B1 та B2, тут він викликається безпосередньо з конструктора класу A. Для цього треба змінити рядок 41 лістингу, помістивши туди виклик конструктора віртуального батьківського класу A:

```
C::C( int w_ ) :
  A(0), // виклик конструктора для віртуального предка
  B1( w_*2 ),  // виклик конструктора звичайного предка
  B2( w_+10 ), // те ж саме
  w( w_ )
{
  cout << "C␣␣створено␣з␣" << w << endl;
}
```

Після того, як створиться підоб'єкт віртуального батьківського класу A, починається створення підоб'єктів B1 та B2. В їх конструкторах також записано виклик конструктора класу A, але цей виклик виконуватися *не буде*, бо об'єкт класу A вже створено. Конструктори класів B1 та B2 лише проініціалізують свої власні члени. В цьому легко переконатися, виконавши змінену програму, її запуск дасть такий результат:

```
A  створено з 0
B1 створено з 2
B2 створено з 11
C  створено з 1
```

З одного боку, підоб'єкти B1 та B2 повинні містити кожен по підоб'єкту класу A, з іншого — підоб'єкт класу A всього один. Це означає, що підоб'єкти B1 та B2 *ділять* між собою один і той самий підоб'єкт. Таким чином, при віртуальному наслідуванні об'єкт має таку будову, як показано на рис. 2.5.

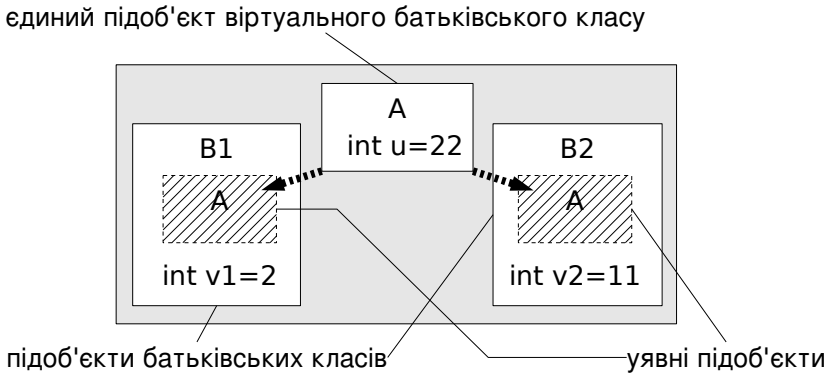

Рис. 2.5. Будова об'єкта при віртуальному двократному наслідуванні

2.6. Поліморфізм і віртуальні методи

Поліморфізм, поряд з інкапсуляцією та наслідуванням, становить третю і останню фундаментальну складову ООП. Саме на поліморфізмі побудовані найважливіші сучасні технології програмування.

Перш ніж вивчати конкретні технічні засоби, розглянемо, як завжди, важливі риси дійсності і мислення, для яких потрібна гарна програмна модель. Вище вже неодноразово підкреслювалося, що людині зручно мислити за допомогою абстрактних понять та їх конкретизацій. Наслідування класів пропонувалося вище саме як програмна модель конкретизації понять. Та наслідування втілює лише один різновид конкретизації: коли конкретне поняття наслідує без змін весь зміст абстрактного і доповнює його новими рисами і властивостями. Разом з тим є й інший спосіб конкретизації: коли конкретне поняття, наслідуючи риси і властивості від абстрактного, уточнює і поглиблює саме ці, унаслідувані риси.

Почнемо з прикладу. Поняття «молоток», «викрутка», «ножиці» є різними конкретизаціями поняття «інструмент». В загальному випадку інструментові (будь-якому; навіть коли не уточнюється, якому саме) має притаманна властивість *працювати*. Але ж різними конкретними інструментами працюють по-різному: молотком наносять удари, викруткою повертають гвинт, ножиціми ріжуть і т.д.

Таким чином, *конкретизуватися та абстрагуватися можуть не лише самі поняття, але й властивості понять.* В нашому прикладі три конкретні різновиди (молоток, викрутка і ножиці) об'єднуються під одним загальним поняттям «інструмент», і так само притаманні їм три різні конкретні дії (наносити удар, крутити та різати) об'єднуються під спільною назвою «працювати».

Нехай тепер x — змінна, яка може приймати значення в множині інструментів. Це означає, що в різні моменти часу і при різних обставинах її значеннями можуть бути і молоток, і викрутка, і ножиці. Розглянемо сенс висловлювання «працювати інструментом x». Зрозуміло, що коли значенням x є, скажімо, молоток, то сенсом цього висловлювання є «наносити удари», а коли x є ножицями, то те ж саме висловлювання означає «різати». Виходить, що *одне висловлювання, може, в залежності від конкретної ситуації, означати три різні дії.*

Важливо також загострити увагу на сенсі змінної x. Об'єкт x не є ні молотком, ні викруткою, ні ножицями, але в залежності від обставин може на деякий час проявляти себе як той чи інший із них. Іншими словами, x виступає як об'єкт заздалегідь невідомого різновиду, який, однак, згодом набуває конкретної форми — будь-якої з трьох можливих. Про об'єкт x можна сказати, що він *поліморфний* (з грецької — такий, що має багато форм).

Ще раз сформулюємо ті риси об'єктів, які роблять можливим поліморфізм і одразу перекладемо їх з мови філософської логіки на мову ООП, див. табл. 2.4.

Приклад, розглянутий вище з позицій філософської логіки, перетворимо на конкретний програмний текст. Клас `CTool` моделює поняття «інструмент» і має лише один метод `work` (працювати). Перед оголошенням методу стоїть службове слово **`virtual`**. Воно означає, що метод віртуальний, тобто призначений саме для реалізації поліморфної поведінки об'єктів. Особливістю віртуальних методів є спосіб виклику (див. нижче в цьому розділі). Реалізація цього методу в батьківському класі порожня — з точки зору змісту задачі ми не можемо її визначити якимось конкретним чином, бо про «інструмент взагалі» невідомо, яким саме чином він працює (див. також нижче про чисті віртуальні методи).

Класи `CHammer`, `CDriver` та `CScissors` моделюють, відповідно, поня-

Табл. 2.4. Що таке поліморфізм

Філософська логіка	**ООП і мова Сі++**
Дано загальне поняття і кілька більш конкретних	Дано батьківський клас і кілька породжених від нього класів
Для загального поняття визначено деяку дію f; але ця дія абстрактна: в зміст загального поняття входить лише сам факт наявності такої дії, а її сенс залишено невизначеним	В батьківському класі оголошено метод `f`, але оголошується він як *віртуальний*: в батьківському класі такий метод може взагалі не мати визначеного тіла (т.зв. *чистий віртуальний метод*)
Конкретні поняття наслідують весь зміст абстрактного; тому наслідують вони і наявність дії f	Породжені класи успадковують від батьківського саму наявність методу `f`
Кожне з конкретних понять наповнює отриману у спадок дію f своїм власним сенсом	В породжених класах для методу `f` визначаються різні реалізації
Береться змінна x, областю значень якої є об'єм абстрактного поняття. Згідно законів логіки, її значеннями можуть бути речі з об'єму будь-якого конкретного поняття. Про цю змінну невідомо заздалегідь, до якого з конкретних понять буде відноситися її значення; але коли змінна своє значення отримає, воно, звісно ж, буде конкретним	Оголошується змінна `p` типу покажчика на об'єкт батьківського класу. За викладеним вище правилом її значеннями можуть бути адреси об'єктів породжених класів. Заздалегідь невідомо, яким буде фактичний тип того об'єкту, на який вказуватиме покажчик `p`, але коли цій змінній певне значення буде присвоєно, це буде саме адреса об'єкту породженого класу
Якщо тепер від об'єкта x зажадати, щоб той здійснив дію f, то він зробить той з варіантів дії f, який означено у тому самому конкретному понятті, до якого в цей час належить об'єкт x	Якщо тепер для покажчика `p` викликати метод `f`, то викликано буде ту з його різноманітних реалізацій, яку визначено у породженому класі, до якого належить об'єкт, на який вказує `p`

ття «молоток», «викрутка» та «ножиці». В них віртуальний метод `work` оголошено заново, це означає, що метод в кожному з цих класів набуває нової реалізації. Реалізація методу `work` в кожному з трьох породжених класів друкує повідомлення про те, що робить цей конкретний інструмент.

```
class CTool {
public:
  virtual void work( void );
};

// молоток
class CHammer : public CTool {
public:
  virtual void work( void );
};

// викрутка
class CDriver : public CTool {
public:
  virtual void work( void );
};

// ножиці
class CScissors : public CTool {
public:
  virtual void work( void );
};

void CTool::work( void ) {
} // реалізація порожня

void CHammer::work( void ) {
  coput << "Б'є" << endl;
}

void CDriver::work( void ) {
  coput << "Крутить" << endl;
}
```

```
void CScissors::work( void ) {
  coput << "Ріже" << endl;
}
```

Далі йде основна програма, яка демонструє використання цих класів і поведінку поліморфного об'єкта.

```
int main( void ) {
  CTool *p = NULL;
  int k;
  cout << "Введіть число ";
  cin >> k;
  switch( k % 3 ) {
    case 0: p = new CHammer;
    case 1: p = new CDriver;
    case 2: p = new CScissors;
  }
  cout << "Що робить інструмент?" << endl;
  p->work();
  delete p;
  return 0;
}
```

Змінна `p` має тип покажчика на об'єкт батьківського класу. Але ж кожен об'єкт породженого класу є одночасно і об'єктом базового класу, тому покажчик `p` насправді може вказувати не лише на об'єкт власне типу `CTool`, а й на об'єкт якогось із трьох породжених класів. Від початку покажчик не вказує на жоден об'єкт.

Користувач вводить якесь число. Якщо воно ділиться на 3 без залишку, то створюється об'єкт класу «молоток», якщо з залишком 1 — створюється об'єкт-викрутка, а якщо залишок становить 2, то створюється об'єкт класу «ножиці». Покажчик на створений об'єкт присвоюється змінній `p`. Таким чином, якщо *оголошеним типом* покажчика `p` є клас `CTool`, то *фактичним типом* тепер є якийсь один з трьох породжених класів.

Насамкінець для покажчика `p` викликається метод `work`. На перший погляд могло б здатися: оскільки змінну `p` оголошено як покажчик на об'єкт класу `CTool`, то і викликатися має метод `CTool::work`, який нічого не робить. Але метод `work` віртуальний, а тому викликаєтья той з методів `work`, який відповідає фактичному типу об'є-

кта, на який вказує покажчик. Таким чином один і той самий рядок `p->work()` може, в залежності від ситуації, виконувати три різні функції, а одна змінна `p` може поводити себе в різних випадках як три різних об'єкти! Тому і кажуть, що поліморфізм і віртуальні функції забезпечують високу гнучкість програм.

Щоб остаточно і вичерпно висвітлити сутність поліморфізма і роль віртуальних методів, розглянемо ту ж саму програму, в якій з оголошень методу `work` в усіх класах видалено слово **`virtual`**. Тоді у функції `main` в рядку `p->work()` буде викликано метод, що відповідає *оголошеному*, а не фактичному типу об'єкта, на який вказує покажчик. Тобто, незалежно від того, об'єкт якого з трьох породжених класів буде створено, викличеться все одно порожній варіант методу `work`, успадкований від класу `CTool`.

У наведеному вище прикладі є одна деталь, яку можна вдосконалити. Звернімо увагу на порожню реалізацію методу `CTool::work`. Цей метод нічого не робить і не повинен нічого робити: з точки зору логіки задачі він реалізує роботу абстрактного (тобто невідомо, якого саме) інструмента. Іншими словами, порожня реалізація методу в батьківському класі відіграє роль «заглушки»; вона і не викликається ніколи в процесі роботи реальної програми, бо на ділі викликаються тільки реалізації, перевизначені в породжених класах.

Мова Сі++ містить спеціальний, більш елегантний, ніж порожнє тіло, спосіб оголосити метод-заглушку. Оголошуючи віртуальний метод в батьківському класі, можна позначити його як *чистий*, написавши на кінці його оголошення `=0`. Для чистого методу можна взагалі не описувати жодної реалізації, навіть порожньої. Оголошення чистого віртуального методу — це *обіцянка*, що згодом, у породжених класах, йому буде надано конкретну реалізацію.

Клас, у якого є хоча б один чистий віртуальний метод, називається *абстрактним класом*. В мові Сі++ є правило: неможливо створити об'єкт абстрактного класу. Це правило цілком зрозуміле: у такого об'єкту був би один або кілька методів з невизначеною реалізацією. Проте від абстрактного класу можна породжувати нові класи. Якщо в породженому класі успадкованому чистому віртуальному методу надати певну реалізацію (описати його тіло), то метод перестане бути чистим. Якщо у породженому класі після визначення реалізацій більш не залишається абстрактних методів, такий

клас перестає бути абстрактним. Можна оголошувати змінну типу покажчика на об'єкт абстрактного класу. Звичайно ж, її фактичним значенням може бути лише покажчик на об'єкт неабстрактного породженого класу.

Покажемо це на прикладі, змінивши попередній лістинг. В класі `CTool` зробимо чистим віртуальний метод `work`:

```
class CTool {
public:
  virtual void work( void ) =0; // чистий
};
```

Реалізацію цього методу з програми видалимо. Тепер клас `CTool` абстрактний. Оскільки в породжених класах `CHammer`, `CDriver` та `CScissors` метод реалізовано, ці класи не абстрактні. Робота показаної в прикладі програми залишається такою ж, як описано вище, єдина зміна полягає в тому, що ми змогли позбутися порожньої реалізації-заглушки в базовому класі.

2.7. Короткий огляд теоретичних основ

- В тій мірі, в якій програмування моделює, відображає дійсність, основні принципи та технічні прийоми об'єктно-орієнтованого стилю беруть свій початок у фундаментальних властивостях буття і мислення. Зокрема, наслідування класів є спробою програмно змоделювати притаманні людському мисленню відношення між абстраткними і конкретними поняттями.
- Конкретне поняття має багатший зміст, ніж абстрактне: включає весь зміст абстраткного і доповнює його новими подробицями. Відповідно, породжений клас має всі ті члени, що й батьківський, а також деякі нові.
- З технічної точки зору наслідування це інструмент, який дозволяє економити зусилля на написання нових класів: в реальній роботі клас можна не писати повністю наново, а породити від вже наявного (якщо той вирішує схожу задачу) і дописати в породженому класі лише ті деталі, яких бракувало в батьківському класі. Таким чином, наслідування — це ще однин крок в

розвитку технологій багаторазового використання (англ. reusing) програмного коду.

- Завжди **private**-члени батьківського класу недоступні для методів породженого класу, бо за означенням доступні лише для методів свого класу. Специфікатор **protected** означає, що члени класу доступні для методів свого класу та для методів породжених класів і недоступні для всіх інших (сторонніх) функцій.
- Породжений клас може обмежувати доступ до успадкованих членів. Для цього в оголошенні подродженого класу треба разом з іменем батьківського класу вказувати ключ (специфікатор). Сенс ключа такий: якщо, скажімо, при наслідуванні задано ключ **private**, то успадковані **public**-члени батьківського класу стануть **private**-членами породженого класу, і т.д.
- При ініціалізації об'єкту породженого класу (який містить і власні, і успадковані від батьківського класу члени) діє правило: кожен клас сам відповідає за ініціалізацію своїх членів. Це означає, що конструктор породженого класу сам ініціалізує лише змінні-члени, оголошені в цьому породженому класі, а для ініціалізації успадкованих членів викликає конструктор базового класу.
- З точки зору реалізації об'єкт породженого класу містить в собі об'єкт базового класу як складову частину. Тому створення об'єкту породженого класу починається зі створення цього захованого в ньому об'єкта базового класу (ціле не може виникнути раніше, ніж створено його частини). Це означає, що спочатку викликається конструктор базового класу, а потім відпрацьовує конструктор породженого класу.
- Знищення об'єкту породженого класу відбувається в зворотному порядку. Спочатку відпрацьовує деструктор породженого класу (і руйнує всі, наприклад, динамічні дані, що належать власне породженому класу, залишаючи успадкований підоб'єкт), а потім викликається деструктор базового класу (який руйнує і той підоб'єкт, що залишився).
- Мова Сі++ підтримує множинне наслідування (на відміну від багатьох інших мов ООП, де множинне наслідування заборone-

не, як потенційне джерело проблем): клас можна породжувати одночасно від кількох батьківських класів. Членами породженого класу стають члени всіх класів-батьків.

- Особливо багато клопоту в зв'язку з множинним наслідуванням завдають випадки, коли клас породжується від одного з класів-предків одночасно кількома різними шляхами. Наприклад, від класу `A` породжуються класи `B1` та `B2`, а від цих двох класів породжується клас `C` — тоді клас `C` успадковує дві копії класу `A`. Механізм віртуальних базових класів дозволяє частково впоратися з цією проблемою.
- *Поліморфізм* — це механізм, який дозволяє через одну й ту саму змінну чи вираз (типу покажчика на об'єкт батьківського класу) отримувати кілька різних варіантів поведінки (визначених в породжених класах); який саме варіант поведінки буде реалізовано, програмі та транслятору заздалегідь невідомо — це визначається динамічно, по ходу виконання програми.
- Для того, щоб отримати поліморфну поведінку, треба в батьківському класі оголосити один чи кілька *віртуальних* методів. В породжених класах їх реалізації можна перевизначити, в кожному породженому класі по-своєму. Віртуальний метод — це метод, який (на відміну від звичайного методу) викликається виходячи з фактичного (а не оголошеного) типу об'єкта.
- Якщо оголошеним типом змінної чи виразу `x` є покажчик (чи посилання) на об'єкт батьківського класу, то це означає, що фактичним значенням може бути і покажчик (посилання) на об'єкт породженого класу. Тоді якщо для змінної чи виразу `x` викликати метод, що в батьківському класі оголошено як віртуальний, то викликано буде ту його реалізацію, що відповідає фактичному типу об'єкта.
- Віртуальний метод можна оголосити *чистим*. Для такого методу в батьківському класі можна взагалі не описувати реалізацію. Клас, в якому є хоча б один чистий віртуальний метод, називається *абстрактним*. Забороняється створювати об'єкти абстрактного класу (ця заборона підтримується компілятором). Абстрактний клас можна використовувати лише як батьківський — породжувати від нього нові класи. Якщо в породже-

ному класі для всіх методів дано конкретну реалізацію, такий клас вже не буде абстрактним, і об'єкти цього класу вже можна створювати. Дозволяється оголошувати покажчики на об'єкти абстрактного класу — їх фактичними значеннями будуть покажчики на об'єкти неабстрактних породжених класів.

- На практиці часто роблять віртуальними деструктори. Коли об'єкт породженого класу створюється динамічно, а покажчик на цей об'єкт присвоюється в змінну `p` типу покажчика на об'єкт батьківського класу, то лише віртуальність деструктора гарантує, що при знищенні об'єкту операцією **`delete`** `p` буде викликано деструктор саме породженого класу.

Насамкінець дамо кілька важливих принципів, які узагальнюють все раніше вивчене як єдиний комплекс. ООП являє собою не просто набір технічних прийомів і мовних засобів, а цілісний підхід до програмування, який базується на певному способі бачити дійсність. ООП як парадигма програмування багато в чому протистоїть структурно-модульній парадигмі, втіленої в мові Сі. Зокрема, в структурно-модульному програмуванні функції відіграють чисто активну роль, а об'єкти даних — лише пасивну: функції можуть лише виконувати дії (над об'єктами), а дані — лише зазнавати на собі дії (з боку функцій). В мові Сі до функції при виклику можна передати об'єкт даних (чи покажчик на нього), функція робить з ним все, що хоче. В ООП, навпаки, програма звертається до об'єкта даних, щоб він виконав той чи інший зі своїх методів: тут вже об'єкт володіє функціями.

В цьому посібнику основи ООП викладено лише в одному з багатьох варіантів цієї парадигми. Для повноти картини треба хоча б кілька слів сказати про інший класичний спосіб викладу тих же принципів, про так звану метафору бесіди об'єктів. Уявімо собі комп'ютерну програму як своєрідний всесвіт, населений істотами (об'єктами). Кожна з цих істот відносно самостійна, знає якусь потрібну їй інформацію (члени-дані), має власні цілі, інтереси і самостійно діє (тобто виконує певний алгоритм). Час від часу (коли цього вимагає алгоритм, яким керується об'єкт) об'єкти можуть надсилати один одному *повідомлення*. Частіше за все це відбувається тоді, коли одному об'єкту потрібна послуга, яку (як він знає) може надати ін-

ший об'єкт. Повідомлення має відправника, адресата, тему (що саме відправник просить зробити адресата) і, можливо, деякі параметри.

Об'єкт-адресат, отримавши повідомлення, починає його обробку з того, що дивиться на його тему і вирішує, чи він взагалі спроможний виконувати саме таке прохання. Наприклад, якщо об'єкт вміє надавати послуги з відображення тривимірної графіки, а йому надходить повідомлення «відсортувати масив», він повідомить відправника, що його прохання помилкове. Якщо ж тема отриманого повідомлення правильна, об'єкт шукає серед своїх алгоритмів той, який підходить саме для цього повідомлення. Цей алгоритм обробки, спеціально призначений для певного різновиду вхідних повідомлень, і називають методом (власне, методом обробки повідомлень).

Історично ООП виникло саме як інструмент моделювання систем, подібних до описаної вище, коли об'єкти обмінюються повідомленнями та по-своєму реагують на них. В мові Сі++ посилку повідомлення від одного об'єкту до іншого втілено як виклик функції-члену, темі повідомлення відповідає ім'я цієї функції, а те, що в описаній бесіді об'єктів названо методом, постає як тіло функції-члена.

На цьому закінчено виклад принципових, фундаментальних категорій ООП. В наступних розділах описано переважно технічні засоби та інструменти, що збільшують потужність мови у практичних застосуваннях.

Розділ 3

Перевантаження операцій

3.1. Основні поняття

Гарна мова програмування має робити текст програми ясним і зрозумілим, якомога прозоріше відображати задум програміста. Зокрема, позначення, форма запису повинна не затьмарювати, а навпаки — висвітлювати сенс програми. В цьому розділі розглядається потужний інструмент, призначений саме для того, щоб зробити текст програми лаконічним і виразним.

На практиці часто доводиться програмувати обчислення зі складними математичними величинами — такими як вектори, матриці, комплексні числа. Самі ці величини можна зображувати в програмі об'єктами структурного типу або об'єктами класу — об'єкти мають зберігати в собі параметри, що разом характеризують математичну величину. Наприклад, згадаємо клас `CVector`, модель математичного поняття «вектор», наведений на с. 53: об'єкт цього класу містить в собі масив чисел – елементів.

Та як в програмі відобразити операції над математичними об'єктами: додавання, множення, порівняння на рівність тощо? В стилі мови Сі їх треба було б моделювати функціями. Скажімо, щоб над двома об'єктами, що зображують вектори, здійснити операцію додавання та суму помістити в третій об'єкт, треба було б написати щось на кшталт

```
vector_add( &z, &u, &v );
```

де `vector_add` — функція додавання векторів з трьома аргументами: покажчик на об'єкт, в який треба помістити суму, та покажчики на об'єкти-доданки.

Але такий спосіб запису операцій незручний. В гарній мові програмування речі, схожі за змістом, повинні мати і схожі позначення. Додавання, скажімо, цілих чисел має вигляд

```
c = a + b;
```

Було б дуже зручно і бажано, щоб додавання векторів теж позначалося знаком `+` замість показаного вище громіздкого імені `vector_add`.

На щастя, мова Сі++ має для цього засоби. Вона дозволяє програмісту власноруч визначати операції `+`, `-`, `*`, `/` та ін. так, щоб їх можна було застосовувати не лише до значень стандартних типів (цілого, дійсного тощо), але й до об'єктів власноруч створених типів (векторів, матриць, комплексних чисел і т.д.). Таким чином, обчислення з цими об'єктами в тексті програми виглядатимуть так само природно, як і обчислення зі стандартними типами **`int`** або **`double`**. Скажімо, показане вище додавання двох об'єктів-векторів та присвоювання результату в третій об'єкт набуває вигляду

```
z = u + v;
```

Звернімо увагу: такі операції, як додавання цілих чисел, додавання матриць та додавання векторів реалізуються зовсім по-різному (перша виконується безпосередньо електронною схемою процесора, дві другі виконуються за більш-менш складними алгоритмами, які визначає програміст). З точки зору реалізації це *різні* операції. Але з точки зору логіки прикладної задачі та людської уяви ці операції мають багато спільного: принаймні ми називаємо їх одниковим словом ≪додавання≫.

Явище, коли один знак (такий, як знак плюс, мінус тощо) позначає ту чи іншу операцію в залежності від типу об'єктів-операндів, до яких він застосований, називається *перевантаженням знаку операції* (для короткості також говорять просто ≪перевантаження операції≫). Так, у нашому прикладі перевантажені знаки ≪`+`≫ та ≪`=`≫: ці знаки можуть в різних виразах позначати і операції над стандартними типами **`int`** або **`double`**, і власноруч визначені програмістом операції над об'єктами класу `CVector`.

Починаючи вивчати перевантаження знаків операцій, варто згадати про перевантеження імен функцій (розділ 0.4). Там йшлося про те, що кілька різних функцій можуть мати однакове ім'я, вони мають розрізнятися за аргументами. Хіба що тоді йшлося про звичайні функції, у яких ім'я складається з літер, а аргументи пишуться в дужках.

В загальному випадку операцію можна перевантажити двома способами: звичайною функцією чи методом класу. Ім'я цієї функції

чи методу складається зі службового слова **operator** і знаку операції. Наприклад, функція чи метод, що перевантажує операцію додавання, повинна мати ім'я **operator+**.

Якщо операція перевантажується звичайною функцією, то операнди операції стають аргументами цієї функції. Скажімо, перевантажити знак плюс для векторів, можна функцією

```
CVector operator+( CVector, CVector );
```

Якщо така функція є, то рядок

```
z = u + v;
```

(де `u` та `v` — об'єкти класу `CVector`) компілятор автоматично перетворить операцію «плюс» на виклик цієї функції, підставивши перший доданок `u` до першого аргументу, а другий доданок `v` — до другого аргументу. Іншими словами, цей рядок спрацює так само, як спрацював би рядок

```
z = operator+(u, v);
```

Другий спосіб перевантаження операції — методом класу. Для бінарних операцій (тобто операцій з двома операндами, як додавання або множення) перший операнд операції стає тим об'єктом, для якого викликається метод, а другий — єдиним аргументом цього методу. В нашому прикладі в класі `CVector` треба оголосити метод з іменем **operator+**:

```
class CVector {
public:
  CVector operator+( CVector );
  //...
};
```

Тоді в рядку

```
z = u + v;
```

компілятор перетворить операцію додавання на виклик цього методу для об'єкту `u`, передавши об'єкт `v` в якості аргументу. Іншими словами, програма спрацює так, ніби в тексті було б записано

```
z = u.operator+( v );
```

Транслятор мови Сі++ звичайно ж, не слідкує за тим, що насправді робить створена програмістом перевантажена операція. Наприклад, в тілі функції **operator+** можна записати алгоритм віднімання чи множення — за зміст перевантажених операцій відповідіє програміст, тобто лише від людини залежить, що насправді робить з об'єктами функція, що перевантажує той чи інший знак операції.

Транслюючи вирази, в яких використовуються перевантажені операції, транслятор не намагається «бути розумнішим за програміста», тобто не перетворює ці вирази згідно правил арифметики і логіки, а виконує всі дії в точності так, як записано в тексті програми. Наприклад, якщо `u` та `v` — об'єкти класу `CVector`, то у виразі

```
v = (u-u)+v;
```

компілятор не намагатиметься (як зробила б людина) помітити, що вираз в дужках дає нульовий вектор, а додавання нульового вектора до вектора `v` не змінює останній. Натомість компілятор обчислить весь вираз: викличе функцію **operator-**, в обидва аргументи підставивши об'єкт `u`, а потім до отриманого об'єкту і об'єкту `v` викличе функцію **operator+**.

Мова Сі++ дозволяє перевантажувати багато різних операцій. Їх можна розбити на кілька груп, об'єднавши в групу операції, що мають схожий сенс та спільні особливості реалізації, див. табл. 3.1. Далі розглянемо окремо кожну групу.

Прокоментуємо важливу відмінність між призначенням і способом застосування операцій, відмічених в таблиці цифрами 1 та 2. Розглянемо оператори

```
c = a+b;
a += b;
```

В першому випадку операція `+` бере два значення операндів і вираховує та повертає третє. Ця операція ніяк не впливає на свої операнди. Натомість в другому випадку операція вносить нове значення у змінну `a`, яка є першим операндом. Крім того, значенням виразу `a+=b` є посилання на оновлену змінну `a`, що дозволяє будувати складні вирази вигляду

```
((a += b) -= c) *= d;
```

Табл. 3.1. Операції, які можна перевантажувати

Група	Операції	Прим.
Унарні	`+`, `-` (у виразах `+x`, `-x`)	1
Бінарні	`+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `&&`, `|`, `||`	1
Порівняння	`==`, `!=`, `<`, `>`, `<=`, `>=`	
Присвоювання	`=`	2, 3
Комбіновані	`+=`, `-= *=`, `/=`, `%=`, `&=`, `|=`	2
Префіксні інкремент та декремент	`--`, `++` (у виразах `--x` та `++x`)	2
Постфіксні інкремент та декремент	`--`, `++` (у виразах `x--` та `x++`)	
Індексування	`[]` у виразах `m[i]`	3
Функціональне застосування	`()` у виразах `f(x)`	3
Непряме звертання до члену	`->` у виразах `a->x`	3
Розіменування	`*` у виразах `*x`	3
Виділення і звільнення пам'яті	**`new`**, **`delete`**	3
Перетворення типу	`тип` у виразах `тип(вираз)`	

1 Як правило, створює новий об'єкт-результат і повертає його копію

2 Зазвичай змінює перший операнд та повертає посилання на нього

3 Перевантажується лише методом класу

Таким чином, бінарні та унарні арифметичні операції мають повертати об'єкт по значенню, а операції присвоювання та комбіновані — по посиланню.

Передача об'єкта до функції (в тому числі і до функції, яка перевантажує операцію) в якості аргумента по значенню пов'язана з зайвими затратами пам'яті та часу: для цього, як вже пояснювалося вище, створюється новий об'єкт — тимчасова копія, яка знищується при виході з функції. Набагато економніше передавати об'єкти-аргументи по посиланню. Аби при цьому гарантувати, що функція не зіпсує об'єкт (наприклад, операція + в звичайному розумінні не повинна змінювати свої операнди), варто аргументи оголошувати *константними* посиланнями. Якщо ж перевантажена операція по самому своєму призначенню має змінювати свій операнд (як, наприклад, операція += має занести результат додавання до об'єкту, який є її першим операндом), то цей операнд, зрозуміло, повинен передаватися через неконстентне посилання. Таким чином, для класу «вектор», наприклад, маємо такі прототипи операцій (якщо перевантажувати їх звичайними функціями, не методами класу):

```
CVector operator+( const CVector&, const CVector& );
Cvector& operator+=( CVector&, const CVector& );
```

3.2. Унарні операції

Унарних операцій усього дві: плюс та мінус. Їх є сенс перевантажувати тоді, коли клас чи структурний тип змістовно моделює деякий різновид математичних величин, для яких можна говорити про «протилежні» величини. Наприклад, для кожного вектору існує протилежний вектор. Результатом операції «плюс» має стати просто копія об'єкта-операнда (справді, якщо x — величина, то $+x$ — та ж сама величина).

Покажемо перевантаження цих операцій на двох прикладах. Спершу розглянемо структурний тип, що моделює комплексне число і проілюструємо, як унарні операції перевантажувати звичайними функціями (не членами класу). З математичної точки зору комплексне число можна зобразити парою дійсних чисел $\langle a, b\rangle$, де a називають *дійсною частиною*, а b — *уявною частиною*. Ми навмисно

користуємося тут позначенням вигляду $\langle a, b\rangle$ замість більш звичного $a + bi$, щоб підкреслити: сама по собі пара дійсних чисел однозначно репрезентує комплексне число (уявна одиниця i тому і уявна, що її цілком можна тримати в уяві, не виписуючи на папері).

```
struct TComplex {
  double re; // дійсна
  double im; // уявна
};
```

Якщо унарна операція перевантажується функцією — не членом класу, то ця функція повинна мати один аргумент, той операнд, до якого вона застосовується. Якщо z — комплексне число, то $+z$ — те ж саме число. З точки зору програмування унарна операція + просто створює копію свого операнда:

```
TComplex operator+( const TComplex &x ) {
  return x;
}
```

В наших позначеннях комплексне число, протилежне до даного, дається формулою

$$-\langle a, b\rangle = \langle -a, -b\rangle\,,$$

тобто щоб отримати комплексне число, протилежне до даного, треба змінити знак і у дійсної, і в уявної частини. Таким чином, в програмній реалізації маємо

```
TComplex operator-( const TComplex &x ) {
  TComplex y;
  y.re = - x.re;
  y.im = - y.im;
  return y;
}
```

Тепер розглянемо перевантаження унарних операцій методами класу на прикладі класу «вектор». Цей клас буде використовуватися і в подальших розділах, щоб демонструвати на ньому перевантаження інших операцій. Об'єкт цього класу має в собі член-змінну цілого типу `m_size`, в якій зберігається розмірність вектору, та покажчик `m_p` на динамічний масив, в елементах якого зберігаються

компоненти вектора. Конструктор з одним цілочисельним аргументом створює вектор заданої розмірності: передане через аргумент значення розмірності присвоює члену `m_size` та виділяє пам'ять під масив компонентів і заносить покажчик на нього в член `m_p`. Для реалізації перевантаженої операції «мінус» необхідний конструктор копіювання.

Звичайно ж, у класу має бути й деструктор, який знищує динамічний масив, на який вказує покажчик `m_p`. Щоб не засмічувати розбір прикладу зайвими подробицями, приймемо такі спрощення: в оголошенні класу перераховувати лише ті методи, які суттєві саме для даного прикладу; тіла конструктору та деструктору не наводимо — читач, який добре засвоїв матеріал розділів 1.7 та 1.8, легко напише їх самостійно. Нижче показано фрагмент оголошення класу «вектор»: конструктор, прототип методу, що перевантажує унарну операцію «мінус», та члени-змінні.

```
class CVector {
public:
  CVector(int);
  CVector(const CVector&);
  // ...
  CVector operator-( void ) const;
protected:
  int m_size;
  double *m_p;
};
```

Оскільки унарна операція над об'єктом класу перевантажується методом цього класу, метод не має аргументів: об'єкт-операнд стає тип об'єктом, для якого викликається метод.

Реалізація методу, що перевантажує операцію «мінус», полягає в тому, щоб створити новий об'єкт-вектор `r` тієї ж розмірності (див. аргумент, що передається до конструктора), а потім один за одним присвоїти значення компонентам нового вектора: це мають бути компоненти даного вектора (тобто того, для якого викликано метод), взяті з протилежним знаком (див. оператор присвоювання, що виконується у циклі):

```
CVector CVector::operator-( void ) const {
  CVector r( m_size );
```

```
  for( int i = 0; i < m_size; ++i )
    r.m_p[i] = - m_p[i];
  return r;
}
```

Треба звернути увагу на те, що відбувається в останньому операторі. Об'єкт `r` є локальним для даного методу. Коли оператор **return** повертає його значення, створюється тимчасовий об'єкт-копія (для цього викликається конструктор копіювання), а сам об'єкт `r` знищується з викликом деструктора, який видаляє пов'язаний з цим об'єктом масив елементів. Тому ще раз наголосимо на вирішальній ролі конструктора копіювання.

3.3. Бінарні операції

Знов візьмемо в якості прикладу структурний тип, що моделює комплексні числа. Комплексне число $a+bi$ однозначно визначається парою $\langle a, b\rangle$ та програмно моделюється об'єктом структурного типу

```
struct TComplex {
  double re;
  double im;
};
```

Суму двох комплексних чисел задає формула

$$\langle a_1, b_1\rangle + \langle a_2, b_2\rangle = \langle a_1 + a_2, b_1 + b_2\rangle\,,$$

тобто дійсна та уявна частини додаються окремо одна від одної. Прямим перекладом цієї формули на мову програмування є наступний код. У функції створюється новий об'єкт `z`, його дійсній частині присвоюється значення суми дійсних частин доданків, так само і для уявної частини. Обчислений таким чином об'єкт `z` функція повертає.

```
TComplex operator+(
  const TComplex &x,
  const TComplex &y
) {
  TComplex z;
  z.re = x.re + y.re;
```

```
  z.im = x.im + y.im;
  return z;
}
```

Наступний приклад показує, що операція додавання має сенс тоді, коли один з двох доданків дійсне число, а другий — комплексне. З точки зору математики дійсне число додається до дійсної частини комплексного числа:

$$\langle a, b\rangle + x = \langle a + x, b\rangle .$$

Оскільки, як сказано вище, компілятор нічого сам не домислює за програміста, нам треба описати обидві такі операції:

```
TComplex operator+( const TComplex &x, double y ) {
  TComplex z;
  z.re = x.re + y;
  z.im = x.im;
  return z;
}

TComplex operator+( double x, const TComplex &y ) {
  TComplex z;
  z.re = x + y.re;
  z.im = y.im;
  return z;
}
```

Формула множення двох комплексних чисел в наших позначеннях виглядає так:

$$\langle a_1, b_1\rangle \cdot \langle a_2, b_2\rangle = \langle a_1 a_2 - b_1 b_2, a_1 b_2 + a_2 b_1\rangle .$$

Відповідна програмна реалізація також створює новий структурний об'єкт, заносить в його члени `re` та `im` відповідні значення і повертає об'єкт:

```
TComplex operator*(
  const TComplex &x,
  const TComplex &y
) {
  TComplex z;
```

```
  z.re = x.re * y.re - x.im * y.im;
  z.im = x.re * y.im + x.im * y.re;
  return z;
}
```

Насамкінець дамо без детальних пояснень функції, що перевантажують операцію множення комплексного числа на дійсне і дійсного на комплексне згідно формули

$$\langle a, b\rangle \cdot y = \langle a \cdot y, b \cdot y\rangle .$$

```
TComplex operator*( const TComplex &x, double y ) {
  TComplex z;
  z.re = x.re * y;
  z.im = x.im * y;
  return z;
}

TComplex operator*( double x, const TComplex &y ) {
  TComplex z;
  z.re = x * y.re;
  z.im = x * y.im;
  return z;
}
```

Перейдемо тепер до перевантаження бінарних операцій методами класу на прикладі векторів. Перший з двох операндів бінарної операції є тим об'єктом, для якого метод викликається. Другий операнд виступає аргументом методу. Нижче показано фрагмент оголошення класу «вектор» з прототипами методів, що перевантажують додавання двох векторів, множення вектору на число та скалярне множення вектору на вектор. Тут показано також прототипи двох конструкторів, які неодмінно потрібні в реалізаціях перевантажених арифметичних операцій.

```
class CVector {
public:
  CVector(int);
  CVector(const CVector&);
  // ...
  CVector operator+( const CVector& ) const;
```

```
  CVector operator*( double ) const;
  double  operator*( const CVector& ) const;
protected:
  int m_size;
  double *m_p;
};
```

Нехай $\vec{u}$ та $\vec{v}$ — вектори розмірності n. Тоді їх сумою є такий вектор $\vec{w} = \vec{u} + \vec{v}$, що $w_i = u_i + v_i$ для всіх допустимих значень i. Нижче показано реалізацію методу додавання векторів.

```
CVector CVector::operator+( const CVector &x ) const {
  if( x.m_size != m_size ) {
    cerr << "Розмірності␣різні" << endl;
    return CVector(0);
  }
  CVector r( m_size );
  for( int i = 0; i < m_size; ++i )
    r.m_p[i] = m_p[i] + x.m_p[i];
  return r;
}
```

Сума двох векторів має сенс лише тоді, коли ці вектори мають однакову розмірність. Метод перевіряє цю умову і, якщо вона порушується, виводить повідомлення та повертає вектор нульової розмірності (який не має компонентів). Якщо ж перевірка пройшла успішно, створюється новий вектор `r` тієї ж розмірності, що й вектори-доданки. Цикл пробігає по всіх значеннях індекса, в тілі циклу `i`-му компоненту вектора `r` присвоюється значення суми `i`-го компонента вектора першого вектора-доданка (об'єкт, для якого викликано метод) та `i`-го компонента другого доданка (переданого через аргумент).

Наприкінці метод повертає створений об'єкт `r` по значенню. Це означає, що створюється тимчасова копія цього об'єкта (для чого викликається конструктор копіювання), а сам об'єкт `r` знищується.

У щойно розглянутій операції додавання обидва операнди мали тип «вектор». Наступна операція має перший операнд типу вектору (це той об'єкт, для якого метод викликається), а другий типу дійсного числа (передається до метоуд в якості аргумнету). Якщо $\vec{u}$ — вектор, а k — число, то добуток $\vec{v} = \vec{u} \cdot k$ — це такий вектор, кожен

компонент якого визначається за формулою $v_i = u_i k$ (для всіх допустимих значень i).

```
CVector CVector::operator*( double k ) const {
  CVector r( m_size );
  for( int i = 0; i < m_size; ++i )
    r.m_p[i] = m_p[i] * k;
  return r;
}
```

Цей фрагмент коду зрозуміти неважко. Створюється локальний об'єкт `r` — вектор тієї ж розмірності, що й вектор, до якого застосовано множення. Далі в циклі кожному його компоненту присвоюється значення згідно наведеної вище формули. Насамкінець метод повертає тимчасову копію об'єкта `r` (для створення якої викликаєтсья конструктор копіювання), а сам об'єкт `r` знищується.

Нарешті розберемо метод, який програмно моделює скалярне множення двох векторів. Скалярний добуток має сенс лише тоді, коли вектори-множники $\vec{u}$ та $\vec{v}$ мають однакову розмірність, нехай вона дорівнює n. Скалярний добуток це число, яке визначається за формулою

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$$

(варто нагадати, що в математиці прийнято нумерувати компоненти вектора з 1 до n, тоді як в наступному програмному коді нумерація іде в стилі мови Сі++, від 0 до $n-1$). Цій формулі відповідає програмна реалізація

```
double CVector::operator*( const CVector &x ) const {
  double s = 0;
  if( x.m_size != m_size ) {
    cerr << "Розмірності різні" << endl;
    return s;
  }
  for( int i = 0; i < m_size; ++i )
    s += m_p[i] * x.m_p[i];
  return s;
}
```

Тут змінна `s` — накопичувач суми. Якщо розмірності векторів-множників виявляються різними, метод друкує повідомлення та повертає значення 0. Якщо ж перевірка розмірностей пройшла, у циклі накопичується сума попарних добутків `i`-х компонентів векторів. Отриману суму метод повертає.

3.4. Операції порівняння

Якщо об'єкти, скажімо `x` та `y`, моделюють деякі математичні величини, то вираз `x==y` з перевантаженою операцією порівняння `==` повинен давати значення «істина» (в традиціях мови Сі — відмінне від 0 число) тоді, коли ці об'єкти представляють одну й ту саму величину, в усіх інших випадках значенням виразу повинна бути «хиба» (число 0 згідно правил мови Сі).

Дуже показовий приклад дають структурні об'єкти, що моделюють поняття раціонального дробу. Зазвичай раціональний дріб позначають через $\frac{m}{n}$, де m — ціле число, а n — додатне ціле. Потрібно підкреслити: раціональний дріб це не операція ділення, застовована до двох значень, а просто пара цілих чисел, одне з яких називають чисельником, а друге — знаменником. Програмною моделлю раціонального дробу є об'єкт наступного структурного типу, який містить в собі чисельник та знаменник:

```
struct TRational {
  int m; // чисельник
  int n; // знаменник
};
```

Як відомо з математики, щоб порівняти два дроби, потрібно привести їх до спільного знаменника, домноживши чисельник і знаменник кожного дробу на знаменник іншого, та порівняти отримані чисельники:

$$\frac{m_1}{n_1} = \frac{m_2}{n_2} \quad \Leftrightarrow \quad m_1 n_2 = m_2 n_1.$$

Програмною реалізацією цієї формули є наступна функція:

```
int operator==(
  const TRational& x,
  const TRational &y
) {
```

```
  return (x.m * y.n == y.m * x.n );
}
```

Цей приклад добре ілюструє важливу особливість: два об'єкти можуть мати зовсім різні значення членів, але з точки зору математики моделювати одну й ту саму величину. Наприклад, візьмемо дроби $\frac{2}{4}$ та $\frac{3}{6}$: чисельник першого дробу не рівний чисельнику другого, так само і знаменники у цих дробів різні. Але обидва вони після скорочення дають $\frac{1}{2}$ і отже рівні між собою. Проілюструємо роботу операції порівняння наступним програмним кодом:

```
int main( void ) {
  TRational u = {2, 4}, v = {3, 6};
  if( u == v )
    cout << "однакові"
  else
    cout << "різні";
  cout << endl;
  return 0;
}
```

Знову підкреслимо, що компілятор не намагається ≪додумати≫ програму замість програміста. Операцію порівняння на нерівність `!=` теж треба перевантажити власноруч: компілятор не покладається на те, що можна було б взяти наявну операцію `==` та піддати її запереченню.

Розглянемо перевантаження операції порівняння методом класу на прикладі векторів. Оголошення класу, до якого додано прототип методу порівняння на рівність, показано нижче:

```
class CVector {
public:
  // ...
  int operator==( const CVector& ) const;
protected:
  int m_size;
  double *m_p;
};
```

Будь-який вектор рівний самому собі: якщо відомо, що зліва та справа від операції порівняння стоїть один і той самий вектор, то

можна одразу повернути значення «істина», не витрачаючи часу на інші операції. Два вектори, $\vec{u}$ та $\vec{v}$, завідомо не можуть бути рівними між собою, якщо в них різна розмірність. Якщо ж розмірність у них однакова, треба одну за одною перевірити рівності $u_1 = v_1, \ldots, u_n = v_n$. Якщо хоча б одна з них не справджується, вектори не рівні. Якщо перевірено всі такі рівності, і жодна не виявилася хибною, вектори рівні. Прямим перекладом цих міркувань на мову програмування є наступний код:

```
int CVector::operator==( const CVector& x ) const {
  if( this == &x ) return 1;
  if( x.m_size != m_size ) return 0;
  for( int i = 0; i < m_size; ++i )
    if( x.m_p[i] != m_p[i] ) return 0;
  return 1;
}
```

3.5. Операція присвоювання

Операцію присвоювання можна перевантажувати лише одним способом — методом класу. Цей метод викликається для об'єкту з лівої частини присвоювання (того, об'єкту, якому присвоюється нове значення), а його аргументом стає об'єкт з правої частини (той, значення якого присвоюється).

Варто зазначити, що присвоювання — це та операція, яку (на відміну від інших) транслятор взмозі здійснити сам, навіть якщо не перевантажувати її власними руками. Для прикладу візьмемо вже відомий структурний тип «комплексне число». Хоча перевантажена операція = для цього типу відсутня, оператор

```
TComplex x = {1, -1}, y;
y = x;
```

має сенс: в таких випадках транслятор самостійно копіює весь вміст об'єкта з правої частини присвоювання в об'єкт, що стоїть зліва. Після присвоювання члени-змінні об'єкту з лівої сторони мають такі ж значення, що й відповідні члени-змінні об'єкту з правої сторони.

Такого способу присвоювання цілком досить для таких типів, як комплексні числа, раціональні дроби та інших, у яких є лише члени-

змінні простих типів. Потреба самостійно визначати алгоритм присвоювання об'єктів виникає тоді, коли об'єкти мають члени-змінні типу покажчиків, особливо коли це покажчики на динамічні масиви, як у класу вектор. Причини, що зумовлюють потребу в перевантаженій операції присвоювання ті ж самі, що й для конструкторів копіювання (розділ 1.9). Якщо дозволити присвоювання самих лише значень змінних-членів, то після присвоювання два об'єкти (з лівої та з правої частин оператора) містили б покажчик на одну й ту саму ділянку динамічної пам'яті, тоді як правильно було б виділити нову ділянку і скопіювати туди зміст першої.

Нижче показано оголошення класу «вектор» з прототипом методу присвоювання. Треба звернути увагу, що метод повертає посилання на об'єкт-вектор: значенням виразу `a=b` має стати посилання на об'єкт `a` з лівої частини присвоювання.

```
class CVector {
public:
  // ...
  CVector& operator=( const CVector& );
protected:
  int m_size;
  double *m_p;
};
```

Реалізація цього методу має кілька тонких особливостей, на які варто звернути увагу. По-перше, треба передбачити захист від самоприсвоювання — від операцій вигляду `a=a`, коли в правій в лівій частинах оператору стоїть один і той самий об'єкт. Цілком очевидно, що в цьому випадку жодних операцій виконувати не потрібно, достатньо повернути посилання на об'єкт. Далі треба передбачити випадок, коли до присвоювання об'єкт-вектор з лівої частини операції має іншу розмірність, ніж вектор з правої частини. В цьому випадку треба знищити масив елементів, що був в об'єкті раніше, виділити пам'ять під новий масив потрібного розміру та оновити значеня члену `m_size`. Нарешті виконується основна частина алгоритму — значення компонентів з об'єкта-аргумента копіюються до поточного об'єкта один за одним, в циклі. Метод повертає посилання на об'єкт, для якого його було викликано. Відповідну програмну реалізацію показано нижче:

```
CVector& CVector::operator=( const CVector& x ) {
  if( &x == this ) return *this;
  if( x.m_size != m_size ) {
    delete[] m_p;
    m_p = new double[x.m_size];
    m_size = x.m_size;
  }
  for( int i = 0; i < m_size; ++i )
    m_p[i] = x.m_p[i];
  return *this;
}
```

3.6. Комбіновані операції присвоювання

На відміну від бінарних арифметичних операцій, які здійснюють дію над величинами, що зберігаються в двох об'єктах, та записують результат дії до третього об'єкту, комбіновані операції здійснюють дію та записують її результат до першого з двох своїх операндів. Зручно, коли комбінована операція повертає посилання на цей перший операнд, тобто коли результатом операції стає посилання на об'єкт, до якого внесено нове значення. Це дозволяє поєднувати операції в ланцюжки, тобто до результату однієї операції застосовувати другу і т.д., наприклад:

```
((a += b) *= c) -= d;
```

Почнемо з прикладу перевантаження таких операцій звичайними функціями. Треба звернути увагу, що в наступній функції другий аргумент передається через константне посилання, а перший — через неконстантне. Справді, операція додає другий аргумент до першого, причому значення другого аргументу вона змінювати не повинна.

```
TComplex& operator+=(
  TComplex &x,
  const TComplex &y
) {
  x.re += y.re;
  x.im += y.im;
  return x;
}
```

Наступна функція домножує дане комплексне число на дійсне число:

```
TComplex& operator*( TComplex &x, double y ) {
  x.re *= y;
  x.im *= y;
  return x;
}
```

Наступні операції (перевантажені методами класу) додають вектор до вектора та домножують вектор на число і коментарів, ймовірно, не потребують:

```
class CVector {
public:
  // ...
  CVector& operator+=( const CVector& );
  CVector& operator*=( double );
protected:
  int m_size;
  double *m_p;
};

CVector& CVector::operator+=( const CVector& x ) {
  if( x.m_size != m_size ) {
    cerr << "Розмірності різні" << endl;
    return *this;
  }
  for( int i = 0; i < m_size; ++i )
    m_p[i] += x.m_p[i];
  return *this;
}

CVector& CVector::operator+=( double k ) {
  for( int i = 0; i < m_size; ++i )
    m_p[i] *= k;
  return *this;
}
```

3.7. Операції інкременту та декременту

Перевантажувати для об'єктів свого класу операції `++` та `--` слід тоді, коли за прикладним змістом задачі для цих об'єктів мають сенс поняття «наступне значення» та «попереднє значення». Нагадаємо: операція інкременту застосовується до певного об'єкту, який зберігає в собі деяке поточне значення x, і примушує об'єкт прийняти нове значення — наступне після x. Так само, операція декременту змінює значення об'єкту на те, яке передує x.

Операції інкременту і декременту в складі виразів виступають у двох формах: префіксній і постфіксній. Префіксна операція (знак операції пишеться перед іменем змінної: `--x`) означає, що при обчисленні виразу треба використати вже змінене (збільшене чи зменшене) значення об'єкту, а постфіксна (пишеться після імені змінної `x++`) означає, що значення об'єкту змінюється, але при обчисленні виразу враховуєьться старе, незмінене значення. Операції інкременту та декременту (як префіксні, так і постфіксні) можна перевантажувати обома способами: звичайними функціями та методами класу.

Спочатку розглянемо, як перевантажувати префіксні операції звичайними функціями. Єдиним аргументом такої функції повинно бути посилання на змінну-об'єкт, до якого застосовується операція. Далі, значенням, що повертає префіксна операція, має бути оновлений об'єкт: наприклад, у виразі `(++x)*y` першим операндом операції множення є змінна `x` *після* інкременту. Тому функція **`operator++`** чи **`operator--`** повертає посилання на той самий об'єкт, який отримала в якості аргумента та якому змінила значення.

Проілюструємо це на прикладі раціональних дробів. Нагадаємо означення структурного типу, що моделює дроби:

```
struct TRational {
  int m; // чисельник
  int n; // знаменник
};
```

Для того, щоб до дробу додати 1, треба чисельник збільшити на величину знаменника:

$$\frac{m}{n} + 1 = \frac{m}{n} + \frac{n}{n} = \frac{m+n}{n}.$$

Таким чином, маємо функцію, що перевантажує операцію префіксного інкременту:

```
CRational& operator++( CRational &x ) {
  x.m += x.n;
  return x;
}
```

Дещо складніше справи з перевантаженням постфіксної операції. По-перше, постфіксна операція пишеться так само, як і префіксна, отже функція, що її перевантажує, має те ж саме ім'я **operator++**. Щоб відрізнити цю функцію від функції, що перевантажує префіксну операцію, автори мови Сі++ знайшли такий штучний прийом: функція, що перевантажує постфіксну операцію інкременту чи декременту, має фіктивний другий аргумент типу **int**. Цей аргумнет при обчисленнях не використовується, ніяке осмислене значення через нього не передається — він потрібен лише для того, щоб компілятор міг розрізнити між двома функціями з однаковими іменами.

По-друге, якщо префіксна операція вносить зміни в об'єкт-операнд та повертає посилання на нього (і її результатом стає об'єкт *після* здійсненої над ним операції), то постфіксна операція повинна змінити об'єкт, але повернути його старе, незмінене операцією значення. Тому постфіксна операція має запам'ятати копію об'єкта-операнда, потім здійснити свої дії над об'єктом-операндом і повернути збережену копію. Звідси слідує, що операція постфіксного інкременту чи декременту має повертати не посилання на об'єкт, а значення. У підсумку маємо таку реалізацію постфіксного інкременту для дробів:

```
CRational operator++( CRational &x, int ) {
  CRational t = (*this);
  x.m += x.n;
  return t;
}
```

Перевантаження операцій інкременту да декременту методами розглянемо на прикладі класу, що моделює дату. Дата складається з трьох цілочисельних параметрів: року, номера місяця і дня. Для будь-якої дати є сенс казати про наступну і попередню дату. Метод, що перевантажує операцію, викликаєтья для об'єкта-операнда. То-

му метод, що перевантажує префіксну операцію, не має аргументу, а метод для постфіксної має один фіктивний аргумент типу **int**, потрібний лише для того, щоб компілятор відрізнив його від префіксної операції. Як вже пояснювалося вище, префіксна операція повертає посилання на об'єкт, значення якого вона щойно змінила, а постфіксна — копію значення цього об'єкта до зміни. Нижче показано оголошення класу «дата» з прототипами методів інкременту да декременту.

```
class CDate {
public:
  CDate(int, int, int);
  viod print(void);
  CDate& operator++(void);
  CDate operator++(int);
  CDate& operator--(void);
  CDate operator--(int);
protected:
  int m_y;
  int m_m;
  int m_d;
};
```

Прокоментуємо алгоритм, який для заданої дати обчислює наступну дату. В найпростішому випадку він просто збільшує на 1 номер дня. Але якщо після цього номер дня стане більше за кількість днів у поточному місяці, треба день скинути до першого, а номер місяця збільшити на 1. Якщо ж при цьому номер місяця перевищить 12, треба скинути номер місяця в 1 та збільшити рік. Залишається єдина деталь: місяці мають різну кількість днів, причому в лютому кількість днів залежить від року. Допоміжна функція `lastDayInMonth` приймає два аргументи: номер місяця і рік і повертає кількість днів в даному місяці даного року. Вона бере кількість днів з масиву `days` (для лютого в ньому закладено значення 28) і додає поправку `r`. Поправка дорівнює 1, якщо місяць другий, а рік вісокосний, та 0 в усіх інших випадках. Оскільки в мові Сі нумерація елементів масиву починається з 0, а місяці нам зручніше нумерувати з 1, напочатку масиву `days` вставлено «зайвий» елемент.

```
// кількість днів у місяцях
```

```
int days[] = {
  -1, // 0-й елемент, щоб місяці нумерувати з 1-го
  31, 28, 31, 30,
  31, 30, 31, 31,
  30, 31, 30, 31
};

// повертає кількість днів в місяці m року y
int lastDayInMonth(int m, int y) {
  int r = 0; // поправка на лютий вісокосного року
  if( (m == 2) &&
    (y%4 == 0) &&
    ((y%100 != 0) | (y%400 == 0))
  )
    r = 1;
  return days[m_m] + r;
}
```

Маючи цю допоміжну функцію, можна реалізувати методи, що перевантажують префіксні операції інкременту да декременту над датами:

```
CDate& CDate::operator++(void) {
  ++m_d; // день збільшити на 1
  // якщо перевищили кількість днів у місяці
  if( m_d > lastDayInMonth(m_m, m_y) ) {
    m_d = 1; // встановити 1-е число
    ++m_m;   // наступного місяця
    // якщо місяць наступний після грудня
    if( m_m > 12 ) {
      m_m = 1; // встановити січень
      ++ m_y;  // наступного року
    }
  }
  return *this;
}

CDate& CDate::operator--(void) {
  --m_d; // попереднє число
  // якщо вийшли назад за 1-е число
  if( m_d < 1 ) {
```

```
    --m_m; // встановити попередній місяць
    // коли місяць попередній перед січнем
    if( m_m < 1 ) {
      m_m = 12; // встановити грудень
      -- m_y;   // попереднього року
    }
    // встановити останній день місяця
    m_d = lastDayInMonth(m_m, m_y);
  }
  return *this;
}
```

Залишається розібрати реалізацію постфіксних операцій. Як вже пояснювалося вище, треба зберегти в тимчасову змінну поточне значення дати, потім перемкнути поточний об'єкт на наступну чи попередню дату і повернути збережене попереднє значення. Щоб не переписувати повторно наведені вище громіздкі алгоритми, для викличемо для поточного об'єкту вже реалізовані префіксні операції:

```
CDate CDate::operator++(int) {
  CDate t = (*this); // копія поточної дати
  ++ (*this); // виклик префіксної операції
  return t; // повернути попереднє значення
}

CDate CDate::operator--(int) {
  CDate t = (*this); // копія поточної дати
  -- (*this); // виклик префіксної операції
  return t; // повернути попереднє значення
}
```

3.8. Операція індексування та контейнери

Якщо для класу перевантажити операцію індексування `[]`, об'єкти цього класу зможуть поводити себе так, ніби вони є масивами; принаймні, робота з ними зовні виглядатиме саме так. Для ілюстрації якнайкраще підходить клас «вектор». В показаному нижче лістингу превантажена операція індексування приймає один аргумент — номер (індекс) компонента вектора та повертає посилання на комірку, в якій відповідне значення зберігається. Крім того, метод **operator[]**

робить дуже важливу перевірку: чи допустиме значення індексу. В разі, коли програма намагається звернутися до неіснуючуго елементу (індекс менший за 0 або більший чи рівний кількості елементів), метод виводить відповідне повідомлення та звертається до першого елементу вектора (за індексом 0).

```
class CVector {
public:
  // ...
  double& operator[]( int ) const;
protected:
  int m_size;
  double *m_p;
};

double& CVector::operator[]( int i ) const {
  if( (i < 0) || (i >= m_size) ) {
    cerr << "Вихід за границі" << endl;
    i = 0;
  }
  return m_p[i];
}
```

Нехай `u` — об'єкт класу `CVector`. Наявність перевантаженої операції індексування робить можливими оператори вигляду

```
// взяти з вектора значення компонента
double x = u[0];
// присвоїти компоненту вектора нове значення
u[1] = 0;
// взяти значення для операції виведення
cout << u[2];
// взяти посилання на компонент
// і використати в операції введення
cin >> u[3];
```

Як бачимо, реалізована вище операція індексування може використовуватися і в лівій, і в правій частині присвоювання, оскільки повертає посилання на елемент вектору, а не копію значення.

З можливістю перевантажувати для свого класу операцію індексування `[]` тісно пов'язане важливе і широко розповсюджене в су-

часному програмуванні поняття — *контейнер*. Контейнером називається об'єкт, який за своїм призначенням і сутністю є вмістилищем відносно великої кількості деяких відносно простих об'єктів — елементів, причому з контейнера можна вибирати певний елемент (за номером або деяким ключем).

Мова Сі сама по собі підтримує один-єдиний різновид контейнерів — масив. Справді, масив є єдиним цілим об'єктом, що зберігає в собі багато дрібних об'єктів-елементів; до будь-якого елементу масива можна звернутися за номером (індексом). Якщо `m` — ім'я масиву, а `i` — вираз цілого типу, то вираз `m[i]` дає посилання на певний елемент масиву.

Контейнери — це природне продовження ідеї масиву. Мова Сі++ дозволяє програмісту власноруч створювати контейнерні класи, у яких перевантажено операцію індексування і об'єкти яких поводять себе подібно до масивів. Якщо `a` — об'єкт контейнерного класу, `s` — вираз, значенням якого є індекс або ключ для пошуку елементу в контейнері, то обчислення виразу `a[s]` означає виклик для об'єкта `a` методу **`operator`**`[]` з аргументом `s`, а значенням, яке повертає метод, має стати посилання на елемент контейнера, знайдений за даним індексом (ключем).

У мові Сі для звертання до елементів масиву можуть в якості індексу використовуватися лише цілі числа. Мова Сі++ дозволяє створювати контейнери з будь-яким типом індексу: аргумент перевантаженої операції `[]` може бути текстовим рядком, дійсним числом, навіть об'єктом деякого класу (останній варіант широко використовується в професійно написаних бібліотеках класів).

Розглянемо дуже простий приклад. Нижче показано клас, який моделює телефонну книжку. Телефонна книжка складається з певної кількості записів, а кожен запис містить ім'я та номер телефону. Таким чином, телефонна книжка — це контейнер, елементами якого є окремі записи. Основна операція, заради якої, власне, і заводять телефонні книги, — це пошук номера абонента за відомим іменем. Цю операцію було б логічно оформити як перевантажене індексування. Причому в ролі індекса виступає не номер запису (як в масивах), а ім'я абонента, тобто текстовий рядок. Перевантажена операція `[]` з аргументом типу **`char*`** дозволить використовувати в програмі зручну і коротку форму запису на зразок `b["Оксана"]`, де

`b` — контейнерний об'єкт (телефонна книжка).

```
struct TPhoneRec {
  char name[20];
  char phone[20];
};

class CPhoneBook {
public:
  CPhoneBook(int);
  ~CPhoneBook();
  void addPhone( const char*, const char* );
  const char *operator[]( const char* );
protected:
  TPhoneRec *m_book;
  int m_size;
  int m_records;
};

CPhoneBook::CPhoneBook(int size_) :
  m_book( new TPhoneRec[size_] ),
  m_size( size_ ),
  m_records( 0 ),
{
}

CPhoneBook::~CPhoneBook() {
  delete[] m_book;
}

void CPhoneBook::addPhone(
  const char *newName,
  const char *newPhone
) {
  if( m_records == m_size ) {
    cerr << "Записна␣книжка␣заповнена" << cout;
    return;
  }
  strcpy( m_book[m_records].name,  newName );
  strcpy( m_book[m_records].phone, newPhone );
  ++ m_records;
```

```
}

const char *CPhoneBook::operator[](
  const char* theName
) {
  static char notFound[] = "не знайдено";
  for( int i = 0; i < m_records; ++i )
    if( strcpy( m_book[i].name, theName ) == 0 )
      return m_book[i].phone;
  return notFound;
}
```

Коротко прокоментуємо основні ідеї реалізації. Об'єкт класу `CPhoneBook` містить покажчик `m_book` на динамічний масив записів і розмір цього масиву `m_size`. В реальних паперових записниках не всі сторінки мають бути заповненими — в загальному випадку записник містить вільне місце, куди можна дописувати нові телефони. В даній програмній реалізації вважаємо, що перші `m_records` елементів масиву містять занесені користувачем дані, а решта елеменів перебуває в резерві.

В конструктор передається один аргумент — місткість телефонної книжки. Констурктор створює масив заданої довжини, присвоює значення цієї довжини змінній-члену `m_size` та ініціалізує член `m_records` нулем: щойно створений записник містить 0 записів.

Метод `addPhone` отримує два аргументи типу текстових рядків: ім'я абонента та номер телефону, які треба занести до книжки. Метод спочатку перевіряє, чи не заповнена книжка до кінця. Якщо вільне місце в ній є, то дані заносяться в перший незайнятий запис і лічильник зайнятих записів збільшується на 1.

Нарешті, метод **`operator[]`** передивляється в циклі всі зайняті записи і для кожного з них порівнює ім'я абонента з ключем, переданим через аргумент. Якщо у якомусь записі ім'я абонента збігається з ключем, метод завершує роботу і порвертає (у вигляді покажчика на рядок) номер телефону з цього ж запису. Якщо всі записи перевірено і в жодному з них потрібного імені не виявилося, метод повертає покажчик на рядок «не знайдено».

Наступний фрагмент коду ілюструє роботу з об'єктом нашого класу. З останніх рядків видно, що з точки зору користувача об'єкт-

контейнер поводить себе майже як масив, індексами якого замість цілих чисел є текстові рядки.

```
int main( void ) {
  CPhoneBook b(10);
  b.addPhone( "Олена",     "8-067-111-22-33" );
  b.addPhone( "Світлана", "8-050-123-45-67" );
  b.addPhone( "Оксана",    "8-068-987-65-43" );
  b.addPhone( "Леся",      "8-093-444-55-66" );
  cout << b[ "Оксана" ] << endl;
  cout << b[ "Марія" ]  << endl;
  return 0;
}
```

Наостанок треба зазначити, що контейнери — це надзвичайно важлива для високопрофесійного програмування ідіома[1], і її вичерпний розгляд неможливий при першому оглядовому знайомстві з ООП та мовою Сі++. Є стандартизована бібліотека шаблонів-контейнерів STL; надзвичайно популярна бібліотека класів Qt фірми Trolltech також містить потужний арсенал контейнерів. З контейнерами тісно пов'язана ідіома ітераторів. Студенти, які бажають самостійно поглибити свої знання, можуть звернутися до книги [9].

3.9. Короткий огляд теоретичних основ

- Перевантаженням знаків операцій називається явище, коли один знак операції (плюс, мінус тощо) означає різні дії в залежності від типу операндів, до яких його застосовано.
- Мова Сі++ дозволяє програмісту самостійно перевантажувати майже всі наявні в мові операції — довизначати їх для своїх власних типів даних (зокрема, для об'єктів класів).
- Щоб перевантажити операцію, треба визначити функцію, ім'я якої складається зі спеціального слова **operator** та знаку операції.

[1] Нагадаємо: під ідіомою в програмуванні розуміють певний спосіб використання мовних засобів для вирішення достатньо розповсюдженого різновиду реальних задач.

- Операцію в загальному випадку можна перевантажувати як звичайною функцією, так і методом класу. Деякі операції (зокрема, операцію присвоювання `=` чи операцію індексування `[]`) можна перевантажувати лише методом класу.

- Об'єкти-операнди рекомендується передавати по посиланню, а коли можливо — по константному посиланню: це економить пам'ять та час, які інакше витрачалися б на створення тимчасової копії об'єкта.

- Унарна (з одним операндом) операція перевантажується або функцією з одним аргументом (до аргументу передається операнд), або методом класу — такий метод не має аргументів, операнд є тим об'єктом, для якого викликається метод.

- Якщо бінарна операція (тобто операція з двома операндами, як `+`, `*` тощо) перевантажуєтсья звичайною функцією, то ця функція повинна мати два аргументи. Операнд, що стоїть зліва від знаку операції, передається у перший аргумент, а операнд справа від знаку — у другий аргумент.

- Якщо бінарна операція перевантажується методом класу, то метод повинен мати один аргумент. Операнд зліва від знаку операції є тим об'єктом, для якого викликається метод, а правий операнд передається до методу в якості аргумента.

- Операції порівняння порівнюють математичні величини, що їх зображують два об'єкти, та повертають логічне значення (істина чи хиба).

- Операція присвоювання застосовується до двух об'єктів і першому з них присвоює те значення, яке зберігається в другому. Повертає зазвичай посилання на той об'єкт, якому щойно присвоїла нове значення.

- Якщо для того чи іншого типу даних не описувати власноруч перевантажену операцію присвоювання, компілятор взмозі і сам обробити оператори вигляду `x=y` для цього типу. В цьому випадку весь вміст об'єкта `y` з точністю до байта копіюється до об'єкта `x`. Але такий спосіб присвоювання об'єктів не годиться, коли об'єкти серед своїх даних-членів мають покажчики:

після присвоювання вийшло б, що два об'єкти мають покажчики на одну й ту саму область даних. В таких випадках задача перевантаженої операції = — забезпечити, щоб замість копії покажчика на ті ж дані створити копію цих даних за новою адресою.

- Комбіновані операції присвоювання здійснюють математичну дію над двома об'єктами, але, на відміну від бінарних операцій, поміщують результат до свого першого операнду замість того, щоб повертати копію його значення. Повертають, як правило, посилання на перший операнд, значення якого щойно змінили.
- Операції інкременту да декременту є сенс перевантажувати тоді, коли для математичних величин, що їх моделюють об'єкти нашого типу даних, можна говорити про наступне та попереднє значення. Ці операції існують в префіксній та постфіксній формах. Як правило, префіксна операція змінює об'єкт-операнд та повертає посилання на нього. Постфіксна операція запам'ятовую копію поточного значення об'єкта, потім вносить в об'єкт зміну і повертає копію його старого значення. Щоб відрізнити постфіксну операцію від одноіменної префіксної, використовується фіктивний аргумент типу **int**.
- Операцію індексування, як правило, перевантажують для контейнерних класів. Контейнерний клас — це клас, об'єкти якого призначено для зберігання відносно великої кількості деяких відносно дрібних об'єктів-компонентів. Аргументом перевантаженої операції індексування є або індекс (номер компонента в контейнері), або ключ, за яким цей компонент можна в контейнері знайти. Робота з об'єктом-контейнером виглядає в тексті програми так, ніби він є масивом.

Розділ 4

Шаблони функцій та класів

Конструкція, яка вивчається в цьому розділі, не має такого ключового світоглядного значення, як класи чи об'єкти, а є просто засобом додаткової зручності та у багатьох практичних задачах гарно доповнює та підсилює потужність вивчених раніше засобів.

4.1. Шаблони функцій

Окреслимо коло задач, які порівняно часто виникають в повсякденній практиці програмістів, та для зручної підтримки якого виникає потреба в спеціальних мовних засобах.

Часто програмісту доводиться мати справу не з окремими функціями, а з сімействами функцій, абсолютно подібних між собою і по призначенню, і по внутрішній будові і по алгоритму роботи, які відрізняються лише тим, що працюють над даними різних типів.

Наприклад, розглянемо сфмейство функцій знаходження найбільшого з двох значень. Кожна функція з цього сімейства має два аргументи певного типу та повертає значення цього ж типу — найбільше з двох переданих їй на вхід значень. Нижче наведено реалізації кількох функцій з цього сімейства:

```
int max( int a, int b ) {
if( a > b ) return a;
return b;
}

char max( char a, char b ) {
if( a > b ) return a;
return b;
}

long max( long a, long b ) {
if( a > b ) return a;
```

```
return b;
}
```

Як видно, тексти цих функцій повністю однакові, з точністю до підстановки імені типу (**int**, **char** та **long**) в підкреслені позиції.

Написання таких сімейств майже однакових функцій має очевидні недоліки. По-перше, це «роздуває» текст програми. По-друге, програмісту доводиться витрачати робочий час на рутинну роботу — багаторазове клонування однієї «заготовки» програмного тексту та підстановку в неї імен типів. По-третє, якщо вже після того, як функція була розмножена кілька разів, програміст знайде в ній помилку, то і виправлення довелося б вносити вручну в усі функції даного сімейства.

Нарешті, ще один недолік полягає в тому, що, скільки б функцій не включити в сімейство для кожного типу даних, пізніше завжди може виникнути потреба поповнити сімейство новими клонами. Так, в наведеному прикладі немає функції для знаходження найбільшого з двох дійсних чисел

```
long max( long a, long b ) {
if( a > b ) return a;
return b;
}
```

та подібних функцій для типів **unsigned char**, **unsigned int** та безлічі інших. Передбачити наперед всі можливі випадки застосування ідіоми «знайти найбільше з двох значень» неможливо, оскільки її можна застосувати також до типів даних,створених самим програмістом (наприклад, до класу «раціональне число», тощо).

Отже, спосіб ручного написання таких сімейств функцій негнучкий, громіздкий, неуніверсальний, вимагає великого обсягу ручної роботи та не автоматизує рутинні дії над текстом (підстановку імен типів даних у потрібні місця). Саме для боротьби з цією проблемою призначений ще один механізм мови Сі++ — шаблони функцій.

Придивимося ще раз до всіх наведених вище екземплярів функції «знайти найбільше з двох значень» та виділимо в них спільне. Те, що, навпаки, відрізняється в кожній з них — ім'я типу даних, позначимо літерою `T`. Виходить оголошення шаблону:

```
template <typename T>
```

```
T max( T a, T b ) {
if( a > b ) return a;
return b;
}
```

Перший рядок означає, що це оголошення шаблону[1], та повідомляє транслятор, що шаблон має один параметр `T`, замість якого потім будуть підставлятися імена типів[2].

Як видно, один цей компактний запис є універсальною заготовкою безлічі функцій знаходження найбільшого для будь-яких типів даних. Якщо в подальшому тексті програми проінструктувати транслятор, щоб він взяв цей текст (крім першого рядка) та всюди замість `T` підставив певне ім'я типу даних, наприклад, **`double`**, то отримаємо вже конкретну і готову до застосування функцію знаходження найбільшого з двох дійсних чисел.

Зауважимо, що `T` (параметр шаблону) відіграє роль своєрідної «змінної», тільки, на відміну від звичайних змінних, її значеннями стають не цілі чи дійсні числа, а імена типів даних і, по-друге, обробляється ця «змінна» не на етапі виконання програми, а на етапі компіляції. Навіть оголошення «змінної» `T` за формою дуже подібне до оголошення звичайної змінної: запис **`int`** `x` означає, що `x` приймає значення серед цілих чисел, а запис[3] **`typename`** `T` означає, що `T` приймає значення серед імен типів.

Застосування шаблону в програмі в принципі нічим не відрізняється від застосування звичайної функції. Розглянемо кілька викликів розібраного вище шаблону:

```
int a=2, b=8, c;
double pi=3.41, e=2.72, x;
c = max( a, b ); // 1
x = max( pi, e ); // 2
```

В рядку з цифрою 1 транслятор виявить, що даний виклик співставляється з оголошенням шаблону, якщо в шаблон замість «змінної» `T` підставити ім'я типу **`int`**. Тому транслятор зробить таку під-

[1] англ. template — шаблон

[2] англ. type name — ім'я типу

[3] В старих компіляторах замість слова **`typename`** використовувалося слово **`class`**

становку, отримає текст конкретної функції для пошуку найбільшого з двох цілих, відкомпілює цей текст та побудує виклик отриманої функції. Так само в рядку з цифрою 2 транслятор побудує з шаблону функцію для дійсних чисел та викличе її.

Після першого знайомства з шаблонами функцій на прикладі треба розібрати їх більш строго та формально.

Як прийнято в мові Сі, дозволяється і навіть рекомендується окремо писати прототип шаблону, окремо його тіло. І прототип, і тіло шаблону функції починається з заголовку — ключового слова **template**, за яким в кутових дужках (знаках «менше» та «більше») пишеться список параметрів шаблону.

Шаблон функції може мати скільки завгодно параметрів. Параметрами шаблону функції можуть бути лише імена типів. Кожен параметр шаблону функції повинен оголошуватися з ключовим словом **typename** (або **class** для старих компіляторів).

Після заголовку шаблону йде прототип або тіло функції, в яких параметр шаблону може використовуватися будь-де, де має право стояти ім'я типу. Це, зокрема, означає, що параметр шаблону може відігравати роль (одну або декілька з перерахованих) типу аргумента функції, типу значення, що функція повертає, типу локальної змінної, типу об'єкту, для якого виділяється динамічна пам'ять (тобто параметр шаблону може стояти при операції **new**).

На відміну від звичайних функцій, у яких прототипи прийнято виносити в заголовочні файли, а тіла реалізовувати в модулях, у шаблонів функцій і прототип, і тіло мають міститися в заголовочному файлі. Справа в тому, що шаблон — це не готова до застосування функція, яку можна відкомпілювати заздалегідь, незалежно від компіляції того модуля, з якого вона викликається. Як було сказано вище, лише побачивши виклик шаблону, компілятор підставляє конкретні імена типів замість параметрів шаблону та компілює отриманий текст.

4.2. Шаблони класів

В попередньому розділі від практичної потреби у сімействах подібних між собою функцій ми прийшли до спеціального мовного засобу, шаблонів функцій. Подібні міркування можна повторити і для

сімейств схожих між собою класів, у яких також часто виникає потреба. Розглянемо, наприклад, класи, що реалізують

- вектор цілих чисел,
- вектор дійсних чисел,
- вектор комплексних чисел.

Очевидно, ці класи мають однакові переліки методів, однакові члени даних та однакові тіла методів — з точністю до підстановки імен **int**, **double** та `complex` у відповідні місця.

Оскільки загальна теорія про шаблони викладена в попередньому розділі, немає потреби в довгих поясненнях, і можна одразу переходити до прикладу. Нижче наведено повний текст шаблону класу «вектор».

```
template <typename T>
class CVector {
public:
  CVector( int n0 );
  ~CVector( void );
  int get_size( void );
  T get_element( int i );
  void set_element( int i, T x );
  void multiply_by( T x );
  T product( CVector<T>& v );
private:
  int m_size;
  T *p;
};

template <typename T>
CVector<T>::CVector( int n0 ) : m_size( n0 ) {
  p = new T[ m_size ];
}

template <typename T>
CVector<T>::~CVector( void ) {
  delete[] p;
}
```

```
template <typename T>
int CVector<T>::get_size( void ) {
  return m_size;
}

template <typename T>
T CVector<T>::get_element( int i ) {
  return p[i];
}

template <typename T>
void CVector<T>::set_element( int i, T x ) {
  p[i] = x;
}

template <typename T>
void CVector<T>::multiply_by( T x ) {
for( int i = 0; i < m_size; i++ )
  p[i] *= x;
}

template <typename T>
T CVector<T>::product( CVector<T>& v ) {
if( m_size != v.m_size ) return 0;
T s = 0;
for( int i = 0; i < m_size; i++ )
  s += p[i] * v.p[i];
return s;
}
```

Параметр `T` — це невідомий заздалегідь тип елементів вектору: скажімо, підставивши в шаблон замість параметру `T` ім'я **int**, отримаємо клас «вектор цілих чисел».

Підкреслимо, що `CVector` це не ім'я класу, а ім'я шаблону. Ім'ям класу є, наприклад, `CVector<int>` — конкретизація даного шаблону підстановкою слова **int** замість параметру.

Яким би не був тип `T`, вектор характеризується розмірністю (кількістю елементів) та сукупністю значень цих компонентів. Цьому відповідають члени даних `n` та `p`. Останній є покажчиком на масив елементів типу `T`, пам'ять для якого буде видялятися динамічно. Це

ілюструє, що параметр шаблону класу може використовуватися в якості типу членів даних.

Для створення нового вектору треба призначити йому розмірність, тому конструктор має один цілий аргумент `n0`. Перелік операцій для обробки векторів, крім конструктора та деструктора, включає: метод `get_size`, який дозволяє дізнатися у об'єкта-вектора його розмірність, метод `get_element`, який дозволяє дізнатися значення елементу під номером `i`, метод `set_element`, який елементу під номером `i` присвоює значення `x`, метод `multiply_by`, який кожен елемент даного вектору збільшує в `x` разів, метод `product`, який обчислює значення скалярного добутку даного вектору з вектором `v`. Цей перелік, звичайно ж, надто спрощений порівняно з повним набором функцій для роботи з векторами, необхідним у реальній програмі, і наведений лише як навчальний приклад. Студентам рекомендується самостійно дописати решту операцій.

Загальний висновок щодо оголошення полягає в тому, що оголошення шаблона класа практично не відрізняється від оголошення звичайного класу, а параметр шаблону може стояти в будь-якому місці, де має право бути ім'я типу — це може бути тип члена даних, тип значення чи аргумента методу.

Реалізації методів шаблона класу оформлюються як шаблони функцій. Звертаємо увагу: перед знаком `::`, де має стояти ім'я класу, до якого належить метод, стоїть не `CVector`, а `CVector<T>`. В імені конструктора і деструктора, однак, достатньо використати ім'я шаблону.

Робота конструктора полягає в тому, щоб присвоїти початкове значення члену даних `m_size` та динамічно виділити пам'ять для зберігання масиву елементів типу `T`. Слід звернути увагу, що параметр `T` використовується з операцією **`new`** так само, як і звичайне ім'я типу (справді, коли компілятор буде з шаблона робити конкретний клас, він підставить замість `T` ім'я типу, і операція виділення пам'яті прийме вигляд, наприклад, **`new int`**`[m_size]`).

Реалізація деструктора і методу `get_size` очевидна і коментарів не потребує. Методи `get_element` та `set_element` ілюструють, як методи можуть повертати значення та приймати аргументи того типу, який є параметром шаблону.

Метод `product` цікавий тим, що його аргумент сам належить класу, який транслятор отримає з даного шаблону. Справді, з точки зору предметної області можна говорити про скалярний добуток двох векторів цілих чисел, двох векторів комплексних чисел тощо — тобто коли обидва вектори мають однаковий тип елементів. Коли компілятор підставить в параметр шаблону ім'я типу, наприклад **int**, даний метод прийме вигляд:

```
int CVector<int>::product( CVector<int>& v )
```

Отже, метод класу «вектор цілих чисел» буде мати аргумент того ж класу «вектор цілих чисел» і повертати значення цілого типу. Цей приклад ілюструє, що в шаблоні класу (так само і в шаблоні функції) можна використовувати не лише сам параметр шаблону, але й будь-які типи даних, сконструйовані на його основі.

Тепер залишається розібрати, як використовуються шаблони класів в програмах. Продовжимо попередній приклад, допишемо основну функцію програми, я якій створюються конкретні об'єкти-вектори.

```
int main( void ) {
// робота з вектором цілих
CVector<int> u( 3 ); // три елементи
u.set_element( 0, 1 );
u.set_element( 1, 3 );
u.set_element( 2, 2 );
// вийшов вектор (1, 3, 2)
CVector<int> v( 3 );
v.set_element( 0, 2 );
v.set_element( 1, 4 );
v.set_element( 2, 0 );
// (2, 4, 0)
// надрукувати їх добуток
cout << u.product( v ) << endl;
// вектор дійсних, динамічно
CVector<double> *p;
p = new CVector<double> ( 4 );
p->set_element( 2, 1.27 );
delete p;
return 0;
}
```

Як видно, робота з шаблонними класами майже не відрізняється від роботи зі звичайними класами, за винятком того, що тут ім'ям класу є складна конструкція, така як `CVector<int>`, утворена з імені шаблону та конкретного імені типу-параметру.

Шаблони функцій продовжують та розвивають ідею перевантаження імен функцій. Справді, всі конкретні екземпляри функцій, створені транслятором на основі шаблону шляхом підстановки різних імен типів замість параметру, мають одне ім'я і відрізняються типами аргументів, як і функції з перевантаженими іменами. Але шаблони функцій йдуть далі: створюючи функції з перевантаженими іменами, програміст мусить вручну писати тіло кожної з них, а апарат шаблонів дозволяє автоматизувати навіть створення тіл функцій. Таким чином, перевантаження імен слід застосовувати тоді, коли кілька функцій подібні за призначенням, але різні за реалізацією, а шаблони — коли й реалізація цих функцій збігається.

В цілому, шаблони — це потужний засіб, який дозволяє писати універсальний код, придатний для багаторазового використання, коли один і той самий шаблон може включатися в різні програмні продукти. Для багатьох реальних задач, що часто трапляються на практиці, вже написано потужні бібліотеки шаблонів. Одна з них — бібліотека STL (Standard Template Library), створена Олександром Стєпановим, завоювала настільки широку популярність, що була включена в специфікацію мови Сі++ як частина стандарту мови. Це означає, що компілятори мови Сі++, виготовлені різними виробниками по всьому світу, неодмінно повинні підтримувати бібліотеку STL. Ця бібліотека включає в себе шаблони типових контейнерів (вектор, список, таблиця-відображення тощо), які використовуються в найрізноманітніших прикладних алгоритмах. Студентам варто самостійно поцікавитися цими контейнерами і спиратися на них у своїй роботі. Інший приклад потужного та елегантного набору шаблонів-контейнерів з ретельно продуманою логікою побудови і багатою функціональністю дає бібліотека Qt.

Розділ 5

Потоки введення-виведення

5.1. Основні поняття

Потоки введення-виведення у мові Сі++ являють собою цілу технологію зі своїм оригінальним задумом і з широким спектром різноманітних можливостей, об'єднаних у стрункy та логічну систему. Основні характеристики цієї технології:

- Операції введення-виведення безпечні щодо типів даних (якщо у мові Сі++ лише програміст слідкує, щоб специфікатор у форматному рядку відповідав фактичному типу змінної, у яку заносяться введені дані, то використання потоків мови Сі++ перекладає контроль типів на компілятор).
- Можна керувати параметрами і способом введення-виведення (встановлювати максимальну та мінімальну ширину поля, систему числення, спосіб вирівнювання тощо).
- Функціональність стандартних потоків введення-виведення можна як завгодно розширювати на свої власні типи даних (наприклад, написавши клас `CVector`, можна довизначити операцію `<<` так, щоб у потік `cout` можна було виводити об'єкти цього класу разом зі значеннями стандартних типів). Технологія потоків мови Сі++ дозволяє зробити введення та виведення об'єктів своїх власних типів таким зовні невідрізненним від введення-виведення стандартних типів даних.
- Універсальність потоків мови Сі++, незалежність способу роботи з потоком від фізичної реалізації потоку (тобто від того, є потік файлом на диску, екраном, принтером, з'єднанням по мережі, областю пам'яті тощо) покращена порівняно з мовою Сі. Якщо, наприклад, дано програму, що працює через потоки з екраном і клавіатурою, то її дуже легко перетворити так, щоб вона, скажімо, брала вихідні дані з файлу та записувала результат у деяку область оперативної пам'яті.

Саме поняття потоку, нагадаємо, сформувалося ще в мові Сі, яка містила вже доволі потужні засоби для роботи з потоками; бібліотека потокових класів мови Сі++ є подальшим кроком у розвитку тих же ідей. Потік являє собою уявний абстрактний пристрій введення-виведення та відіграє роль посередника між прикладною програмою та реальним фізичним пристроєм (таким як диск, принтер, клавіатура тощо). Програма спілкується щодо введення та виведення даних не безпосередньо з фізичним пристроєм, а з потоком, а потік, в свою чергу, обмінюється даними з пристроєм[1].

З точки зору програмного коду, який використовує потоки, потік виведення виглядає як труба, в яку програма може один за одним кидати об'єкти різних типів. Після того, як об'єкт вкинуто у трубу, його доля програму може не цікавити: потік сам подбає, щоб об'єкт було доставлено до конкретного пристрою. Так само, потік введення можна уявити у вигляді труби, по якій один за одним надходять об'єкти даних з пристроїв. Яким саме чиномвони знімаються з пристроїв та транспортуються по трубі, програму не цікавить — це справа самого потоку. Для програми важливо лише те, що вона може будь-коли звернутися до потоку та прийняти з нього черовий об'єкт. Поки програма до потоку не звертаєтья, займаючись своїми обчисленнями, об'єкти стоять в цій уявній трубі і чекають, поки програма їх звідти візьме.

На тому кінці труби, який звернуто до програми, є також пульт, з якого програма може керувати потоком, налаштовувати параметри його роботи. Скажімо, можна перемикати спосіб, у який відображаються на пристрої відправлені на нього числа (в десятковій чи шіснадцятковій системі числення, з вирівнюванням по правому чи лівому краю тощо), чи прив'язувати потік до певного файлу.

Виведення — це перетворення об'єктів деяких типів (наприклад, чисел) на дані іншого типу: на послідовність символів; введення, навпаки, є перетворенням послідовності символів на об'єкт даних (наприклад, число). Тому «потік» виглядає, якщо дивитися на нього з боку програми, потоком об'єктів, а з боку пристрою — потоком символів.

[1] Насправді, як буе видно з курсу системного програмування, потік спілкується з пристроєм також через ланцюжок посередників.

Стандартна бібліотека потоків мови Сі++ написана в об'єктно-орієнтованому стилі. Кожен потік в програмі зображується *об'єктом* певного класу; різним типам пристроїв відповідають свої потокові класи. Логічна структура бібліотеки побудована на наслідуванні: операції, спільні для будь-яких потоків, винесено до батьківських класів, а породжені класи відповідають лише за специфічні особливості тих чи інших конкретних різновидів потоку. Наприклад, клас `ostream` реалізує довільний потік виведення — в ньому реалізовано операцію поміщення об'єкту в потік і операції, що керують способом відображення значень при виведенні; породжений від нього клас `ofstream` реалізує потік виведення у файл: він наслідує операцію виведення і операції керування виглядом значень та доповнює їх операціями, специфічними саме для роботи з файлами. Від того ж класу `ostream` породжено і клас `ostrstream`, який також наслідує операції виведення даних в потік і керування способом виведення, але реалізовує їх по-іншому: всі дані спрямовуються в текстовий рядок, що вміщується в оперативній пам'яті.

З точки зору використання у прикладних програмах найважливішими є класи `istream`, `ostream` та `iostream` — вони реалізують відповідно потоки введення, виведення і двонаправлені (ведення та виведення). Добре знайомий з попередніх розділів потік `cin` є об'єктом саме класу `istream`, а потоки `cout` та `cerr` — об'єктами класу `ostream`. Клас двонаправленого потоку `iostream` породжено від класів `istream` та `ostream` — він наслідує від першого батьківського класу операції введення, а від другого — виведення. Класи `istream` та `ostream`, в свою чергу, породжено від батьківськоо класу =lstinline!ios!, в якому визначено властивості, спількі для потоків введення та виведення. Невеликий фрагмент ієрархії потокових класів показано на рис. 5.1.

5.2. Прості засоби введення-виведення

Тут ми розглянемо основні особливості поведінки об'єктів-потоків, в тому числі й об'єктів `cin` та `cout`. З усіх функціональних можливостей потоків найчастіше в прикладних програмах використовуються «стрілки» `<<` та `>>`, які показують напрямок руху даних (в потік чи з потоку). Нехай програмі треба ввести два цілих числа, і нехай

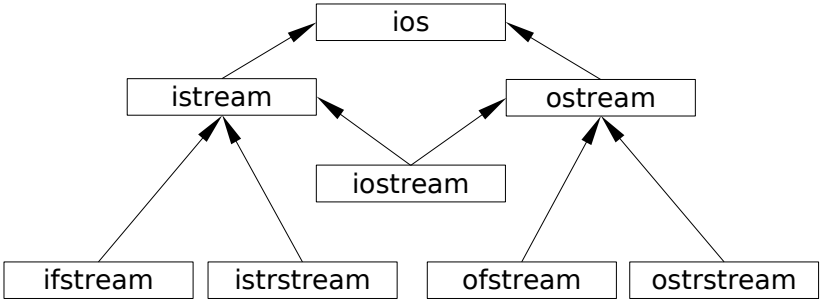

Рис. 5.1. Класи потоків введення-виведення

оголошено змінні `x` та `y` типу **`int`**. Тоді два наступні фрагменти коду цілком еквівалентні, поводять себе в точності однаково:

```
cin >> x >> y; // ланцюжком
```

та

```
cin >> x; // окремо
cin >> y;
```

Потоку все одно, застосовуються операції-стрілки по одній чи зчеплюються в ланцюжок.

Далі, в обох наведених випадках користувач може вводити числа будь-яким з двох наступних способів:

- Набрати перше число, пробіл, друге число, натиснути клавішу Enter.
- Набрати перше число, натиснути Enter, набрати друге число, натиснути Enter.

Іншими словами, потоку все одно, надходять до нього всі дані однією пачкою, чи поділені на дрібніші пачки. Так само, наступному коду

```
int s = 0, x;
for( int i = 0; i < 10; ++i ) {
  cin >> x;
  s += x;
}
```

все одно, чи буде користувач натискати Enter після кожного набраного числа, чи набере всі десять чисел, розділюючи їх пробілами, і натисне Enter один раз наприкінці, чи буде вводити по кілька чисел.

Узагальнимо сказане. При введенні значення простого типу потік читає набрані користувачем символи, поки не зустріне: (1) кінець введеної послідовності (Enter), (2) символ-роздільник (пробіл, табуляція), (3) недопустимий символ (наприклад, при введенні цілих чисел недопустимими є літери та знаки пунктуації). Якщо введених користувачем значень менше, ніж потрбно програмі (наприклад, коли у програмі стоїть оператор `cin >> x >> y`, а користувач вводить одне число і натискає Enter), то виконується стільки операцій введення, скільки можливо (в даному випадку одна), а наступна операція чекає, поки не надійдуть нові символи. Якщо ж користувач дає значень більше, ніж потрібно програмі (скажімо, набирає три числа, коли програма замовляє введення двох змінних), то всі значення змінних вводяться, а «зайві» значення, що не були введені, залишаються в *буфері введення* (спеціальній області пам'яті), звідки їх візьмуть наступні операції введення.

Розглянемо детальніше, як поводить себе потік з недопустимими символами, знову ж на прикладі введення двох цілих чисел. Якщо користувач введе `11Q84`, то перша операція введення виконається і введе число 11 до змінної `x`. Потім друга операція, яка теж намагається ввести число, «побачить» замість числа недопустимий символ «Q» та завершиться, нічого не ввівши. Змінна `y` зберігає своє попереднє значення, а ланцюжок символів `Q84`, які не вдалося ввести, залишається у буфері введення: саме він надійде до наступних операцій введення (якщо в явному вигляді не наказати об'єкту `cin` очистити буфер).

Перейдемо до введення текстових рядків. Найпростіший і очевидний спосіб

```
char s[10];
cin >> s;
```

має серйозні недоліки. По-перше, введений таким чином рядок не може містити пробілів: операція >> зазвичай вважає пробіл розділь-ником між двома сусідніми значеннями. По-друге, ця операція не може контролювати граничний розмір рядка. У прикладі для ряд-

ка `s` зарезервовано простір у 10 символів, значить «справжніх» символів в ньому може бути не більше 9 (ще один є завершальним нуль-символом), але якщо користувач введе 15 чи 100 символів, операція введення спробує розмістити в пам'яті їх всі, починаючи з адреси масиву `s` — це або зіпсує інші дані, або призведе до краху програми.

Для введення текстових рядків з контролем граничної довжини і з можливими пробілами призначено спеціальний метод `getline`. Він має два обов'язкових і один необов'язковий аргумент. Перший аргумент — покажчик на символьни масив, в який треба розмістити введений рядок, другий — ціле число n, розмір масиву. Метод гарантує, що введено буде не більш ніж $n-1$ символ, а наприкінці введеного рядка в масив запишеться нуль-символ. З використанням цього методу попередній приклад перетворюється наступним чином:

```
const int n = 10;
char s[n];
cin.getline( s, n );
```

5.3. Управління форматом

Потоки дозволяють керувати способом і виглядом символьного зображення значень, що вводяться та виводяться. Наприклад, числа можна виводити та вводити в десятковій, вісімковій та шіснадцятковій системах числення, вирівнювати при виведенні по правому чи лівому краю тощо. Більшість параметрів потоку можна налаштовувати двома способами: помістити в потік *маніпулятор* чи викликати спеціальний метод потокового об'єкта.

З точки зору використання маніпулятор це ніби спеціальний об'єкт; коли цей об'єкт надсилається до потоку операцією `<<`, на пристрій нічого не виводиться — натомість змінюється спосіб, яким потік виводить наступні значення. Таким же чином маніпулятори можна використовувати в ланцюжках операцій введення `>>`. Деякі маніпулятори мають сенс лише для потоків введення, деякі для потоків виведення, а деякі — для обох різновидів. Щоб зробити маніпулятори доступними для використання в програмі, треба підключити заголовочний файл `iomanip` (з розширенням `.h` у старих версіях мови). Найпоширеніші маніпулятори введення-виведення зібрано в табл. 5.1. В

третій колонці показано, для яких потоків має сенс той чи інший манімулятор: введення (I), виведення (O) чи обох.

Маніпулятори `dec`, `oct` та `hex` встановлюють, відповідно, десятковий, вісімковий та шіснадцятковий способи зображення цілих чисел як при введенні, так і при виведення. Наприклад, наступна програма запитує у користувача число в десятковій системі і друкує його значення в двох системах:

```
int x;
cin >> dec >> x;
cout << dec << x << hex << x << endl;
```

В останньому рядку в потік виведення поміщується маніпулятор `dec`, після якого потік починає виводити числа в десятковій системі. В цій системі надрукуєтсья наступне за маніпулятором значення змінної `i`. Далі до потоку надходить маніпулятор `hex` та перемикає його режим; потім виводиться знов те ж саме значення, але тепер воно надрукується вже у шіснадцятковій системі.

Існує також маніпулятор `setbase`, який має один цілочисельний параметр. Цей параметр може приймати значення 8, 10 або 16 і встановлює основу системи числення. За допомогою цього маніпулятора рядок з попереднього прикладу можна було б переписати так:

```
cout << setbase(10) << x << setbase(16) << x << endl;
```

Маніпулятор `setw` з цілочисельним параметром `n` встановлює ширину поля виведення: після цього маніпулятора значення будуть виводитися так, щоб займати рівно `n` позицій. При цьому зайві позиції заповнюються символом-заповнювачем, по замовчуванню це пробіл, але за доомогою маніпулятора `setfill` з параметром символьного типу можна встановити будь-який заповнювач. До того ж, якщо застосувати маніпулятор `right`, виведене значення вирівнюється по правому краю (тобто зайві позиції заповнюються до потрібної ширини зліва), а якщо діє маніпулятор `left` — по лівому краю (символи-заповнювачі додаються справа). Проілюструємо це прикладом. Наступна програма друкує таблицю, у якої в лівій колонці стоять квадрати цілих чисел k^2, а в правій — ступені двійки 2^k.

```
int p = 1;
cout << setfill('.');
```

Табл. 5.1. Маніпулятори введення-виведення

Маніпулятор	Значення	Напрямок
`dec`	Виводити числа в десятковій системі	I/O
`oct`	Виводити числа в вісімковій системі	I/O
`hex`	Виводити числа в 16-ковій системі	I/O
`setbase(n)`	Виводити числа в системі з основою n	I/O
`showbase`	Показувати основу системи числення	O
`noshowbase`	Показувати основу системи	O
`fixed`	Виводити дійсні числа в форматі `317.23`	O
`scientific`	Виводити числа в форматі `3.1723E2`	O
`showpoint`	Виводити нульову дробову частину: `1.000`	O
`noshowpoint`	Не виводити нулі попереду	O
`showpos`	Виводити + перед додатними числами	O
`noshowpos`	Не виводити знак + перед числами	O
`setw(n)`	Встановити ширину поля n символів	O
`setprecision(n)`	Втановити ширину дробової частини	O
`setfill(c)`	Заповнювати вільне місце символом `c`	O
`left`	Вірівнювати по лівому краю	O
`right`	Вірівнювати по правому краю	O
`internal`	Знак по лівому, число по правому краю	O
`boolalpha`	Логічні значення зображувати словами	I/O
`noboolalpha`	Логічні значення зображувати числами	I/O

```
for( int k = 0; k <= 14; ++k ) {
  cout << setw(8) << left  << k*k
       << setw(8) << right << p << endl;
  p*=2;
}
```

Перед тим, як виводити значення k^2, встановлюється вирівнювання по лівому краю і ширина поля 8 символів. Скільки б не займало знаків число k^2, решта знаків, якої не вистачає до 8, доповнюється точками. Так само, з доповненням точками до потрібної ширини, виводиться і значення змінної `p`, в якому зберігається поточне значення 2^k. Але перед виведенням цього значення встановлюється режим вирівнювання по правому краю. Таким чином, результатом виконання цього коду стає текст

```
0.............1
1.............2
4.............4
9.............8
16............16
25............32
36............64
49...........128
64...........256
81...........512
100.........1024
121.........2048
144.........4096
169.........8192
196........16384
```

Маніпулятор `setprecision` з цілочисельним параметром дозволяє встановити кількість знаків, що відводяться для дробової частини дійсного числа. Наступна програма друкує значення π, e та $\sqrt{2}$ з точністю, що збільшується (нагадаємо, що константи `M_PI`, `M_E` та функція `sqrt` оголошені в заголовочному файлі `math.h`):

```
double q = sqrt(2.0);
cout << left;
for( int k = 1; k <= 10; ++k )
  cout << setprecision(k)
       << setw(12) << M_PI
```

```
        << setw(12) << M_E
        << setw(12) << q << endl;
```

Перед друком чергового рядка встановлюється нове значення точності, тобто ширини дробової частини. Ширина поля встановлюється щоразу перед друком кожного числа: на відміну від інших модифікаторів, дія яких продовжується доти, доки не буде скасована наступним модифікатором, модифікатор `setw` діє лише на одне наступне за ним значення. Таким чином, програма виводить текст

```
3           3           1
3.1         2.7         1.4
3.14        2.72        1.41
3.142       2.718       1.414
3.1416      2.7183      1.4142
3.14159     2.71828     1.41421
3.141593    2.718282    1.414214
3.1415927   2.7182818   1.4142136
3.14159265  2.71828183  1.41421356
3.141592654 2.718281828 1.414213562
```

Ті ж самі параметри введення-виведення можна налаштовувати й іншим способом: викликаючи для потокових об'єктів спеціальні методи. Скажімо, встановити ширину поля виведення для наступного значення можна за допомогою методу `width` з одним цілим аргументом, ширину дробової частини — методом `precision` також з одним цілим аргументом, символ-заповнювач можна встановити методом `fill` з аргументом символьного типу. Методи з такими ж іменами, але без аргументів повертають поточні значення відповідно ширини поля, точності та заповнювача. Наприклад, замість `cout<<setw(5)` можна записати `cout.width(5)` — результат буде точно такий. Відмінність між перерахованими методами та відповідними маніпуляторами полягає лише у формі запису. В більшості випадків маніпулятори видаються зручнішими, бо їх можна вписувати прямо в ланцюжок операцій-стрілок, по кілька в одному операторі, а виклик методу повинен стояти окремим оператором.

Цей спосіб керування форматом покажемо на прикладі корисної функції, яка приймає масив цілих чисел, його довжину та деякий символ і будує за цим масивом стовпчату діаграму.

```
void barDiagram( int *p, int n, char c ) {
```

```
  for( int i = 0; i < n; ++i ) {
    cout.fill( c );
    cout.width( p[i] );
    cout << "\n";
  }
}
```

Як видно з коду, символ `c`, переданий до функції через аргумент, встановлюється як символ-заповнювач. Для кожного по черзі елементу масиву виводиться лише символ переходу на новий рядок. Але перед цим встановлюється ширина поля, рівна по величині відповідному елементу масива. Тому перед переходом на новий рядок буде надруковано відповідну кількість символів-заповнювачів. Нижче показано приклад програми, що викликає цю функцію.

```
int main( void ) {
  const int n = 7;
  int m[n] = { 20, 14, 18, 12, 6, 19, 24 };
  barDiagram( m, n, 'o' );
  return 0;
}
```

Результатом її роботи стає текст, у якому перший рядок складається з 20 літер «o», другий — з 14 і т.д.:

```
oooooooooooooooooo
oooooooooooooo
ooooooooooooooooo
ooooooooooo
ooooo
oooooooooooooooooo
oooooooooooooooooooooo
```

5.4. Операції введення-виведення для своїх класів

Одна з найпривабливіших рис бібліотеки потокових класів — це можливість як завгодно розширювати її функціональність власноруч. Ця можливість основана на тому, що стрілки `<<` та `>>` з точки зору реалізації являють собою перевантажені операції. Наприклад, у фрагменті коду

```
int x = 1;
cout << "Значення␣" << x;
```

перше та друге входження операції << є звертаннями до функцій

```
ostream& operator<<( ostream&, const char* );
ostream& operator<<( ostream&, int );
```

Перевантажена операція << повертає посилання на той же потік, який був у неї першим операндом. Саме це дозволяє в одному рядку коду застосовувати до потоку кілька таких операцій, як і у щойно наведеному прикладі.

Якщо тепер згадати про можливість самостійно визначати перевантажені операції для власних типів даних, стає зрозумілим, як «навчити» стандартні потоки `cin` та `cout` обробляти об'єкти будь-яких створених програмістом класів. Наприклад, якщо взяти клас `CVector` і описати перевантажену операцію

```
ostream& operator<<( ostream&, const CVector& );
```

то далі у програмі можна виводити у потік об'єкти-вектори так само, як і об'єкти стандартних типів:

```
int n = 3;
CVector u(n);
cout << "вектор␣" << u << endl;
```

Розберемо детально перевантажені операції введення-виведення векторів. Оскільки ці операції мають справу з членами-змінними, прихованими в об'єкті-векторі, варто оголосити їх дружніми до класу `CVector`:

```
class CVector {
public:
  // ...методи...
protected:
  int m_size;
  double *m_p;

friend ostream& operator<<(ostream&, const CVector&);
friend istream& operator>>(istream&, const CVector&);
};
```

Реалізація цих операцій доволі проста: треба по черзі вивести в потік (відповідно, взяти з потоку) всі компоненти вектора і наприкінці повернути посилання на той потік, до якого застосовано операцію. Крім того, зробимо так, щоб вектор брався круглі дужки, а між компонентами ставилися коми.

```
ostream& operator<<(ostream& s, const CVector& v) {
  s << "(";
  for( int i = 0; i < v.m_size; ++i ) {
    s << v.m_p[i];
    if( i < v.m_size-1 )
      s << ",␣";
  }
  s << ")";
  return s;
}

istream& operator>>(istream& s, const CVector& v) {
  for( int i = 0; i < v.m_size; ++i )
    s >> v.m_p[i];
  return s;
}
```

Бібліоґрафія

[1] Глушаков С.В., Коваль А.В., Смирнов С.В. Язык программирования C++: Учеб. курс. — Харків: Фолио, 2001. — 500 с.

[2] Дейтел П.Дж., Дейтел Х.М. Как программировать на C++. Введение в объектно-ориентированное проектирование с использованием UML. — М.: Бином, 2002. — 1152 с.

[3] Сабуров С.В. Языки программирования C и C++. — М.: Познавательная книга плюс, 2001. — 656 с.

[4] Шилдт Г. Справочник программиста по C/C++. — К.: Вильямс, 2001. — 448 с.

[5] Шилдт Г. Самоучитель C++. — СПб.: БХВ-Петербург, 2003. — 688 с.

[6] Марченко А.Л. C++. Бархатный путь. — М.: Горячая линия – Телеком, 1999. — 400 с.

[7] Страуструп Б. Язык программирования C++. — М.: Бином; СПб: Невский Диалект, 2001. — 1099 с.

[8] Вінник В.Ю.

[9] Элджер Дж.

ВІННИК Вадим Юрійович

Основи об'єктно-орієнтованого програмування мовою Сі++

Навчальний посібник

Оформлено в системі LaTeX, макрос $\mathcal{NCC}$

Редактор *Л.В. Гончарук*
Комп'ютерний набір та верстка *В.Ю. Вінник*
Макетування *В.В. Кондратенко*
Обкладинка *Д.В. Скачков*

Свідоцтво про державну реєстрацію КВ № 8079 від 30.10.2003. Підписано до друку 00.00.2008. Формат $60 \times 84\frac{1}{16}$.
Гарнітура Computer Modern.
Друк офсетний. Ум. друк. арк. 10
Тираж 300 екз. Зам. ??

Редакційно-видавничий відділ
Житомирського державного технологічного університету
10005, м. Житомир, вул. Черняховського, 103