РОЗДІЛ 6.
ОСНОВИ РОБОТИ З ФАЙЛАМИ Й ПОТОКАМИ

Бібліотеки систем програмування містять ієрархічну систему класів потоків уведення й виведення, у якій класи вищих рівнів успадковують атрибути й методи класів нижчих рівнів.

Розділ лише знайомить із невеличкою підмножиною бібліотечних засобів роботи з файлами. Їх докладніший розгляд виходить за межі цієї книжки.

6.1. Файли й потоки

Файл і потік

Під **файлом** (**фізичним файлом**) прийнято розуміти деяку іменовану область даних або послідовність байтів на зовнішньому носії даних, організовану певним чином.

Роботу з файлами здійснюють бібліотечні засоби в складі систем програмування. Завдяки цим засобам у С++-програмі описується робота не з файлом, а з його представником – **файловою змінною**.

У засобах уведення-виведення мови С++ основним різновидом файлових змінних є **потік** – об'єкт певного *бібліотечного класу*.

Бібліотечні класи потоків забезпечують простий інтерфейс обміну даними з різноманітними зовнішніми носіями, *незалежний від конкретних носіїв*. Обмін даними описується за допомогою методів цих класів і операцій << і >>.

Класи потоків утворюють ієрархічну систему – класи вищих рівнів використовують засоби, означені в класах нижчих рівнів. У цьому розділі описано лише окремі можливості кількох класів вищих рівнів, утім достатні для розв'язання широкого кола задач.

Потік як об'єкт має певний набір атрибутів, від значень яких залежить спосіб обробки файлу. Наприклад, існують різні *режими* обробки файлу – уведення або виведення, обробки даних з тим

або іншим їх перетворенням тощо. Обробка файлу може також мати різні *стани*, наприклад, коли все гаразд, або коли виникла помилка. Надання тих чи інших значень атрибутам потоку дозволяє керувати обробкою файлу. Це керування здійснюється за допомогою методів, означених у класах потоків.

Засоби введення-виведення, наведені нижче, розглядають і обробляють вміст файлу як *послідовність байтів*. У будь-який момент обробки файлу в ньому є тільки один байт, до якого може бути застосована наступна операція – **доступний байт**.

Коли виконуються послідовні операції введення або виведення, доступними по черзі стають послідовні байти файлу, тому інколи кажуть про **послідовний доступ** до файлу.

Початок і закінчення роботи з потоком

Розглянемо три класи потоків, якими можна користуватися для роботи з файлами на диску. Клас *вхідних потоків* `ifstream` (від англ. *input file stream*) призначено для введення даних із файлу, клас *вихідних потоків* `ofstream` (від *output file stream*) – для виведення у файл. Клас `fstream` дозволяє як уводити дані з файлу, так і виводити в нього. Щоб користуватися цими класами, потрібно включити файл заголовків `<fstream>`.

Потік спочатку необхідно *зв'язати* з файлом і *відкрити*, установивши потрібні значення атрибутів потоку. Ці дії виконує метод `open`. Першим аргументом у його виклику є рядкова константа, ім'я символьного масиву або інший вираз типу `char*`; компілятори середовища Microsoft Visual Studio, починаючи з версії 2012 року, допускають тільки вираз типу `string`. Вираз задає *шлях до файлу* у файловій системі (включно з власне його ім'ям). Другий і третій аргументи у виклику необов'язкові. За стандартних настройок вони задають режим роботи з файлом (відповідно до класів потоків це введення, виведення або введення-виведення) і можливість одночасної обробки в різних програмах.

Приклад. Оголосимо потік `fi` для введення даних і `fo` – для виведення. Зв'яжемо `fi` з файлом `in.txt` папки, в якій міститься програма, `fo` – з файлом `out.txt` у папці `D:\MyDir`.

```
ifstream fi; fi.open("in.txt");
ofstream fo; fo.open("D:\MyDir\out.txt");
```

Після цього відкривання в кожному з потоків доступним стає перший байт файлу, з яким зв'язано потік. Подальші операції виведення у потік `fo` знищують дані у файлі, якщо файл уже існував. ◂

Конструктори вказаних класів дозволяють відкрити й зв'язати потік просто в його означенні.

```
ifstream fi("in.txt");
ofstream fo("D:\MyDir\out.txt");
```

Конструктори забезпечують також інші способи ініціалізації потоків, але тут вони не розглядаються.

Успішно відкритий потік дозволяє обробляти файл – уводити дані з нього або, навпаки, виводити у файл. Після обробки потік треба закрити методом `close()`, наприклад `fo.close();`. Робота з вихідними потоками має певні особливості, через які частина даних, виведених у потік, до файлу насправді може й не потрапити. Закривання ж потоку гарантує, що всі виведені дані потрапляють у файл.

Стандартні файли й потоки

Потоки `cin` і `cout` застосовуються в багатьох програмах і є потоками класів `istream` і `ostream` ("потік уведення" і "потік виведення"). Ці класи використовують дані та методи, означені в класі нижчого рівня `ios`, і, у свою чергу, постачають засоби для класів `ifstream` та `ofstream`. Клас `iostream` є представником поняття "потік уведення-виведення" і основою для класу `fstream`. Класи `ios`, `istream` та `ostream` оголошено в бібліотечному файлі заголовків `<iostream>`.

Потоки з іменами `cin` і `cout` – це **стандартні потоки** введення й виведення, зв'язані зі **стандартними файлами** консолі – клавіатурою й екраном. Ці потоки означено в стандартному просторі імен `std`, тому для їх використання потрібна інструкція `using` або їх позначення з операцією `::` – `std::cin`, `std::cout`.

Зв'язувати й відкривати потоки `cin` і `cout` не потрібно. Необхідність їх відкривання може виникнути лише після того, як їх було закрито методом `close()`.

Помилки в потоці

Відкривання потоку може не бути успішним, наприклад, якщо файлу, з яким зв'язується потік, немає на диску. Після невда-

лого відкривання *в потоці виникає помилка* і спроби роботи з потоком можуть призвести до аварійного закінчення програми. Отже, варто перевірити, чи є в потоці помилка, і якщо є, то вжити відповідних заходів.

Перевірити, чи успішно відкрито потік, можна різними засобами. Метод `is_open()` повертає `true`, якщо потік відкрито, інакше повертає `false`. Метод `fail()` повертає ненульове значення, якщо в потоці трапилася помилка. Так само вираз, що є іменем потоку, у разі помилки має значення `false`. Отже, після зв'язування й відкривання потоку, наприклад `f`, ознакою помилки є ненульове значення виразу `!f.is_open()` або `f.fail()`, або `!f`.

✓ Після відкривання потоку варто додати інструкцію вигляду

```
if(ознака-помилки)
  {реакція на помилку}
```

Реагуючи на помилку, можна, наприклад, повернути ознаку невдалої спроби відкрити потік та ім'я файлу, з яким зв'язано потік, запитати інше ім'я файлу тощо.

Потік або ім'я файлу як параметр функції

За необхідності функцію, що працює з файлом, можна параметризувати ім'ям потоку або ім'ям файлу.

Параметр-потік може знадобитися, якщо, наприклад, робота з файлом починається до виклику цієї функції або закінчується після нього.

✓ Параметр, що є потоком, *має бути параметром-посиланням*, наприклад, як у заголовку `int inpFunc(ifstream & fi)`.

Якщо ж усю роботу з файлом зосереджено у функції, то можна зробити її параметром ім'я файлу, а потік оголосити в її тілі. Схематично функція може виглядати так:

```
int outpFunc(char * fName)     //або (string fName)
{ ofstream fo(fName);
  ...           // робота з потоком fo
  fo.close();
  return 0;
}
```

6.2. Виведення в текст і введення з тексту

Операція вставлення в потік

Текстові файли створюються дуже часто, оскільки це найзручніша форма передавання даних між різними програмними системами та/або користувачами, яка не залежить від внутрішнього зображення даних у системах.

Арифметичні значення й рядки можна вивести у файл, зображений потоком, за допомогою **операції вставлення** (або **виведення**) `<<`, означеної в класі `ostream` і застосовної також до потоків класів `iostream`, `ofstream` і `fstream`. Нехай `f` позначає потік одного з цих класів, *E* – довільний вираз, значенням якого є число, булеве значення, символ або рядок. У виразі вигляду `f<<`*E* обчислюється значення виразу *E*, створюється послідовність символів, що його зображує, і дописується до файлу.

Приклад. Розглянемо програму з функцією `outRands`, яка створює послідовність псевдовипадкових натуральних чисел і виводить їх по одному на рядок у файл. Головна функція передає ім'я файлу у виклику `outRands("rands.txt")` і повертає отримане значення (0 або 1). Якщо файлу `rands.txt` у папці з програмою не було, то він створюється.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <time.h>
using std::cout;
int outRands(char *fName)
{ ofstream f(fName);
  if(!f)
  { cout << "Problems with file " << fName;
    return 1;
  }
  int n = rand(), k;
  for(k=0; k<n; ++k)              //виведення чисел
    f << rand() << '\n';          //і кінців рядків
  f.close();
  return 0;
}
int main()
{ srand(unsigned(time(NULL)));
  return outRands("rands.txt"); }
```

Файл заголовків <iostream> потрібен для роботи з екраном, <fstream> – з потоком, <cstdlib> – для утворення псевдовипадкових чисел, <time.h> містить фукцію time.

Операція введення з потоку

Найпростіший за формою спосіб уведення даних – це **операція введення** (або **добування**) >>, означена в класі istream і застосовна також до потоків класів iostream, ifstream і fstream. Нижче в цьому розділі ім'я f позначає потік одного із зазначених класів.

Вираз уведення має вигляд f>>v, де v – позначення змінної. Операція >> дозволяє отримувати з потоку числові, символьні й рядкові значення, що утворюються за даними файлу. Тут розглянемо введення числових значень, а символьних і рядкових – у підрозд. 6.3.

Числові константи у файлі повинні мати вигляд, що відповідає їх типу, і відокремлюватися одна від одної не менш ніж одним *порожнім* (незначущим) символом – ' ', '\t', '\n' (узагалі, порожніми є символи з номерами 9–13 і 32).

Нехай v – змінна числового типу. Під час виконання операції f>>v, починаючи від доступного символу файлу, пропускаються порожні символи, якщо вони є. Далі вводяться символи константи, за ними утворюється відповідне числове значення й присвоюється змінній v. Після цього доступним у файлі стає порожній символ, наступний за константою, або кінець файлу, або символ, яким константу продовжити неможливо (якщо такий є в тексті).

Цикли введення

Уведення даних із файлу дуже часто є циклічним; умова продовження циклу виражає можливість уведення. Наприклад, уведення можна припинити, якщо досягнуто кінець файлу або в ньому виявлено помилкові дані. У цих ситуаціях без спеціальних дій з боку програми подальші спроби отримати дані з файлу не будуть успішними.

Розглянемо цикл, який закінчується після невдалої спроби введення. Скористаємося виразом (f>>v): якщо під час виконання операції f>>v уведення відбулося успішно, то значення

виразу відмінне від 0, інакше – нульове. Отже, вираз (`f>>v`) виражає *ознаку успішності введення*.

```
while(f>>v)
  обробка v              //значення введено
```

✓ Останнє обчислення умови продовження в цьому циклі, тобто невдала спроба введення, не змінює значення `v`.

Аналогічний цикл можна записати, щоб отримувати дані від клавіатури за допомогою потоку `cin`. Наприклад, розглянемо інструкції, які отримують цілі числа по одному від клавіатури й після кожного виводять суму введених чисел.

```
int n, sum=0;
cout << "Enter int: ";
while(cin>>n)
{ cout << (sum+=n) << endl;
  cout << "Enter int: ";
}
```

Виконання цього циклу закінчується, якщо замість того, щоб набрати цілу константу, користувач натискає на клавіші `Ctrl-Z`, а потім – на `Enter`.[7] Уведення також припиниться після того, як користувач уведе послідовність символів, які не утворюють цілої константи.

Кінець файлу й недопустимі символи

Основними причинами неуспішності спроби введення з файлу є *досягнення кінця файлу* й *поява помилкових даних*.

Якщо під час уведення даних досягнуто кінець файлу, то подальші спроби введення будуть неуспішними. Цю ситуацію можна виявити за допомогою методу `eof`. Якщо кінець файлу, з яким зв'язано потік `f`, досягнуто *після спроби введення* з нього, то виклик методу `f.eof()` повертає значення `true`. Якщо ж кінець файлу не досягнуто або спроб уведення з нього не було, то виклик `f.eof()` повертає `false`.

Помилкові дані – це дані, які не мають потрібної структури, зокрема містять символ, якого не може бути в правильних даних узагалі або в певному їх місці. Наприклад, у дійсній константі не може бути літери `M` або крапки після іншої крапки. Символ, яко-

[7] Це стосується роботи тільки на базі ОС сім'ї Windows.

го не може бути у відповідному місці в константі, назвемо **недопустимим**.

Якщо під час виконання виразу `f>>v` доступним стає недопустимий символ, то введення на ньому зупиняється, а подальші дії залежать від версії компілятора та типу змінної `v`. Найчастіше символ з потоку не береться й не обробляється. Змінна `v` при цьому може як отримати значення, так і не отримати.

Приклад. Нехай виконується вираз `f>>v`. Якщо файл містить символи `"12,3"`, то дійсна змінна `v` отримує значення 12.0, символ `","` після `12` залишається доступним, спроба введення є успішною, а значення виразу `f>>v` є ненульовим. Проте виконання виразу `f>>v`, коли доступним символом у файлі є `","`, не змінює значення `v` і залишає цей символ доступним, а значенням виразу стає 0. ◂

✓ Якщо під час зчитування вхідних даних з тексту необхідна гарантована й правильна обробка всіх наявних помилок, зокрема зв'язаних із форматом даних, то операцією вставлення `>>` з потоку краще *не користуватися*.

При появі помилки в потоці всі подальші операції з ним, окрім закриття, блокуються, поки не буде *скинуто помилку*. Для цього до потоку можна застосувати метод `clear()`. Не всі помилки можна скинути, але помилки, пов'язані з неправильним форматом даних у текстовому файлі, скинути можна.

Приклад. Припустимо, що файл `rands.txt` містить цілі константи, відокремлені порожніми символами. Уведемо всі числа, задані константами, і виведемо їх середнє арифметичне.

Напишемо функцію `inpInts`, параметризовану ім'ям файлу, яка визначає кількість цілих чисел у файлі й зберігає її у параметрі-посиланні. Якщо вхідний файл оброблено успішно, то функція повертає 0; якщо відкрити файл неможливо, то вона повертає -1; якщо виникла помилка формату вхідних даних, – то -2.

```
#include <iostream>
#include <fstream>
using namespace std;
int inpInts(char fName[], int &num)
{ ifstream f(fName);
  if(!f) return -1;
  int n=0, a;
```

```
while(f>>a)              //уведення чисел,
    {++n;}                 //обчислення їх кількості
  if(f.fail()&&!f.eof()) { //помилка формату
    f.close(); return -2; }
  f.close();
  num=n;
  return 0;
 }
int main(){
  int count=0;
  char fName[]="rands.txt";
  int res=inpInts(fName, count);
  if(res==-1)
    cout << "Problems with file " << fName<<endl;
  else
  if (res==-2)
    cout<<"Input file " <<fName<<
    " has not only integers"<<endl;
  else
    cout<<"The number of integers in " <<fName<<
          " is " << count <<endl;
  return 0;
}
```

Вправи

6.1. Написати програму створення тексту з таблицею степенів числа 2 від 1 до 62.

6.2. Написати програму створення тексту з цілими константами, які користувач уводить за допомогою клавіатури. Перший рядок тексту має містити кількість констант, другий – самі константи, відокремлені пропуском.

6.3. У перукарні працює один перукар. Клієнти приходять, займають чергу (якщо вона є) і стрижуться в порядку черги. Для кожного клієнта відома тривалість його стрижки *t*: клієнт залишає салон через *t* одиниць часу після початку стрижки. Моменти приходу клієнтів задано відносно початкового моменту часу в порядку неспадання. Текст містить по два цілих числа на рядок – момент приходу клієнта й тривалість

його стрижки. Вивести рядками в інший текст моменти приходу й виходу клієнтів.

6.4. Написати програму, яка вводить цілі числа з тексту й виводить їх на екран сторінками по 20 чисел (по одному на рядок). Після виведення кожної сторінки потрібно запитати користувача, чи продовжувати виведення, і в разі ствердної відповіді вивести наступну сторінку, інакше припинити роботу. Урахувати, що остання сторінка може бути неповною.

6.5. Написати програму, яка обчислює середнє арифметичне й дисперсію (середнє квадратичне відхилення від середнього арифметичного) цілих чисел, записаних у тексті. *Вказівка*. Після введення чисел і обчислення їх середнього арифметичного виправити помилку в потоці й закрити його, потім відкрити й увести числа вдруге.

6.6. Текст містить послідовність дійсних констант, що задають числа a_0, a_1, …. Про їх кількість відомо, що вона менше максимального значення типу `int`. Увести їх і вивести в інший файл "згладжену" послідовність b_1, b_2, …, де $b_1=(a_0+a_1)/2$, $b_2=(a_1+a_2)/2$, …. Якщо у вхідній послідовності тільки одне число, то воно й виводиться.

6.7. Є два тексти, в яких записано неспадні послідовності додатних цілих чисел. Записати в третій текст неспадну послідовність чисел, що є результатом злиття двох заданих. Наприклад, за послідовностями (2, 2, 4, 6) і (1, 3, 6, 7) утворюється (1, 2, 2, 3, 4, 6, 6, 7).

6.8. Числову множину зображено в текстовому файлі зростаючою послідовністю цілих чисел. За двома такими файлами створити третій файл, послідовність чисел у якому також є зростаючою й зображує: а) об'єднання; б) перетин; в) різницю; г) симетричну різницю двох заданих множин.

6.9. У тексті записано послідовність цілих чисел типу `int`; їх кількість нічим не обмежено. Відомо, що одне з них зустрічається в послідовності частіше, ніж усі інші, разом узяті. Знайти це число.

6.10. Реалізувати клас цілих масивів, довжина яких не більше 100. У класі мають бути методи введення масиву з файлу й консолі та виведення у файл і на консоль.

6.11. Реалізувати клас цілих матриць, розмір яких не більше ніж 20×20. У класі мають бути методи введення матриці з файлу й консолі та виведення у файл і на консоль.

6.3. Уведення символів і послідовностей символів

У цьому підрозділі ім'я `f` позначає потік одного з класів `istream`, `iostream`, `ifstream` або `fstream`.

Символи

Операція добування. За стандартних налаштувань, з погляду операції `f>>ch`, де `ch` позначає змінну типу `char`, символьною константою є непорожній символ. Символьні константи можуть як відокремлюватися порожніми символами, так і записуватися поспіль. Під час виконання `f>>ch` пропускаються порожні символи (якщо є), найближчий значущий стає значенням змінної, а наступний за ним стає доступним.

✓ Винятком є символ `(char)26` – він позначає кінець тексту й з потоку не зчитується. Значенням виразу `f>>ch` стає хибність, а `ch` не змінюється.

Приклад. Розглянемо головну функцію, яка в циклі вводить символи з файлу `infi.txt` і виводить їх разом з їх номерами на екран. Для її компіляції необхідно включити файли заголовків `<iostream>` і `<fstream>`.

```
int main()
{ char ch; ifstream f("infi.txt");
  if (!f))
  { cout << "cannot open input file\n";
    return 1;
  }
  while (f >> ch)      /* цикл уведення */
    cout<<ch<<"; int: "<<(int)ch<<'\n';
  f.close();
  return 0;
}
```

У файл `infi.txt` запишемо літери та пропуск у двох рядках.

```
ab
 c
```

Програма з цими вхідними даними виведе таке:

```
a; int: 97
b; int: 98
c; int: 99
```

Як бачимо, у файлі буде пропущено незначущі символи. ◂

Методи get і peek. У виклику `f.get(ch)` з потоку добувається доступний символ, яким би він не був (окрім `(char)26`); доступним стає наступний за ним. Якщо досягнуто кінець файлу, то змінна `ch` залишається без змін, а з виклику повертається нульове значення. Так, якщо в наведеному прикладі в циклі введення вираз `f>>ch` замінити виразом `f.get(ch)`, то обробка того самого файлу дає інший вихід.

```
a; int: 97
b; int: 98

; int: 10
 ; int: 32
c; int: 99
```

Тут порожні символи (їх номери 10 і 32) отримано зі вхідного потоку `f`. Вихідний потік `cout` інтерпретував символ кінця рядка (з номером 10), завдяки чому курсор на екрані перейшов у наступний рядок.

Виклик `f.peek()` повертає значення типу `int`, молодший байт якого зображує доступний символ потоку, тому це значення можна присвоїти символьній змінній. *Сам символ залишається доступним*. Якщо досягнуто кінець файлу, то виклик повертає цілу константу `-1`, іменовану `EOF`. При цьому помилка в потоці не виникає.

Послідовності символів

Операція добування. Операція введення у виразі `f>>s`, де `s` – адресний вираз типу `char*`, добуває з потоку послідовність символів і записує в послідовні байти, починаючи з того, на який указує `s`. При цьому, за стандартних настройок, пропускаються порожні символи й уводиться послідовність непорожніх. Особливості виконання й потенційну небезпеку цієї операції описано в підрозд. 2.1.

Якщо файл може містити порожні символи, то отримувати з нього порожні й непорожні символи краще за допомогою кількох переозначених методів `get` і `getline`. До того ж ці методи дозволяють контролювати кількість символів, які записуються в послідовні байти пам'яті.

Метод `get`. У виклику `f.get(s,lim)`, де значенням `lim` є додатне ціле число, з потоку добувається `lim-1` символ, а можливо, і менше – якщо раніше з'являється символ кінця рядка `'\n'` або кінець файлу. До символів, записаних у пам'ять за допомогою вказівника `s`, додається `'\0'`. Поява символу `'\n'` зупиняє введення, і цей символ залишається доступним. Добути його з потоку можна за допомогою, наприклад, методу `get` із символьним аргументом або методу `getline` (див. нижче).

Інший варіант цієї функції має додатковий параметр символьного типу. Виклик `f.get(s,lim,delim)`, де `delim` – символ, також добуває з потоку не більше ніж `lim-1` символ, але зупиняється не на `'\n'`, а на символі, заданому аргументом `delim`.

Метод `getline`. Виклик `f.getline(s,lim)` відрізняється від `f.get(s,lim)` тим, що символ `'\n'` у потоці пропускається і доступним стає символ, наступний за ним. Проте, якщо спочатку від доступного символу до найближчого `'\n'` більше ніж `lim-1` символ, то перші `lim-1` з них записуються в пам'ять, але виклик `getline` повертає нульове значення, а в потоці виникає помилка.

Інший варіант цієї функції має додатковий параметр символьного типу. Виклик `f.getline(s,num,delim)`, де `delim` – символ, аналогічно попередньому варіанту добуває з потоку не більше ніж `lim-1` символ, але обмежувачем добутих символів є не `'\n'`, а символ, заданий аргументом `delim`.

Вправи

6.12. Підрахувати, скільки разів у тексті з'являється кожне з 256 символьних значень, за виключенням `char(26)`.

6.13. Написати програму виведення вмісту файлу: а) на екран; б) в інший файл; в) у кінці іншого файлу.

6.14. Написати програму перевірки, чи збігаються послідовності байтів у двох файлах.

6.15. Рядок у тексті – це послідовність символів з `'\n'` наприкінці. Останній рядок цього символу може й не мати. Програма має підрахувати кількість рядків у файлі.

6.16. У кожному рядку тексту є послідовність дужок `(` і `)`, тобто *дужковий вираз*, а інших символів немає. Дужковий вираз є *правильним*, якщо це `()` або правильний вираз у дужках, або послідовність правильних виразів. Наприклад, вирази `(())`, `()()()` є правильними, вирази `(()`, `)(` – ні. З'ясувати, чи є вирази в рядках правильними, і вивести в інший текст послідовність із `0` і `1` (`1`, якщо вираз у рядку правильний, інакше `0`). Порожні рядки, тобто без дужок, ігнорувати. Вважати, що довжини рядків можна зобразити в типі `int`.

6.17. Скопіювати вміст файлу в інший файл, замінюючи символи кінця рядка пропусками.

6.18. Рядки у файлі, що закінчуються символом `'\n'`, мають довжину не більше 10000. Скопіювати вміст файлу в інший файл, не копіюючи рядки, в яких немає значущих (непорожніх) символів.

6.19. Слово – довільна послідовність непорожніх символів. Довжину слів у файлі нічим не обмежено. Вивести всі слова з тексту на екран по одному на рядок.

6.4. Буферизоване введення й виведення

Особливістю зовнішніх носіїв даних є те, що обмін даними між ними й оперативною пам'яттю відбувається великими порціями (десятки й сотні кбайт). Кожна операція обміну даними із зовнішніми носіями відбувається відносно повільно, тому бажано, щоб цих операцій було якомога менше. Проте в програмі зазвичай потрібна велика кількість операцій обміну даними, і дані найчастіше зображають окремі скалярні значення, тобто складаються з кількох байтів. Ця суперечність між можливостями зовнішніх носіїв даних і потребами програми розв'язується за допомогою *буферизації введення-виведення*.

Об'єкти класів, представлених у цьому розділі, містять спеціальний масив символів – *буфер*. Байти з файлу надходять у буфер або навпаки, з буфера у файл, великими порціями. Коли потік виконує операції введення, він обробляє байти не у файлі, а в

буфері. За символами, записаними в буфері, потік утворює арифметичні та інші значення, які присвоює змінним програми. Коли всі байти в буфері оброблено, а введення продовжується, буфер заповнюється наступною порцією байтів з файлу.

Виконуючи операцію виведення, потік накопичує символи у своєму буфері. Коли буфер заповнюється або в нього надходять певні символи, наприклад `'\n'`, його вміст однією великою порцією переписується у файл (буфер спорожнюється).

Спорожненням буфера можна керувати. Наприклад, виведення `flush` задає одноразове спорожнення буферу, а виведення `endl` додає в потік символ `'\n'` і спорожнює буфер.

У системі введення-виведення мови C++ є також класи потоків, які виводять дані у файл без накопичення їх у буфері, але тут вони не розглядаються.

Контрольні запитання

6.1. Що таке файл і доступний елемент файлу?
6.2. Які є різновиди файлів залежно від способу розгляду та обробки їх елементів?
6.3. Що таке файлова змінна? Що таке потік?
6.4. Які класи реалізують поняття вхідного й вихідного потоків?
6.5. Що таке зв'язування потоку з файлом і відкривання потоку?
6.6. Виразом якого типу зображується ім'я файлу?
6.7. Чи здійснюють зв'язування й відкривання конструктори класів `ifstream` і `ofstream`?
6.8. З якими файлами зв'язано потоки `cin` і `cout`? Чи потрібно їх відкривати у програмі?
6.9. Навести ознаку того, що потік не відкрився успішно.
6.10. Наслідки введення з файлу, коли в ньому досягнуто кінець.
6.11. Опишіть наслідки спроби введення числової константи, коли доступним у файлі є недопустимий символ.
6.12. Опишіть можливий вміст файлу, починаючи з доступного символу, за якого кінець файлу ще не досягнуто, але наступна спроба введення числового значення не буде успішною.

РОЗДІЛ 3.
СТРУКТУРИ ТА КЛАСИ

3.1. Структури

Почнемо з прикладу. Точка площини визначається двома декартовими координатами, які *утворюють пару* (x, y). Щоб реалізувати погляд на точки саме як на пари, бажано зображувати їх *складеними з двох частин, але єдиними змінними*.

Зображувати об'єкти, що мають кілька складових частин (компонент), можна у вигляді структур. Розглянемо оголошення імені типу структур для точок

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

Зарезервоване слово `struct` є скороченням від англ. structure – структура. Ім'я `Point` – це ім'я нашого типу ("Точка"). Далі у фігурних дужках указано типи та імена складових частин.

> Змінна типу структур називається **структурою**, а її складові частини – **полями**. Значенням структури є послідовність значень її полів.

Ім'я типу структур дозволяє оголошувати змінні, що зображують точки, наприклад `Point a, b;`. Поля змінних типу `Point` є дійсними змінними й позначаються іменами `x` та `y` після імені змінної типу структури: `a.x`, `a.y`, `b.x`, `b.y`. Ці поля можна використовувати так само, як інші дійсні змінні.

Імена типів структур прийнято починати з великої літери, імена змінних – з малої.

У виразі, що описує тип структур, однотипні поля можна оголосити разом (як однотипні змінні). Тип `Point` можна було б описати так:

```
struct Point { double x, y; };
```

✓ Поля структури можуть мати будь-які типи, скалярні чи структурні, не обов'язково однакові.

Приклади.

1. Припустимо, що коти характеризуються цілим віком (*age*) і дійсною вагою (*weight*). Для котів можна оголосити такий тип:

```
struct Cat { int age; double weight; };
```

2. Відрізок на площині можна зобразити в кілька різних способів, наприклад двома точками, що є його кінцями. Отже, розглянемо такий тип відрізків:

```
struct Segment { Point p1, p2; };
```

Якщо змінна `seg` має тип `Segment`, то її поля-точки ідентифікуються як `seg.p1` і `seg.p2`, а дійсні координати цих точок – як `seg.p1.x`, `seg.p1.y` тощо. ◂

✓ Структуру можна ініціалізувати, присвоїти іншій структурі, оголосити параметром функції або повернути з функції.

Приклади

1. Початкове значення структури задають у дужках `{}` після знака присвоювання `=` або без нього.

```
Point a={1,2}, b{2,3}; // ініціалізація
```

Значеннями полів `a.x` і `a.y` стають відповідно `1.0` і `2.0`, полів `b.x` і `b.y` – `2.0` і `3.0`.

Якщо ініціалізуюча послідовність містить менше значень, ніж є полів у структурі, то перші поля отримують задані значення, а решта – нулі відповідних типів. Наприклад, ініціалізація `Point a={1};` надає полям `a.x`, `a.y` значень `1.0`, `0.0`.

Ініціалізуючим виразом може бути також довільний вираз типу структури в дужках `()`, наприклад ім'я змінної: `Point a{1,2},c(a)`.

Неініціалізована структура в автоматичній пам'яті, як і будь-яка інша змінна, містить "сміття".

2. Якщо `a` й `b` – структури типу `Point`, то присвоювання `b=a;` за результатом рівносильне присвоюванням `b.x=a.x; b.y=a.y;`.

3. Функція введення точок зчитує координати в поля свого параметра-посилання типу `Point` і повертає ознаку успішності введення.

```
bool input(Point& a)
{ return (cin >> a.x >> a.y); };
```

Виклик функції має вигляд `input(a)`, де `a` – ім'я змінної типу `Point`, і є виразом логічного типу.

4. Наступна функція отримує точку як параметр, створює іншу точку, надає її полям значення, протилежні координатам параметра, і повертає її з виклику.

```
Point symmetric(const Point & a)
{ Point b; b.x=-a.x; b.y=-a.y;
  return b;
};
```

Виклик цієї функції є виразом типу `Point`. Його можна записати, наприклад, у присвоюванні: `p2=symmetric(p1)`.

5. Наведемо також функції виведення й порівняння точок:

```
void output(const Point & p)
{ cout << '(' << p.x << ',' << p.y << ')'; }
bool eq(const Point & p1, const Point & p2)
{ return (p1.x == p2.x && p1.y == p2.y); }
```

◂

Поля структури, на яку встановлено вказівник, можна позначати за допомогою операції розіменування * або **операції непрямого доступу** зі знаком `->`. Наприклад, якщо вказівник `Point* p` встановлено на деяку змінну, то її поля можна позначити `(*p).x`, `(*p).y` або `p->x`, `p->y`. Аналогічно можна присвоїти значення полям структури *p.

```
p->x=1; p->y=2*p->x; // подвоєне значення x
```

✓ Поля структур можуть бути як полями-даними, так і полями-функціями. Про цю можливість ідеться в наступних розділах, проте на прикладі не структур, а іншого різновиду структурованих даних – класів.

Вправи

3.1. Оголосити тип структури для зображення:

а) кола на площині;

б) прямокутника на площині;

в) студента, який зображується ім'ям (масив символів), масивом з 10 оцінок (цілі числа) і середнім балом (дійсне число).

3.2. Написати функції введення, виведення й визначення рівності структур з попередньої задачі.

3.2. Поняття класу

Приклад класу

Нехай є живі тіла, що мають масу. Поведінка тіл полягає в тому, що вони їдять, збільшуючи масу, і бігають, втрачаючи її. Також тіла порівнюються одне з одним своїми масами. Маса тіла може бути додатним цілим числом не більше 100 й може виводитися на екран. Тіло також має вік – невід'ємне ціле число, яке збільшується на один під час кожного прийому їжі.

Уточнимо тип тіл. Дані про тіло – це його вік і маса: `int age, mass`. Те, що відбувається з тілом, опишемо функціями з такими іменами:

`init` – ініціалізує масу тіла після його народження;

`eat` – прийом їжі збільшує масу на деяку величину й вік на 1;

`run` – біг зменшує масу;

`compare` – порівнюються маси двох тіл;

`out` – виводить масу й вік тіла на екран.

Наведені дані й функції можна розглядати як нарис типу тіл. Проте, по-перше, бажано, щоб опис типу був виразом. По-друге, кажучи про тип, зазвичай вважають, що інших операцій, окрім означених, немає, тобто всі інші дії *недопустимі, заборонені*. Отже, бажано мати означення типу, яке:

а) виглядає як *цілісний вираз*, що зображує тип саме як пару $T=(A,\Omega)$, де A – множина значень, Ω – перелік операцій;

б) дозволяє обробляти змінні цього типу за допомогою означених операцій і *забороняє будь-які інші дії* з ними, тобто забезпечує *захист змінних від недопустимого доступу*.

> **Клас** – це тип, заданий цілісним виразом, що описує значення та операції типу. Змінну типу, що є класом, називають **об'єктом**. Добре спроектований клас має забезпечувати захист об'єктів цього класу від недопустимого доступу.

Запишемо початковий варіант класу тіл (певні недоліки цього варіанта усунемо в підрозд. 3.3 та 3.6). Максимальну масу задамо цілою константою `MMAX`.

```
const int MMAX=100;
class Body {
private:            // приховані дані – вік і маса
  int age, mass;
public :            // відкриті операції з тілом:
  void init(int m);     // ініціалізувати з масою m
  void eat(int d);      // їсти
  void run(int d);      // бігати
  int compare(Body b);  // порівняти масу з тілом b
  void out();           // вивести вік і масу
};
```

Це оголошення каже: Body є ім'ям типу, яке дозволяє оголошувати змінні цього типу – об'єкти. Кожен об'єкт має поля даних з іменами age і mass. "Поля-функції" з іменами init, eat, run, compare, out зображують операції з тілом.

Поля-дані називаються **атрибутами об'єктів** класу, "поля-функції" – **методами класу**.

Поля-дані й виклики методів класу позначаються однаково – їх указують після імені об'єкта та крапки. Наприклад, якщо a – ім'я об'єкта типу Body, то вирази a.mass і a.age позначають поля mass і age об'єкта a, а вираз a.out() – виклик функції out для обробки об'єкта a.

Ім'я об'єкта, до якого застосовується метод, записується перед викликом методу, тому цей об'єкт *не зображується параметром методу*.

- ✓ Поля, оголошені після зарезервованого слова private (приватний, прихований), можна використовувати *лише* в методах цього класу.
- ✓ Поля, оголошені після слова public (*публічний*, *відкритий*), можна використовувати будь-де в області дії оголошення імені класу й відповідного об'єкта.

Отже, поля mass і age, приховані в класі Body, *доступні лише в методах* init, eat, compare, run, out, які означено нижче. За межами методів *приховані поля недоступні*.

Член класу – це атрибут або метод. Термін "член класу" – синонім терміна "поле класу".

Інтерфейс класу – це сукупність відкритих методів класу, тобто, неформально, опис, як обробляти об'єкти класу або як взаємодіяти з ними.

Програмуючи взаємодію з об'єктами класу, достатньо знати лише інтерфейс класу й навіть не замислюватися над тим, як реалізовано операції.

Реалізація та використання методів класу

Означимо (або *реалізуємо*) методи класу Body. Метод, означений за межами класу, у заголовку перед ім'ям містить ім'я класу й знак операції **розв'язання контексту** : : (*англ.* scope resolution).

У методі ініціалізації присвоїмо масі значення аргументу функції, якщо воно допустиме, інакше – половину максимально можливої маси. Вік тіла покладемо рівним 0.

```
void Body::init(int m){
  if(0<m && m<=MMAX) mass=m;
  else mass=MMAX/2;
  age=0;
}
```

У методі прийому їжі врахуємо, що маса тіла не може зменшитися або стати більше `MMAX`. Припустимо також, що кожний прийом їжі збільшує вік тіла на 1.

```
void Body::eat(int d){
  if(d<0)d=-d;
  if((mass+=d)>MMAX)mass=MMAX;
  ++age;
}
```

Біг зменшує масу тіла на задану величину. Проте, якщо маса тіла стає не більше 0, то тіло їсть, щоб його маса стала `MMAX/2`.

```
void Body::run(int d){
  if(d<0)d=-d;
  if((mass-=d)<=0) {mass=0; eat(MMAX/2);}
}
```

Маса тіла порівнюється з масою іншого тіла, зображеного параметром `b`. Виклик методу порівняння повертає -1, якщо тіло легше, ніж тіло `b`; повертає 0, якщо маси тіл рівні, і +1, якщо тіло важче, ніж тіло `b`.

```
int Body::compare(Body b) {
  if(mass < b.mass) return -1;
  else if(mass == b.mass) return 0;
  else return 1;
}
```

Масу тіла, що є першим операндом порівняння, тут позначає ім'я `mass` без уточнення, а масу другого операнда – ім'я з уточненням `b.mass`.

Нарешті, метод `out` виводить вік і масу тіла.

```
void Body::out(){
  cout << " Age:" << age << " Mass:" << mass;
}
```

В області видимості імені типу `Body` можна оголошувати змінні цього типу й описувати їх обробку за допомогою функцій `init`, `eat`, `run`, `compare`, `out`, *і ні в який інший спосіб*. Зокрема, можливість обробляти об'єкт-тіло лише методами класу `Body` гарантує, що маса тіла буде в допустимих межах. Нижче клас `Body` буде модифіковано й розвинено.

Оголошення класу містить заголовки функцій, що реалізують операції класу. У зоні дії оголошення класу можна записувати виклики цих функцій. Для прикладу наведемо початок програми з головною функцією, яка протягом деякого часу відстежує й порівнює маси двох тіл за припущення, що кожне тіло почергово їсть і бігає. Кількість одиниць маси, які тіла отримують або втрачають, утворюється за допомогою генератора випадкових чисел.

```
#include <iostream>
#include <cstdlib>
#include <time.h>
using namespace std;
int const MMAX=100;
class Body
{
  ... // див. вище
};
int main()
{ srand(unsigned(time(NULL)));
  Body a1, a2;
  a1.init(rand()*MMAX/RAND_MAX);
  a2.init(rand()*MMAX/RAND_MAX);
  for(int i=1; i<=7; ++i){
    a1.run(rand()*MMAX/RAND_MAX);    // бігають
    a2.run(rand()*MMAX/RAND_MAX);
    a1.eat(rand()*MMAX/RAND_MAX);    // їдять
    a2.eat(rand()*MMAX/RAND_MAX);
    a1.out(); a2.out();  // виведення віку й маси
    cout<<a1.compare(a2)<<'\n'; // порівняння мас
  }
  return 0;
}
// реалізація методів класу – див. вище
...
```

Об'єкт і виклик методу

Кожен об'єкт класу містить власний екземпляр полів-даних, але "поля-функції", тобто методи, в об'єкті насправді *відсутні* (тому цей термін береться в лапки). Кожен метод класу створюється в *одному екземплярі* незалежно від кількості об'єктів у програмі.

Уточнимо зв'язок між об'єктом і методом.

> **Об'єкт, якому належить виклик методу**, – це об'єкт, позначений перед викликом методу.

Наприклад, у наведеній програмі виклики методів класу `Body` належать об'єктам `a1` і `a2`. Виклик методу `compare` належить об'єкту `a1`, а об'єкт `a2` вказано у виклику як аргумент.

У методах класу перед іменами членів класу `mass` та `eat` немає імені об'єкта, якому вони належать, тобто ці імена в методах не уточнені.

- ✓ У тілі методу неуточнені імена членів класу позначають поля об'єкта, якому належить виклик методу.

Адреса об'єкта, якому належить виклик методу, передається у виклик як неявний додатковий аргумент. Адреса об'єкта присвоюється неявному параметру методу, що є вказівником і має ім'я `this`.

- ✓ Вказівник `this` дозволяє в тілі методу явно позначити об'єкт, якому належить виклик, і його поля.

Вираз `*this` позначає об'єкт, а вирази, наприклад, `(*this).mass` або `this->eat(MMAX/2)` у методах класу `Body` – поле цього об'єкта або виклик методу, який належить об'єкту.

Принцип інкапсуляції

> **Інкапсуляція** – це механізм, що описує дані та операції з ними в цілісній структурній одиниці, приховує їх реалізацію та *захищає* їх від будь-якого іншого використання, зокрема за межами описаних операцій.
>
> **Принцип інкапсуляції** полягає в організації даних і операцій їх обробки на основі інкапсуляції.

Інкапсуляція є *засобом забезпечення цілісності даних*, яка полягає в тому, що дані зберігають певний наперед означений

вигляд. Інкапсуляція дозволяє зберігати цілісність даних, оскільки запобігає змінам даних у недопустимий спосіб.

✓ Завдяки застосуванню інкапсуляції *підвищується надійність коду*.

Класи найбільш адекватно зображують поняття реального або уявного світу, а об'єкти – представників цих понять. У програмуванні з об'єктами програма створюється не як система окремих даних і підпрограм їх обробки, а як система об'єктів, що взаємодіють між собою та із "зовнішнім світом".

Кілька зауважень щодо класів

Імена класів прийнято починати з великої літери, об'єктів – з малої.

У класі може бути скільки завгодно розділів, що починаються заголовками `public:` або `private:`. Якщо на початку тіла класу жодного з цих заголовків немає, то поля до найближчого заголовка є *прихованими*.

Клас *може* мати *відкриті атрибути*, проте це суперечить принципу інкапсуляції, а тому *не рекомендується*. У класі можливі приховані методи (зазвичай вони є допоміжними до відкритих методів).

Клас *може* мати *вбудовані методи*, тобто означені цілком у класі. Наприклад, розглянемо метод виведення класу `Body`:

```
class Body {
public:
  ...
  void out()
  { cout << " Age:" << age << " Mass:" << mass;}
  ...
};
```

Вбудовані методи суперечать принципу **абстракції даних** – одному з важливих інструментів об'єктно-орієнтованого програмування. Основна ідея абстракції даних полягає у відокремленні використання об'єктів від деталей їх реалізації. Зокрема, абстракція даних дозволяє підвищити гнучкість коду. Отже, використовувати вбудовані методи *не рекомендується*.

Інколи, працюючи з об'єктами, потрібно отримувати або встановлювати значення їх окремих атрибутів. Для цього вико-

ристовують *методи отримання й установлення атрибутів* (від англ. getter і setter). Наприклад, до класу `Body` можна додати методи `getMass` і `setMass` отримання й установлення маси (у допустимих межах).

```
int Body::getMass(){ return mass; }
void Body::setMass(int m){
  if(0<m && m<=MMAX) mass=m;
  else mass=MMAX/2;
}
```

У мові C++ структури, як і класи, можуть мати "поля-функції", а також довільну кількість розділів із заголовками `public:` та `private:`. Проте, якщо на початку тіла структури жодного заголовка немає, то поля до найближчого заголовка, на відміну від класів, описаних зі словом `class`, є *відкритими*.

Ініціалізація та присвоювання об'єктів

Значення типу, що є класом, можна розуміти як значення структури, утвореної атрибутами. Найпростіший вираз типу, що є класом, – ім'я об'єкта. Звідси об'єкт можна *ініціалізувати за допомогою іншого об'єкта* або *присвоїти йому інший об'єкт*.

Приклад. Розглянемо присвоювання та ініціалізацію об'єктів класу `Body`, означеного вище.

```
Body p1;
p1.init(30);       //присвоювання маси в об'єкті p1
Body p2; p2=p1;    //присвоювання об'єктові p2
Body p3=p1,p4(p1); //ініціалізація об'єктів p3, p4
```

Значення маси, отримане об'єктом `p1` під час виклику `p1.init()`, копіюється в об'єкт `p2` , а потім у `p3` й `p4`. ◂

- ✓ Коли об'єкт *B* присвоюється об'єкту *A*, значення атрибутів об'єкта *B копіюються* в *A*. Присвоювання об'єктів виконує *неявний метод*, що його для класу створює компілятор.
- ✓ Мова C++ дозволяє означати *власні методи* ініціалізації та присвоювання об'єктів (про це див. у підрозд. 3.3 й розд. 4).

Вправи

3.3. Життєвий цикл тіла складається з того, що тіло спочатку бігає, а потім їсть. Написати головну функцію, яка створює два тіла й відстежує їхні стани до досягнення хоча б одним з них певного віку. *Вказівка*: додайте метод отримання віку.