МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Т. О. Карнаух
Ю. В. Коваль
М. В. Потієнко
А. Б. Ставровський

ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ C++

ОРГАНІЗАЦІЯ ДАНИХ

Навчальний посібник

КИЇВСЬКИЙ ВПЦ 2015 УНІВЕРСИТЕТ

УДК 004.43(075.8)
ББК 32.973.26-018.1я73
В85

Автори:
Т. О. Карнаух, Ю. В. Коваль, М. В. Потієнко, А. Б. Ставровський

Рецензенти:
д-р фіз.-мат. наук, проф. С. С. Шкільняк,
канд. фіз.-мат. наук, доц. П. П. Кулябко

Рекомендовано до друку вченою радою факультету кібернетики
(протокол № 2 від 13 жовтня 2014 року)

Ухвалено науково-методичною радою
Київського національного університету імені Тараса Шевченка
23 грудня 2014 року

В85 Вступ до програмування мовою С++. Організація даних : навчальний посібник / Т. О. Карнаух, Ю. В. Коваль, М. В. Потієнко, А. Б. Ставровський. – К.: Видавничо-поліграфічний центр "Київський університет", 2015. – 151 с.

Висвітлено основні поняття, пов'язані зі структурованими даними – даними, складеними з елементів, які можна ідентифікувати та обробляти як окремі одиниці. Представлено організацію даних засобами мови програмування С++. Особлива увага приділена класам як засобу реалізації загального поняття "тип", принципу інкапсуляції та успадкуванню класів. Наведено численні приклади, що наочно ілюструють викладений матеріал, і завдання для самостійної роботи.

Для студентів математичних напрямів навчання.

УДК 004.43(075.8)
ББК 32.973.26-018.1я73

ПЕРЕДМОВА

Пропонований навчальний посібник створено на основі базової дисципліни "Програмування" для студентів факультету кібернетики Київського національного університету імені Тараса Шевченка. Він є другим у серії книжок з цієї дисципліни. Багаторічне викладання свідчить, що організація даних у програмах опановується студентами не легше, ніж організація послідовностей обчислень.

Посібник висвітлює основні поняття, зв'язані зі *структурованими даними* – даними, складеними з елементів, які можна ідентифікувати та обробляти як окремі одиниці. Знайомство з організацією даних здійснюється з використанням мови програмування C++.

У посібнику розглядаються мовні засоби для утворення однорідних і неоднорідних послідовностей – масиви та структури. Представлені також динамічні дані, що створюються та обробляються за допомогою вказівників. Описано такі класичні типи організації роботи з даними, як стеки, черги, зв'язані списки та бінарні дерева. Посібник також знайомить із потоками та введенням і виведенням текстових даних.

Особлива увага приділена класам як засобу реалізації загального поняття "тип". Класи є засобом, який дозволяє визначати дані та операції з ними у вигляді цілісної структури й забезпечує захист даних від недопустимих дій. Саме завдяки концепції класів свого часу було значно піднято рівень надійності програмних систем.

Посібник призначений для використання на практичних заняттях і самостійної роботи студентів. Він містить численні приклади та вправи для підготовки до виконання комплексу лабораторних робіт. Робота з посібником передбачає знайомство з основами програмування в цілому й, зокрема, мовою C++ в обсязі, наприклад, посібника [1]. Для подальшого вдосконалення знань із C++ та техніки програмування рекомендуємо монографії [2–7] та останні стандарти мови [8, 9].

РОЗДІЛ 1.
МАСИВИ ТА ВКАЗІВНИКИ

1.1. Масив як змінна

Поняття масиву

У багатьох задачах у пам'яті програми треба зберігати великі набори однотипних даних. Для цього використовують масиви.

> **Масив** – це змінна, утворена послідовністю змінних, які називаються **елементами**, є однотипними й ідентифікуються номерами (**індексами**). Елементи займають послідовні ділянки пам'яті й до них є **прямий доступ**: будь-який елемент масиву доступний за допомогою його індексу.

Означення масиву має вигляд *T a[n]*, де *T* – тип елементів, *a* – ім'я масиву, *n* – *цілий константний вираз*, що задає довжину масиву. Елементи можуть мати довільний скалярний або структурний тип (масив та інші, про які йдеться в наступних розділах). Індексами елементів є цілі числа від `0` до *n*–1.

- ✓ Тип і кількість елементів масиву (його **довжина**) фіксуються в означенні або під час створення й далі не змінюються.
- ✓ Константу, що визначає довжину масиву, рекомендується іменувати.

Елемент масиву ідентифікується іменем масиву й індексом. Наприклад, елемент масиву зі ста елементів, `int a[100]`, позначається виразом вигляду `a[`*IE*`]`, де вираз *IE* повинен мати ціле значення від `0` до 99.

- ✓ Зазвичай програміст повинен сам стежити, щоб індекс елемента був у межах масиву. Вихід індексу за межі масиву може мати *непередбачувані наслідки*.

Приклади

1. Розглянемо інструкції, які дають ім'я `N` довжині 10, означають масив з `N` цілих елементів, присвоюють їм послідовні значення від 0 до 9 й виводять квадрати цих значень на екран.

```
const int N=10;
int a[N], i;
for (i=0; i<N; ++i)
  { a[i]=i; cout << a[i]*a[i] << ' '; }
```

2. Програма отримує мільйон псевдовипадкових цілих чисел від 0 до 99 й виводить кількість поява кожного з них. Числа утворюються як остачі від ділення на 100 значень, отриманих від бібліотечної функції `rand()`. Для роботи з нею потрібен файл `<cstdlib>`. "Зерно", яке визначає послідовність чисел, задається користувачем. Кожне число `rand()%100` використовується як індекс у масиві лічильників `cnt`, а кількість його появ стає значенням `cnt[rand()%100]`. Накопичені кількості виводяться.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
  const int MLN=1000000;
  const int N=100;
  int k, seed;
  cout << "Enter int seed>"; cin >> seed;
  srand(seed);
  int cnt[N]; for(k=0; k<N; ++k) cnt[k]=0;
  for(k=0; k<MLN; ++k)     //утворення та
    ++cnt[rand()%100];     //підрахунок чисел
  for(k=0; k<N; ++k)
    cout << k << ' ' << cnt[k] << endl;
  system("pause"); return 0;
}
```

◂

✓ Деякі, але не всі, компілятори дозволяють задати довжину масиву *ім'ям змінної* після того, як вона отримала значення, наприклад, так: `int n; cin>>n; int cnt[n];`. Проте покладатися на цю можливість *не будемо*.

Операція `sizeof`, застосована до імені масиву, дає *розмір масиву* в байтах. Наприклад, якщо є масив `int a[5]`, то `sizeof a` має значення 20 = 5×4, адже `sizeof int` дорівнює 4. Відповідно значенням виразу `sizeof(a)/sizeof(int)` є 5 – довжина масиву.

На практиці насправді може оброблятися лише частина елементів масиву, розташованих зазвичай на його початку. Звідси часто під довжиною масиву розуміють саме довжину послідовності значень на початку масиву, що утворюються та обробляються. Найчастіше цю реальну довжину зберігають в окремій змінній. Приклади наведено нижче.

Ініціалізація масиву

Значення, що ініціалізують послідовні елементи масиву, задаються списком констант у дужках {}, наприклад `int a[4]={2,10,3,15};`. Якщо констант менше, ніж елементів у масиві, то вони присвоюються першим елементам масиву, а решта елементів отримують нульові значення відповідного типу. Наприклад, означення `int cnt[4]={};` ініціалізує всі елементи значенням 0. Якщо констант у списку більше, ніж елементів масиву, то це є помилкою.

В ініціалізації можна не вказувати довжину масиву – вона стає рівною кількості констант. Наприклад, означення `int a[]={9,8,7};` задає масив із трьох елементів.

Якщо ініціалізацію не задано, то статичні змінні (зокрема елементи масивів, означених за межами функцій) як початкові значення отримують нулі відповідних типів. Значеннями автоматичних змінних, у тому числі й елементів масивів, є випадкове "сміття".

Приклад обробки масиву

Числа Фібоначчі було введено для підрахунку кількості пар кролів за такого припущення. Кожна пара кролів приносить щорічно приплід в одну пару (самку й самця), які, у свою чергу, починають давати приплід через два роки після народження. Смертністю кролів нехтують. Якщо спочатку є одна пара новонароджених кролів, то через n років буде F_{n+1} пар. Змінимо цю модель, вважаючи, що кожна тварина живе 4,5 роки, та обчислимо, скільки пар кролів і якого віку буде через n років, де n – задане натуральне число.

Запишемо в рядок початкові кількості кролів, вік яких 0 років (новонароджених), 1 рік, 2, 3 й 4 роки. У наступні рядки – кількості через рік, два роки тощо.

```
спочатку     : 1 0 0 0 0
через 1 рік  : 0 1 0 0 0
через 2 роки : 1 0 1 0 0
через 3 роки : 1 1 0 1 0
через 4 роки : 2 1 1 0 1
через 5 років: 2 2 1 1 0
```

Неважко зрозуміти, що кількість кролів віку i ($i > 0$) – це кількість кролів віку i–1 у попередній рік, а кількість новонароджених є сумою кількостей кроликів, вік яких становить 2, 3 й 4 роки.

Отже, промоделюємо життя кролів за допомогою масиву `int r[5]`. Індекс елемента зображує вік кролів, значення – кількість кролів цього віку. Ініціалізуємо масив значеннями {1, 0, 0, 0, 0}. Далі його обробка здійснюється в циклі по роках. На кожному кроці спочатку значення `r[3]` копіюється в `r[4]`, потім `r[2]` – в `r[3]`, `r[1]` – в `r[2]`, `r[0]` – в `r[1]` (саме в такому порядку!). Елемент `r[0]` отримує значення `r[2]+r[3]+r[4]`.

```
#include <iostream>
using namespace std;
int main()
{ int years;
  cout<<"Enter number of years (>0):\n";
  cin>>years; if(years<=0) return 1;
  const int N=5;     // довжина масиву
  int r[N] = {1};    // r[0]=1, решта нулі
  int y, i;          // рік і вік
  for(y=1; y<=years; ++y){
    for(i=4; i>0; --i)
      r[i]=r[i-1];
    r[0]=r[2]+r[3]+r[4];
  }
  for(i=0; i<N; ++i)
    cout << r[i] << " ";
  cout << endl;
  return 0;
}
```

Вправи

1.1. Модифікувати наведену вище програму про кролів за умови:
   а) кролі живуть не 4,5 роки, а 12,5 років;
   б) кількість кролів виводиться щорічно.

1.2. Риби народжуються навесні й живуть не довше 9,5 років. Навесні на кожну рибу припадає в середньому B новонароджених мальків. Кількість риб, незалежно від їх віку, за рік (від весни до весни) зменшується в D разів. Навесні першого року у водоймище випустили M новонароджених

мальків. Написати програму обчислення, скільки риби й якого віку буде у водоймищі через *Y* років.

1.3. За заданим роком і номером дня в році (від 1 до 365 або до 366, якщо рік високосний) обчислити дату (число, місяць). Наприклад, за 2012 61 і за 2013 60 обчислюється 1 3 – перше березня.

1.4. За заданою датою (число, місяць, рік, наприклад, 6, 5, 2012) обчислити номер дня в році (від 1 до 365 або 366, якщо рік високосний).

1.5. За допомогою генератора псевдовипадкових чисел утворити послідовність цілих чисел у діапазоні від 140 до 220 (см), що виражають зріст студентів. Вивести кількість студентів кожного можливого зросту.

1.2. Типізовані вказівники

У цьому підрозділі наведено мінімальні відомості про вказівники. Головним їх призначенням є робота з динамічними даними (див. розд. 4, 7).

Адреси та вказівники

Оперативну пам'ять, доступну програмі, можна розглядати як послідовність байтів; у кожного її байта є номер – **адреса**. Кожна змінна займає послідовні байти пам'яті, кількість яких визначається типом змінної. **Адреса змінної** – це адреса її першого байта.

Вказівник – це змінна, можливими значеннями якої є адреси.

Будемо розглядати лише **типізовані вказівники** – їхніми значеннями можуть бути адреси даних тільки певного типу. **Тип адрес** даних типу *T* позначається виразом *T**. При цьому тип *T* називається **базовим**, а вказівник типу *T** – **вказівником на дані типу** *T* (**вказівником типу** *T*).

✓ У 32-розрядних середовищах програмування вказівник *будь-якого типу займає чотири байти*, у 64-розрядних – вісім байтів.

Термін "вказівник" – це переклад англійського ***P**ointer*, тому варто дотримуватися неписаного правила: у кінці або на початку імен вказівників присутня літера p, а імена типів вказівників (про це йдеться в наступному підрозділі) починаються з P.

Символ * є окремою лексемою й в оголошенні вказівника стосується імені цього вказівника, а не типу.

Приклад. Інструкції оголошення

```
int * ip1, * ip2, i2; char *chp; char ** chpp;
```

визначають: змінні ip1, ip2 – це вказівники на цілі числа, i2 – ціла змінна, chp – вказівник на символи, chpp – вказівник на вказівники на символи.

Підкреслимо: наведені означення *не надають змінним значень*. Якщо ці змінні є автоматичними, то їхні значення – це випадкове "сміття". ◂

Розглянемо, як вказівнику присвоїти значення. Перша можливість – надати йому адресу деякої змінної. Адресу змінної позначає вираз із **операцією взяття адреси** &, наприклад, &x. Якщо адресу змінної x присвоєно вказівнику p виразом p=&x, то кажуть, що вказівник p *встановлено на змінну* x. Тоді значенням виразу p є адреса змінної. Присвоїти значення p іншому вказівнику означає встановити його на ту саму змінну.

Приклад. Інструкції

```
int x, *ip1 = &x, * ip2; ip2 = ip1;
char ch, *chp = &ch, ** chpp = &chp;
```

встановлюють вказівники ip1, ip2 на змінну x, chp – на ch, chpp – на chp (рис. 1.1). ◂

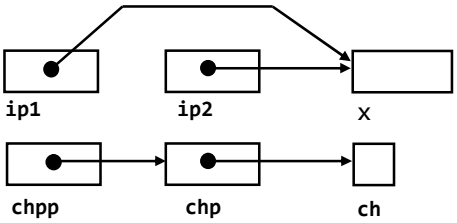

Рис. 1.1. Встановлення вказівників

Вираз *p з операцією **розіменування вказівника** * позначає дані, на які встановлено p, тобто *ідентифікує* їх.

Приклади

1. Після встановлень вказівників, як на рис. 1.1, вирази *ip1, *ip2 позначають змінну з іменем x, а вирази *chp, **chpp є си-

нонімами імені ch, тому кожен з виразів ch='a', *chp='a', **chpp='a' присвоює значення 'a' змінній ch.

2. У наведеній програмі функція ідентифікує дві цілі змінні своїми параметрами-вказівниками та обмінює місцями значення змінних. У функції вважається, що аргументи в її виклику дійсно задають адреси двох цілих змінних.

```
#include <iostream>
using namespace std;
void swapByPtrs(int * p1, int * p2){
  int t = *p1; *p1 = *p2; *p2 = t;
  return;
}
int main(){
  int a=1, b=2;
  swapByPtrs(&a, &b);
  cout << a << " " << b << endl; // вихід: 2 1
  system("pause");
  return 0;
}
```

Ім'я вказівника й вираз вигляду *&x*, де *x* – ім'я змінної, є найпростішими **адресними виразами** – виразами, значеннями яких є адреси. У мові C++ є також адресна константа NULL, яка позначає адресу 0. Ця адреса не може бути адресою жодних даних, тому найчастіше використовується як ознака того, що вказівник на жодні дані не встановлено. Складніші адресні вирази розглядаються нижче.

✓ Якщо *AE* – адресний вираз, то вираз вигляду **AE* позначає дані, адреса яких є значенням виразу *AE*.

Оголошення імені типу вказівників

Типу можна дати ім'я за допомогою **інструкції оголошення імені типу** typedef такого загального вигляду[1]:

```
typedef  вираз_що_задає_тип  ім'я;
```

Виразом, що задає тип, може бути ім'я стандартного або іншого типу, оголошеного програмістом, або складніший вираз, що описує

[1] Насправді вигляд може бути й іншим, але в даному випадку це не важливо.

тип. Також зазначимо, що в мові C++ таке оголошення задає не новий тип, а нове ім'я вже існуючого типу.

Приклади

1. Після інструкції

```
typedef int * PInt;
```

ім'я `PInt` позначає тип вказівників на цілі типу `int`. Це дозволяє оголошувати вказівники без знаків * перед іменами.

```
PInt pi1, pi2;
```

Оголошення імені типу вказівників гарантує від такої "дитячої" помилки, коли мали оголосити два вказівники, а оголосили вказівник `pp1` і цілу змінну `pp2`.

```
int * pp1, pp2;
```

2. Оголосимо імена типів вказівників на символи й вказівників на вказівники на символи.

```
typedef char * PChar;
typedef char ** PPChar;
```

Далі встановимо два вказівники й двічі виведемо символ `'a'`.

```
char ch = 'a';
PChar p1 = &ch;
PPChar p2 = &p1;
cout << *p1 << **p2;
```

◀

Імена типів вказівників замість виразів із * спрощують вигляд заголовків функцій, зокрема оголошення параметрів-посилань, що є вказівниками. Порівняйте такі прототипи.

```
int * f1(int * & p);
PInt f2(PInt& p);
```

Вони допустимі й еквівалентні, але другий виглядає простіше.

Арифметичні дії з адресами

Мова C++ дозволяє додати до адреси або відняти від неї ціле число. Можна також до цілого числа додати адресу. Отже, якщо *A* та *I* позначають, відповідно, адресний і цілий вирази, то *A+I*, *A–I*, *I+A* є адресними виразами.

Якщо доданок-адреса має тип `T*`, то ціле значення розглядається як *кількість одиниць даних типу* `T`. Наприклад, якщо `x` – змінна типу `int`, то адресні вирази `&x+1` і `1+&x` задають адресу цілої змінної, розташованої відразу після змінної `x`. Ця адреса

більше адреси змінної x на sizeof(int), тобто на 4. Віднімання цілого від адреси аналогічне. Так, значенням виразу &x-1 є адреса змінної, яка в пам'яті займає місце безпосередньо перед x.

Звідси, якщо p – вказівник типу T*, то вирази p+1, p+2, … задають адреси даних типу T, розташованих після *p "на відстані" 1, 2, … від *p. Так само вирази p-1, p-2, … позначають адреси даних типу T перед *p (рис. 1.2).

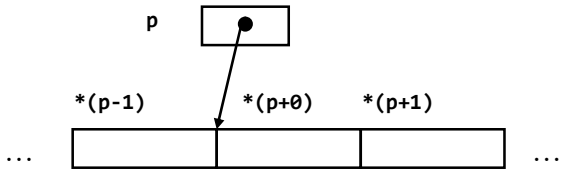

Рис. 1.2. Адресні вирази й дані

До вказівників як змінних застосовні складені присвоювання +=, -= з цілим виразом праворуч, а також ++ і --. Вони збільшують або зменшують значення вказівника на відповідну кількість sizeof(T), тобто переставляють його з кроком sizeof(T). Наприклад, якщо p має тип int*, то вираз p+=2 збільшує значення p (*адресу*) на 8, переставляючи його вперед на дві одиниці даних типу int.

До однотипних адрес застосовна операція віднімання. Якщо *AE1*, *AE2* – адресні вирази типу *T**, то значенням *арифметичного* виразу *AE2-AE1* є *ціла кількість одиниць даних типу T*, які "уміщаються між адресами" *AE1* і *AE2* (можливо, це число від'ємне). Зокрема, виразами *AE2* та *AE1* можуть бути імена вказівників.

Нарешті, адреси однотипних даних можна **порівнювати** за допомогою операторів ==, !=, <, <=, >, >=. Наприклад, якщо p – вказівник на деякий тип, то значенням виразів p<p+1, p!=p-1 є істина.

✓ Арифметичні операції з вказівниками вимагають підвищеної уваги. Помилки в цих операціях призводять зазвичай до спроб змінити зовсім не ті дані, які передбачає програміст, тому є *дуже небезпечними*.

Вправи

1.6. Припустимо, що p – вказівник, встановлений на цілу змінну зі значенням 3. Що можна сказати про значення виразів p, *p, &p, *&p й &*p?

1.7. Чому дорівнюють `sizeof(char)` і `sizeof(char*)`?

1.8. Припустимо, що `p1` і `p2` – вказівники типу `int*`. У чому різниця між присвоюваннями `p1=p2` та `*p1=*p2`? Чи є допустимими присвоювання `*p1=p2`, `p1=*p2`, `p1=&p2`?

1.9. Що буде виведено за програмою?

```
#include <iostream>
using namespace std;
void swapPtrs(int * & p1, int * & p2){
  int* t = p1; p1 = p2; p2 = t;
  return;
}
int main(){
  int a=1, b=2;
  int *pA=&a, *pB=&b;
  swapPtrs(pA, pB);
  cout << a << " " << b << endl;
  cout << *pA << " " << *pB << endl;
  system("pause");
  return 0;
}
```

1.3. Вказівники й масиви

Вказівники на елементи масиву

Якщо `p` – вказівник типу `T*`, то вираз вигляду `*(p+i)`, де `i` має ціле значення, ідентифікує дані типу `T` "на відстані `i`" від `*p`. Ці самі дані позначає й вираз `p[i]`, тобто вирази `*(p+i)` та `p[i]` *еквівалентні*; але вираз другого типу підтримується більшою кількістю мов програмування, ніж перший, і не використовує технічні особливості реалізації масивів у мові C++.

✓ У мові C++ ім'я масиву – це *адресний вираз, що позначає незмінюване значення*, яке є адресою першого елемента масиву. Ім'я масиву може бути операндом арифметичних операцій з адресами (звісно, окрім присвоювань).

Приклади

1. Нехай оголошено масив цілих `a` й вказівник на цілі `p`.

```
int a[10]; int * p;
```

За дії цих оголошень обидва вирази `p=&a[0]` і `p=a` встановлюють `p` на елемент `a[0]`. Після цього вирази `p[0]`, `p[1]`, …, `p[9]` ідентифі-

кують елементи масиву a так само, як і вирази a[0], a[1], …, a[9]. Можна встановити p на якийсь інший елемент масиву, присвоївши адресу елемента, наприклад, p=&a[5] або p=a+5.

2. Розглянемо таку програму:

```
#include <iostream>
using namespace std;
typedef int * PInt;
int main(){
  int a[] = {10, 11, 12, 13};
  int N = sizeof(a)/sizeof(int);
  PInt p = a, p2;
  for(int i=0; i<N; ++i) cout<<p[i]<<' '; /*1*/
  cout << endl;
  for(p2=a+N; p2>a; cout<<*(p2-=1)<<' '); /*2*/
  cout << endl;
  for(p2=a; p2<a+N; cout<<*(p2++)<<' ');  /*3*/
  cout << endl;
  cout << p2-a << ' ' << *a << endl;
  return 0;
}
```

Спочатку утворюється масив із чотирьох елементів. Його довжина обчислюється та зберігається в змінній N. Вказівник p встановлюється на його перший елемент.

У циклі в рядку /*1*/ вказівник p вказує на початок масиву, а вираз p[i] за значень i 0, 1, 2, 3 позначає послідовні елементи масиву. Виводяться їхні значення від 10 до 13.

У рядку /*2*/ вказівник p2 встановлюється на ділянку пам'яті *після останнього елемента* масиву. Далі в циклі адресне значення p2 спочатку зменшується на величину sizeof(int), тобто на 4, а потім виводиться значення елемента масиву, на який він вказує. Ці дії закінчуються, коли p2 досягає першого елемента масиву. Виводяться значення від 13 до 10.

У рядку /*3*/ вказівник p2 встановлюється на початок масиву. Далі дії аналогічні /*2*/, лише p2 *збільшується після виведення*, а все закінчується, коли значення p2 вийде за межі масиву. Виводяться значення від 10 до 13.

Наприкінці виводиться 4 – кількість змінних типу int, що вміщаються між вказівниками a й p2, тобто довжина масиву, і 10 – значення елемента з індексом 0.

Параметри функції, що зображують масив

Ім'я масиву є адресним виразом, незмінюваним значенням якого є адреса початку масиву. У виклику функції в мові C++ підстановка аргументу, який є масивом, на місце параметра – це *передавання адреси* початку масиву в пам'ять виклику функції. Звідси параметр функції, що зображує масив, *має бути вказівником і параметром-значенням*.

Оголошення параметра, що зображує масив у функції, зазвичай має вигляд `T ім'я[]` або `T *ім'я`, де `T` – вираз, що задає тип елементів масиву. Обидва вирази в заголовку функції задають одне й те саме – *вказівник на дані типу* `T`.

- ✓ Якщо параметр функції `a` оголошено у вигляді `T a[]` або `T *a`, то вираз `sizeof(a)` у тілі функції має значення 4 – кількість байт у вказівнику (у 64-розрядному середовищі буде не 4, а 8). У мові C++ елементи масиву, заданого аргументом у виклику функції, у пам'ять виклику *не копіюються*.

Проте адреса початку масиву ніяк не вказує на його довжину, необхідну для обробки масиву у функції. Отже, довжину масиву у функції зазвичай зображує окремий параметр.

Приклад. Розглянемо функцію, що виводить значення елементів масиву цілих. Параметрами є масив і його довжина.

```
void outAIntN(const int a[], int n)
{ for (int i=0; i<n; ++i)
    cout << a[i] << ' ';
  cout << endl;
  return;
}
```

Якщо у виклику цієї функції, наприклад `outAIntN(x, m);`, указано масив `x`, що має розмір `NMAX`, то значення `m` має бути не більше `NMAX`. Це повинен забезпечити програміст, що пише виклик, адже у функції справжня довжина масиву-аргументу невідома. Якщо значення `m` більше `NMAX`, то під час виконання виклику обробляються дані за межами масиву, а це може мати непередбачувані наслідки. Отже, функція працює з масивами будь-якої довжини, і це може бути *небезпечним*.

Специфікатори `const`, записані в заголовку наведеної функції, означають: *елементи й довжина масиву незмінювані* в тілі функції. Узагалі специфікатор `const` перед іменем типу позна-

чає незмінюваність даних цього типу. Отже, елементи масиву цілих у наведеній функції мають тип `const int`, тобто "незмінюване ціле". Водночас параметр `a` – це вказівник на дані цього типу, тобто змінна, й у функції за потреби його можна змінювати. ◂

Обробка початку масиву. У багатьох ситуаціях функція має обробляти не всі елементи масиву, а деяку їх частину, найчастіше – на початку масиву. Довжину цієї *робочої частини* масиву доцільно зобразити окремим параметром. Якщо ця довжина перед викликом функції невідома, то параметр має бути параметром-посиланням.

Приклад. Функція має заповнювати масив цілих значеннями, що їх задає користувач за допомогою клавіатури. Ознакою кінця введення елементів масиву є введення символів, що не зображують ціле значення. Перший параметр функції зображує масив, другий – його довжину. Значенням третього параметра (це параметр-посилання) під час виконання виклику стає кількість елементів, що отримують значення. Функція повертає булеве значення – ознаку того, що введено не більше значень ніж уміщує масив.

```
bool inAInt(int a[], int Nmax, int & n)
{ cout << "Enter int values (not more than " <<
          Nmax << ")\n";
  bool ok=true; n=0; int v;
  while(ok && cin >> v)
    if(n<Nmax)
      a[n++]=v;
    else ok=false;
  return ok;
}
```

Вираз `n<Nmax` у тілі циклу не дозволяє записати в масив більше ніж `Nmax` значень. Вираз уведення `cin >> v` записано в умові продовження. Він має ненульове значення, якщо введення відбулося, а інакше – значення 0. Уведення не відбувається, якщо символи, набрані користувачем, не зображують значення з типу `int`, зокрема, якщо користувач натиснув на клавіші `Ctrl-Z` і `Enter`. Це дозволяє користувачу заповнити не весь масив, а лише деякий його початок. ◂

Вправи

1.10. Написати функцію, яка за двома цілочисловими масивами однакового розміру визначає, чи мають їх відповідні елементи (з однаковими індексами) однакові значення. За її допомогою визначити, чи збігаються перша й друга половини масиву (за непарного розміру серединний елемент масиву не розглядається).

1.11. Написати функцію, яка виводить значення елементів свого параметра-масиву з парними індексами (0, 2, 4 тощо). Скористатися нею для виведення значень елементів масиву з парними індексами, а потім – елементів з непарними індексами.

1.12. Написати програму з трьома функціями: уведення значень елементів масиву цілих (можливо, значення отримують не всі елементи), виведення, порівняння двох масивів. У головній функції ввести два масиви довжиною не більше 10, вивести їх і результат їх порівняння (масиви рівні, мають різні довжини, відрізняються деякими елементами).

1.13. Прочитати натуральне число типу `int`, основу системи числення p, де $p < 37$, і вивести:

а) p-ковий запис числа;

б) значення p-кових цифр у вигляді багаточлена зі степенями числа p; піднесення до степеня позначити символом `^`, множення – символом `*`. За цифри 0 відповідний степінь числа p не виводиться, а за цифри 1 виводиться без неї, наприклад, за числа 1407 і основи 10 виводиться багаточлен `10^3+4*10^2+7*10^0`. *Вказівка*. Заповнення масиву, що зображує запис числа, і його виведення оформити у вигляді окремих функцій.

1.14. Написати функцію, що за масивом дійсних знаходить середнє арифметичне значень його елементів.

1.15. Написати функцію, яка за послідовністю дійсних чисел x_0, x_1, …, x_{n-1} у масиві обчислює таку величину:
$x_{n-1}(x_{n-1}+x_{n-2})(x_{n-1}+x_{n-2}+x_{n-3})\ldots(x_{n-1}+\ldots+x_0)$.

1.16. Написати функцію, що циклічно зсуває на одну позицію ліворуч значення елементів масиву цілих, не використовуючи додаткових масивів.

1.17. Написати функцію, що циклічно зсуває на одну позицію праворуч значення елементів масиву цілих, не використовуючи додаткових масивів.

1.18. Написати функцію, що без використання додаткових масивів обертає послідовність цілих у масиві. Наприклад, послідовність 0, 1, 2, 3, 4 має перетворитися на 4, 3, 2, 1, 0.

1.19. Написати функцію, яка без використання додаткових масивів перебудовує масив так, що спочатку підряд у тому ж самому порядку розташовані всі ненульові значення його елементів, а потім усі нульові.

1.20. Дано натуральне число n і дійсні числа $x_1, \ldots, x_{2n}$, зображені в масиві. Підрахувати суму тих чисел з $x_{n+1}, \ldots, x_{2n}$, які за величиною перевищують кожне з чисел $x_1, \ldots, x_n$.

1.21. Написати функцію, що без використання додаткових масивів перебудовує масив так, щоб спочатку були розташовані всі від'ємні елементи масиву, потім усі нульові, а потім усі додатні. Порядок елементів усередині групи можна змінювати.

1.4. Масиви, елементами яких є масиви

Поняття масиву масивів

Прямокутна таблиця з $m \times n$ однотипних елементів має *два виміри* – рядки й стовпчики. Її можна розглядати як послідовність m рядків, кожен з яких має n елементів, тобто як *масив, елементами якого є масиви*. Погляд на двовимірну таблицю як на масив масивів утілено в мові C++. Наприклад, таблицю цілих чисел 2×3 (2 рядки й 3 стовпчики) можна зобразити масивом `int a2[2][3]`. Цей масив має два елементи: кожен є масивом із трьох цілих змінних і зображує рядок таблиці (з індексом 0 або 1). Елементу таблиці відповідає ціла змінна, визначена двома індексами: перший указує рядок, другий – стовпчик. Послідовні цілі змінні в пам'яті зображують таблицю "рядками": `a2[0][0]`, `a2[0][1]`, `a2[0][2]`, `a2[1][0]`, `a2[1][1]`, `a2[1][2]`.

Послідовність двовимірних таблиць утворює тривимірну таблицю. Її можна зобразити масивом, елементи якого зображують двовимірні таблиці. Наприклад, масив `int a3[5][2][3]` має п'ять елементів, кожен як масив `a2`.

Масиви, елементами яких є масиви, для зручності будемо називати **багатовимірними**.[2] Наприклад, масив `a2`, наведений вище, назвемо двовимірним, масив `a3` – тривимірним. Перший вимір масиву називається **зовнішнім**, останній – **внутрішнім**. Наприклад, у масиві `int a3[5][2][3]` зовнішній вимір містить п'ять елементів (це двовимірні масиви), внутрішній – три (цілі змінні). Елементи внутрішнього виміру (як цілі змінні в масиві `int a3[5][2][3]`) будемо також називати елементами багатовимірного масиву.

Елементи тривимірного масиву `int a3[5][2][3]`, тобто цілі змінні, розташовуються в пам'яті аналогічно змінним масиву `int a2[2][3]`:

```
a3[0][0][0], a3[0][0][1], ..., a3[0][1][2],
...
a3[5][0][0], a3[5][0][1], ..., a3[5][1][2].
```

Отже, у послідовних елементів багатовимірного масиву найшвидше змінюється внутрішній індекс, найповільніше – зовнішній.

Ім'я багатовимірного масиву є незмінюваним адресним виразом, значення якого – адреса першого елемента масиву (що сам є масивом).

Приклади

1. Нехай означено масив `int a2[2][3]`. Значення виразів `a2` й `a2+1` – це адреси початків першого й другого масивів (рядків таблиці), кожен з яких складається із трьох цілих змінних. Вирази `a2[i]` та `*(a2+i)`, де `i` = 0 або 1, позначають `i`-й масив так само, як ім'я `a2` – увесь двовимірний масив. Аналогічно значеннями цих виразів є *адреси перших елементів масивів*. Звідси вирази `*(a2+i)+j` та `a2[i]+j`, де `j` = 0, 1, 2, є адресними й значенням кожного з них є адреса цілої змінної, що розташована в `i`-му "рядку" на `j`-му місці, тобто адреса цілого елемента `a2[i][j]`. Цей елемент позначають також вирази `*(*(a2+i)+j)`, `*(a2[i]+j)` та `(*(a2+i))[j]`.

2. Нехай означено масив `int a3[5][2][3]`. Значенням виразу вигляду `a3+i` (значення `i` від 0 до 4) є адреса початку `i`-го масиву з 2×3 цілих змінних. Вирази `*(a3+i)` та `a3[i]` позначають `i`-й масив 2×3; значенням цих виразів є його адреса. Аналогічно

[2] У мові C++ цього терміна немає, на відміну від деяких інших мов програмування.

значенням виразу `*(a3+i)+j`, де `j`=0 або 1, є адреса масиву з трьох цілих змінних, а виразу `*(*(a3+i)+j)+k` – адреса `k`-го елемента цього лінійного масиву, тобто адреса цілої змінної `a3[i][j][k]`. Її позначають також вирази, наприклад `*(a3[i][j]+k)` або `(*(a3+i)+j)[k]`. ◂

Багатовимірний масив ініціалізується аналогічно одновимірному, лише ініціалізуючим виразом може бути як послідовність значень, так і послідовність послідовностей. Зокрема, для двовимірного масиву може бути один або два рівні дужок `{}`, для тривимірного – від одного до трьох рівнів тощо. Наприклад, такі дві ініціалізації створюють масиви з однаковими значеннями (11, 12, 0 у першому рядку, 21, 0, 0 – у другому).

```
int a2[2][3] = {11,12,0,21};
int b2[2][3] = {{11,12},{21}};
```

Параметри для багатовимірних масивів

Параметри, що у функціях зображують багатовимірні масиви, розглянемо на прикладах із двовимірними масивами.

Невизначений зовнішній розмір. Параметр функції, що зображує багатовимірний масив, є вказівником на елементи масиву, які самі є масивами.

Елементи двовимірного масиву позначаються виразами вигляду `a[i][j]`, `(*(a+i))[j]` або `*(*(a+i)+j)`. Щоб компілятор правильно обробив ці вирази, йому потрібен розмір рядків масиву (розмір внутрішнього виміру). Розмір є добутком довжини рядка-масиву й розміру елементів масиву.

✓ Оголошуючи параметр, що зображує багатовимірний масив, *необхідно вказати довжину по всіх вимірах*, окрім зовнішнього.

Параметр, що зображує матрицю цілих, можна оголосити, наприклад, як `int a[][10]` або `int (*a)[10]`. За обома оголошеннями `a` є вказівником на масиви з 10 цілих, тому вираз вигляду `a[i][j]` або `(*(a+i))[j]` ідентифікує елемент у *i*-му рядку та *j*-му стовпчику. Аналогічно параметр, що зображує тривимірний масив цілих, можна оголосити як, наприклад, `int a[][20][30]` або `int (*a)[20][30]`.

В оголошенні параметра, наприклад `int(*a)[5]`, *дужки обов'язкові*, оскільки пріоритет постфіксної операції `[]` вище, ніж префіксної `*`. Вираз без дужок `int*a[5]` і еквівалентний йому `int*(a[5])` оголошують масив із 5 вказівників на цілі.

Для роботи з параметром функції, що зображує багатовимірний масив, може знадобитися параметр для довжини за зовнішнім виміром. Проте частіше в багатовимірному масиві обробляється робоча частина – перші рядки й перші стовпчики. Тоді у функції потрібні параметри, що зображують робочі довжини по всіх вимірах масиву.

Приклади.

1. Розглянемо програму з двовимірним масивом цілих `m`, що зображує матрицю розміром у межах 20×30. Довжини вимірів масиву (кількість рядків і стовпчиків) задано глобальними константами `RMAX` і `CMAX`. Матриця може мати й менше ніж 20 рядків або 30 стовпчиків, тому справжні кількості рядків і стовпчиків матриці зберігаються в окремих змінних `r` і `c`.

Функція `fillRands` отримує від користувача кількість рядків і стовпчиків матриці. Ці величини визначаються, коли виконується виклик функції, тому їх зображено параметрами-посиланнями. Якщо вони більше ніж, відповідно, `RMAX` і `CMAX`, то функція змінює їх; далі вона заповнює матрицю псевдовипадковими цілими числами.

Функція `outMInt` виводить значення елементів матриці. Вона отримує вже відомі кількості рядків і стовпчиків, тому має параметри-значення.

Головна функція оголошує змінні й викликає функції заповнення матриці та її виведення.

```
#include <iostream>
using namespace std;
const unsigned RMAX = 20, CMAX = 30;
void fillRands(int m[][CMAX],
       unsigned & r, unsigned & c)
{ cout << "Enter numbers of rows and columns "
       << "(positive less or equal "
       << RMAX << " and " << CMAX << "): " ;
  cin >> r >> c;

 if(r>RMAX) {
    r=RMAX;
    cout << "Too many rows. Set to " << RMAX;
  }
```

```
  if(c>CMAX) {
    c=CMAX;
    cout << "Too many columns. Set to " << CMAX;
  }
  int i, j;
  for(i=0; i<r; ++i)
    for(j=0; j<c; ++j)
       m[i][j]=rand();
}
void outMInt(int m[][CMAX],
       unsigned r, unsigned c)
{ for(int i=0; i<r; ++i)
    { for(int j=0; j<c; ++j)
        cout << m[i][j] << ' ';
      cout << endl;
    }
}
int main()
{ int m[RMAX][CMAX];
  unsigned r, c;
  fillRands(m, r, c);
  outMInt(m, r, c);
  return 0;
}
```

2. Функція `transp` транспонує квадратну матрицю, тобто відображає її відносно головної діагоналі. Кількість рядків і стовпчиків у квадратній матриці однакова й залишається без змін, тому її зображує параметр-значення `r`. Матриця займає частину масиву з довжиною рядків `NMAX`, тобто вважається, що значення `r` не більше `NMAX`.

```
void transp(int m[][NMAX], int r)
{ int i, j; int t;
  for(i=0; i<r; i++)
    for(j=i+1; j<r; j++)
      { t=m[i][j]; m[i][j]=m[j][i]; m[j][i]=t; }
}
```

Матриця з фіксованими розмірами. У багатьох задачах фігурують матриці, обидва розміри яких лежать у певних межах.

Можна оголосити ім'я типу двовимірних масивів для зображення матриць і користуватися ним у заголовках функцій.

Нехай матриці цілих чисел мають розміри в межах 20×30. Для їх зображення оголосимо ім'я типу масивів. В інструкції `typedef` ім'я типу масивів записується перед описом вимірів.

```
const unsigned RMAX = 20, CMAX = 30;
typedef int MInt[RMAX][CMAX];
```

Після цих оголошень можна означати та обробляти змінні типу `MInt` – двовимірні масиви. У зоні дії цих оголошень запишемо функцію виведення значень елементів матриці.

```
void outMInt(const MInt m, unsigned r, unsigned c)
{ // вважаємо, що r<RMAX, c<CMAX
  int i, k;
  for (i=0; i<r; ++i)
    { for(k=0; k<c; ++k)
        cout << m[i][k] << ' ';
      cout << endl;
    }
}
```

Вправи

1.22. Написати функцію побудови за дійсними числами a_0, a_1, …, a_{n-1} ($n \le 50$) такої квадратної матриці:

$$\begin{pmatrix} a_0 & a_1 & \dots & a_{n-2} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & a_0 \\ \dots & & \dots & & \dots \\ a_{n-2} & a_{n-1} & \dots & a_{n-4} & a_{n-3} \\ a_{n-1} & a_0 & \dots & a_{n-3} & a_{n-2} \end{pmatrix}$$

1.23. За лінійним масивом a_0, …, a_{n-1} побудувати таку матрицю Вандермонда:

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ a_0 & a_1 & \dots & a_{n-1} \\ \dots & \dots & \dots & \dots \\ a_0^{n-1} & a_1^{n-2} & \dots & a_{n-1}^{n-1} \end{pmatrix}.$$

1.24. Написати функцію побудови матриці розмірністю $n \times n$ з елементами $a_{i,j} = \begin{cases} \sum_{k=i}^{j} \frac{x^k}{k!}, i \le j, \\ a_{j,i}, i > j. \end{cases}$

1.25. У матриці поміняти місцями: а) два рядки; б) два стовпчики; в) одночасно два рядки й два стовпчики, задані номерами.

1.26. Написати функцію множення двох матриць.

1.27. Не використовуючи додаткових масивів, повернути квадратну матрицю за годинниковою стрілкою на: а) 180°; б) 90°.

1.28. Не використовуючи додаткових масивів, транспонувати матрицю.

1.29. За квадратною числовою матрицею одержати послідовність чисел обходом матриці: а) "змійкою", починаючи по горизонталі з лівого верхнього кута; б) "спіраллю" за годинниковою стрілкою з лівого верхнього кута.

1.30. За квадратною числовою матрицею одержати суму чисел, записаних: а) у центральному ромбі; б) у правому чвертьтрикутнику.

1.5. Вказівники: додаткові можливості

Перетворення типів адрес

✓ Присвоювати адреси даних одного типу вказівникам на дані іншого типу *заборонено*. Проте адресу даних одного типу можна *перетворити* до типу адрес іншого типу.

Приклади

1. Якщо `a` – ім'я масиву цілих чисел, то на його перший байт можна встановити вказівник на символи:

```
char*p = (char*)&a[0]; або
char*p = (char*)a;.
```

2. Розглянемо функцію, в якій масив `ch` із чотирьох символів (по одному байту) обробляється як одне ціле число типу `int` за допомогою вказівника `p` на цей тип. Спочатку значення елементів масиву виводяться. Потім вказівник `p` встановлюється на елемент `ch[0]`, адресу якого перетворено до типу "адреса даних типу

int". Далі обробляється значення змінної *p – виводяться значення шістнадцяткових цифр числа, а цифри "викреслюються".

```
int main(){
  const int N=4;
  char ch[N] = {'0', 'A', '1', 'a'};
  for (int i=0; i<N; ++i)
    cout << '<' << ch[i] << "> ";
  cout << endl;
  int*p = (int*)ch;
  while (*p)
  { cout << (*p)%16 << ' ';
    *p/=16;
  }
  return 0;
}
```

Виходом функції будуть два таких рядки:

```
<0> <A> <1> <a>
0 3 1 4 1 3 1 6
```

Перший містить символи в масиві, другий – шістнадцяткові цифри цілого числа в тому самому масиві, від молодшої до старшої. Вони свідчать: символ '0' має шістнадцятковий код 30, 'A' – 41, '1' – 31, 'a' – 61. Звідси також ясно, що біти цілого значення насправді розташовані в пам'яті від молодшого до старшого, а не так, як ми їх зображуємо (старший – ліворуч, молодший – праворуч).

- ✓ *Обробка даних одного типу як даних іншого типу* (зокрема, за допомогою вказівників) вимагає підвищеної уважності й знання подробиць і особливостей зображення даних.

Вказівники на функції

Функція під час виконання програми займає певну ділянку пам'яті, тобто має адресу.

- ✓ Ім'я функції – це *адресний вираз*, значенням якого є адреса функції.

На функцію можна встановити вказівник, тип якого відповідає прототипу функції. Наприклад, якщо функція f має прототип int f(int), то вказівник pf на функції з цим прототипом оголошується так:

```
int (*pf)(int);
```

Дужки навколо *pf обов'язкові. Пріоритет префіксного оператора * нижче, ніж пріоритет дужок (), тому оголошення без дужок

`int*pf(int);` є прототипом функції, яка має ім'я `pf` і повертає вказівник на `int`.

Установити вказівник на функцію можна за допомогою її імені, наприклад `pf=f`. Після цього ім'я `pf` позначає функцію так само, як і `f`, тому, наприклад, вираз `pf(10)` є викликом функції `f` з аргументом `10`.

✓ Параметр функції може зображувати функції. Для цього параметр оголошують як *вказівник на функції* з відповідним заголовком.

Приклад. Табуляція функції полягає в тім, що виводяться її значення в певних точках. Припустимо, що точки – це цілі числа з деякого діапазону, а функція в цих точках має цілі значення. Потрібно пройти цілі точки діапазону й вивести значення функції в них.

Опишемо табуляцію у функції `fTab`. Дії з табуляції не залежать від того, яка саме функція табулюється, тому цю функцію зображує параметр функції `fTab`. За умовою, табульовані функції мають заголовок вигляду `int f(int)`, інші два параметри визначають діапазон точок. Отже, функція `fTab` може мати такий прототип:

```
void fTab(int (*f)(int), int, int);
```

У тілі функції `fTab` табульована функція викликається за допомогою параметра `f`.

Розглянемо програму з двома викликами функції `fTab`, у яких аргументи вказують на функції $f_1(x)=x^2+1$ і $f_2(x)=2x$. Функції табулюються в цілих точках від 0 до 8. Аргументом, що у виклику відповідає параметру-вказівнику на функції, може бути як ім'я функції, так і ім'я вказівника, установленого на неї.

```
void fTab(int (*f)(int), int low, int up);
int f1(int), f2(int);
int main()
{ int (* pf)(int) = f1;  // pf - синонім f1
  fTab(pf,0,8);          // табуляція f1
  fTab(f2,0,8);          // табуляція f2
  return 0;
}
// функція табуляції
void fTab(int (*f)(int), int low, int up)
{ for(int i=low; i<=up; i++)
    cout << i << ':' << f(i) << "  ";
  cout << endl;
}
```

```
// табульовані функції
int f1(int x) { return x*x+1; }
int f2(int x) { return 2*x; }
```

У наведеному прикладі можна оголосити *ім'я типу вказівників на функції*. Оголошення імені `PFII` типу вказівників на функції з прототипом вигляду `int f(int)` виглядало б так:

```
typedef int (*PFII)(int);
```

Відповідно прототип функції табулювання був би таким:

```
void fTab(PFII f, int low, int up);
```

Решта програми не змінилася б.

Вправи

1.31. Що виводиться за програмою?

```
#include <iostream>
using namespace std;
int main(){
  int i, a = 66*256*256*256+50*256*256+65*256+49;
  char * p = (char *)&a;
  for (i=0; i<4; cout << p[i++]);
  cout << endl;
  return 0;
}
```

1.32. Якщо функція $f(x)$ визначена на відрізку $[a; b]$, неперервна на проміжку $(a; b)$ і набуває на кінцях відрізка значення різних знаків, то рівняння вигляду $f(x) = 0$ має хоча б один корінь на $[a; b]$. Метод половинного поділу дозволяє наближено обчислити значення x, для якого $|f(x)| < \varepsilon$ (у деяких випадках воно буде наближенням кореня рівняння, але не в усіх). Суть методу полягає в тім, що обчислюється середина m відрізка $[a; b]$ і за умови $f(m) \neq 0$ пошук продовжується у такий самий спосіб на тому з відрізків $[a; m]$ і $[m; b]$, на кінцях якого функція набуває значення різних знаків. Пошук припиняється, якщо на якомусь кроці $f(m) = 0$ або довжина відрізка стає меншою за задану межу. Написати функцію, яка має параметр, що зображує функцію дійсного аргументу з дійсними значеннями, і реалізує метод половинного поділу. Написати програму, що розв'язує задачу для $\sin x - c = 0$, де $-1 < c < 1$, на відрізку $[-\pi/2; \pi/2]$.

1.33. Для того, щоб наближено обчислити інтеграл функції, визначеної на відрізку $[a; b]$, можна застосувати метод

трапецій. Відрізок $[a; b]$ розбивають на відрізки вигляду $[a+i\times h; a+(i+1)\times h]$ і обчислюють суму площ трапецій, утворених точками $a+i\times h$, $f(a+i\times h)$, $f(a+(i+1)\times h)$, $a+(i+1)\times h$. При цьому крок h вибирають так, що $a+m\times h = b$ за певного $m > 0$. Інтеграл можна обчислити шляхом подвоєння точності. Спочатку вибирають $m = 1$, тобто $h = b-a$, і обчислюють інтеграл. Далі на кожному наступному етапі крок h зменшують удвічі й обчислюють інтеграл. Етапи повторюють, поки значення інтеграла, одержані на двох суміжних етапах, відрізняються більше ніж на деяке задане ε. Написати функцію, яка має параметр, що зображує функцію дійсного аргументу з дійсними значеннями, і обчислює інтеграл методом трапецій з подвоєнням точності. Написати програму наближеного обчислення інтеграла функції $\sin x$ на відрізку $[0; \pi]$ (його точним значенням є 2).

Контрольні запитання

1.1. Що таке масив? Опишіть вигляд означення масиву.
1.2. Які вирази забезпечують доступ до окремого елемента масиву?
1.3. У чому полягає прямий доступ до елементів?
1.4. Чим з погляду математики є значення масиву?
1.5. Що таке адреса змінної?
1.6. За допомогою якої операції можна задати адресу змінної?
1.7. Що таке типізований вказівник?
1.8. Який вираз позначає тип адрес даних певного типу?
1.9. Чи залежать розміри типізованих вказівників від їх типів?
1.10. Що таке встановлення вказівника на змінну?
1.11. Що таке розіменування вказівника? Як воно позначається?
1.12. Як виглядає оголошення імені типу типізованих вказівників?
1.13. Які арифметичні дії можливі з адресами?
1.14. Яке значення має вираз, що є ім'ям масиву?
1.15. Як елементи масиву масивів зберігаються в пам'яті?
1.16. Чи допустима в мові C++ інструкція присвоювання, яка ліворуч містить лише ім'я масиву? Чому?
1.17. Чи завжди можна промоделювати масивом послідовність, довжина якої заздалегідь невідома?

РОЗДІЛ 2.
Рядки

З погляду математики рядок – це послідовність символів. У мові C++ є кілька способів зображення рядків. Далі розглянемо два з них: С-рядки та тип `string`.

2.1. С-рядки

Найпростіша структура даних для зберігання послідовності символів – масив символів. У мові С, від якої походить мова C++, рядок зображується масивом символів, який містить послідовні символи й маркер кінця рядка – символ `'\0'` (нуль-символ).

> **С-рядок** – це зображення послідовності символів у масиві символів з доданим маркером кінця `'\0'`.

Значенням С-рядка (англ. – C-string value) є послідовність символів від його початку, обмежена найближчим `'\0'`. Під час виконання програми С-рядок, будучи масивом символів, ідентифікується *адресним виразом*, що задає адресу даних типу `char`. Послідовність символів, розташовану, починаючи з адреси, заданої адресним виразом, називатимемо **рядковим значенням**, або просто **рядком**.

У мові C++ для роботи із С-рядками є спеціальні засоби, реалізовані як у самій мові, так і бібліотечні. Проте сучасні реалізації мови C++ надають інший спосіб зображення рядків, набагато зручніший і безпечніший, ніж С-рядки (див. підрозд. 2.2). Тому розглянемо лише деякі мовні особливості, зв'язані із С-рядками, і кілька бібліотечних функцій.

Ініціалізація рядковою константою

Рядкова константа, наприклад `"abc"`, зображується в пам'яті як С-рядок, тобто масивом із чотирьох (а не трьох!) символів. До байтів із символами `'a'`, `'b'`, `'c'` *додається ще один* – з нуль-символом `'\0'`. У пам'яті програми константа `"abc"` задається *адресою першого байта* цієї послідовності байтів. Звідси рядковою константою можна ініціалізувати масив символів або присвоїти чи ініціалізувати нею вказівник типу `char*`.

Приклад. Ініціалізуємо вказівник на символи й масив символів рядковими константами.

```
char *p = "123"; char s[6] = "abc";
```

Вказівник `p` встановлено на перший із чотирьох послідовних байтів, що містять символи `'1'`, `'2'`, `'3'`, `'\0'`. Елементи масиву `s` від `s[0]` до `s[2]` мають значення `'a'`, `'b'`, `'c'`, а від `s[3]` до `s[5]` – `'\0'`. ◂

Ініціалізація масиву символів рядковою константою та присвоювання цієї константи вказівнику типу `char*` мають, як мінімум, одну відмінність. Обидві рядкові константи `"123"` та `"abc"` з прикладу задають *незмінювані послідовності символів* типу `char const[4]`. Проте інструкція

```
char s[6] = "abc";
```

створює *змінну* `s` (масив із 6 байтів) та ініціалізує його значеннями `'1'`, `'2'`, `'3'`, `'\0'`. Елементи масиву є *символьними змінними, які можна змінювати*. Наприклад, значення масиву `s` можна зробити порожнім рядком, присвоївши `s[0]='\0'`. Водночас інструкція

```
char *p = "123";
```

створює змінну-вказівник і встановлює його на перший байт С-рядка `"123"`. Вказівник `p` насправді вказує на послідовність *незмінюваних* символів. Спроба під час виконання програми змінити символ у межах цієї послідовності буде проігнорована або призведе до аварійного закінчення – залежно від версії та настройок компілятора.

Якщо розмір масиву в ініціалізації рядковою константою не вказано, то з додатковим `'\0'` він буде на 1 більше ніж довжина константи. Наприклад, після оголошення

```
char a[]="abc";
```

маємо `sizeof(a)`=4. Водночас ініціалізація

```
char a[]={'a','b','c'};
```

означає масив із трьох елементів, в якому немає `'\0'` (і це є *потенційно небезпечним*).

- ✓ Якщо масив символів має розмір *N*, то довжина ініціалізуючого С-рядка повинна бути не більше *N*–1, інакше компілятор повідомить про помилку.
- ✓ Якщо всі елементи масиву символів мають значення, відмінні від `'\0'`, то значення С-рядка, зображеного в масиві, закінчується *невідомо де*, і це може мати *непередбачувані наслідки*.

Операція `cout<<s`, де `s` – ім'я масиву, тип якого не є символьним, виводить значення вказівника `s`, тобто деяку адресу. Проте, якщо `s` – масив символів, то операція виводить його рядкове значення. Узагалі, якщо *AE* – адресний вираз типу `char*`, то операція вигляду `cout<<`*AE* виводить рядкове значення С-рядка – від байта, на який указує адресний вираз, до байта перед найближчим символом `'\0'`. Наприклад, якщо вказівник `p` встановлено на рядкову константу `"12345"`, то вираз

```
cout << p << '::' << p+2
```

виводить символи `12345::345`.

Під час уведення в масив символів, тобто в *змінну*, наприклад `s`, операція вставлення з потоку `cin>>s`, як і для інших типів, пропускає порожні символи (пропуск, табуляція, кінець рядка), переписує в масив `s` непорожні до найближчого порожнього й дописує до них символ `'\0'`.

✓ Кількість уведених символів *не контролюється*, тому, якщо заповнюються байти за межами масиву `s`, можливі *непередбачувані наслідки*.

Приклад. Нехай діє оголошення `char s1[10];`. Під час виконання `cin>>s1` у вхідному потоці пропускаються порожні символи, потім непорожні записуються в масив до появи порожнього. Якщо цих символів більше дев'яти, то символ-завершувач дописується вже за межами масиву `s1`, і це може бути небезпечним. ◂

✓ Оголошуючи масив символів, необхідно забезпечити, щоб його розмір був достатнім для обробки можливих вхідних даних.

Узагалі, якщо *AE* – адресний вираз типу `char*`, що вказує на *змінювані дані*, то операція вигляду `cin>>`*AE* записує непорожні символи в послідовні байти, починаючи з байта, на який указує вираз *AE*. *І це теж може бути небезпечним.*

Спроба ввести символи в незмінюваний рядок, на який установлено вказівник, призводить до аварійного закінчення, адже *не можна змінити незмінювані дані*. Наприклад, таким чином:

```
char *s="123456789";
cin>>s; // помилка під час виконання
```

Параметри головної функції

Запустити програму на виконання можна не тільки засобами системи програмування або файлового менеджера. Будь-яка операційна система дозволяє запустити програму в **командному рядку**, або **рядку виклику**.

У рядку виклику вказується шлях до файлу з програмою та, можливо, ще кілька слів (послідовностей непорожніх символів). Операційна система зчитує цей рядок, визначає кількість *n* слів у ньому й створює масив з *n*+1 вказівників на символи. Вона також послідовно записує слова з рядка виклику з обмежувачами `'\0'` у масив символів і встановлює вказівники на початки цих слів. Вказівник з індексом *n* отримує нульове значення, позначене в мові C++ константою `NULL`. Слова з рядку виклику програми називають **аргументами виклику**.

У заголовку функції `main()` можна записати два параметри: перший типу `int`, другий типу `char**` або `char*[]` (тобто *вказівник на вказівники на символи*). Тоді на початку виконання головної функції значеннями параметрів стають, відповідно, кількість слів у рядку виклику й адреса першого елемента в масиві вказівників на рядки. За традицією, параметри програми мають імена `argc` (типу `int`) і `argv` (типу `char**`), як на рис. 2.1.

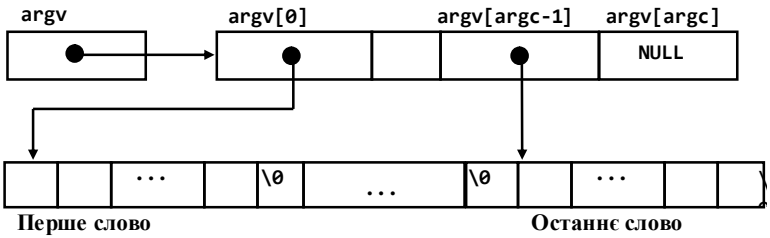

Рис. 2.1. Зберігання параметрів

Приклад. Програма виводить на екран слова, отримані з командного рядка. Слова є значеннями С-рядків, які позначаються виразами `argv[0]`, `argv[1]`, …, `argv[argc-1]`.

```
#include <iostream>
usung std::cout;
int main(int argc, char **argv)
{ for (int i=0; i<argc; ++i)
    cout << argv[i] << '\n';
  return 0;
}
```

Якщо запустити цю програму із системи програмування або файлового менеджера, то виходом буде лише ім'я виконуваного файлу (зі шляхом до нього у файловій системі). ◂

Вправи

2.1. Що буде виведено за програмою?

```
#include <iostream>
using namespace std;
void printSuffs(char * p){
  while(*p) cout << p++ << ' ';
  cout << endl;
}
int main(){
  printSuffs("gambit");
  system("pause");
  return 0;
}
```

2.2. Написати програму, яка перевіряє кількість аргументів виклику й друкує повідомлення `Too many`, якщо аргументів більше ніж 3; повідомлення `Not enough`, якщо їх менше ніж 3; якщо ж їх 3, то повідомлення `OK` і самі аргументи.

2.2. Тип string

У мові C++ для зображення рядків є стандартний бібліотечний тип `string`. Для роботи з ним необхідно включити файл `<string>` і скористатися простором імен `std`. Тип `string` насправді є *класом*, а змінні цього типу – *об'єктами* (див. розд. 3).

Аналогічно С-рядкам, рядкове значення змінної типу `string` – це зображена нею послідовність символів.

Змінні типу `string` можна ініціалізувати за допомогою рядкових констант, С-рядків, ідентифікованих іменем масиву символів або вказівником на символи, та інших змінних типу `string`. Наприклад, інструкції

```
char cs[100] = "123";
string s0, s1("ABC"), s2(s1), s3 = cs;
```

надають неініціалізованій змінній `s0` як значення порожній рядок, змінним `s1` і `s2` – рядок `ABC`, змінній `s3` – рядок `123`.

✓ Довжина рядкового значення змінних типу `string` визначається не за символом `'\0'`, а зберігається окремо в пам'яті,

тому, на відміну від С-рядків, символ '\0' може входити в рядкове значення змінної типу string.

2.3. Операції з рядками

Розглянемо кілька типових операцій, специфічних для рядків.

Визначення довжини рядка, тобто кількості символів рядка. Наприклад, довжина рядка ABC дорівнює 3, порожнього рядка – 0.

Конкатенація рядків. Конкатенація двох рядків – це результат дописування другого рядка в кінці першого. Наприклад, конкатенацією рядків ABC і 12 є ABC12.

Лексикографічне порівняння рядків. Це порівняння рядків, засноване на порядку символів у ASCII-таблиці й аналогічне звичайним словникам. Наприклад, рядок bca менше ніж cw або A, але більше ніж abcd та bc.

До рядків застосовні операції пошуку входження символу чи підрядка в рядок, а також копіювання рядкового значення, уведення з потоку >>, виведення в потік <<.

Зазначені операції реалізовані в мові C++ як для С-рядків, так і для рядків типу string. Далі розглянемо лише деякі бібліотечні засоби обробки рядків. Детальнішу інформацію шукайте в довідкових системах середовищ програмування та інших джерелах.

Обробка С-рядків

Функції обробки С-рядків оголошено в бібліотечному файлі <string.h> (включати потрібно файл <cstring>). Розглянемо лише деякі з них.

Функція із заголовком

```
int strlen(char * s)
```

повертає ціле число – довжину рядкового значення s.

Функція із заголовком

```
char* strcat(char* s1, const char* s2)
```

реалізує дописування рядків: рядкове значення s2 дописується в кінець С-рядка s1, адресою початку результуючого С-рядка є s1, який і повертається функцією. Ця функція небезпечна, оскільки не контролює, чи достатньо ділянки пам'яті, адресованої s1, щоб умістити результат конкатенації. Тому в деяких версіях мови C++ замість неї рекомендовано використовувати безпечніший аналог strcat_s.

Функція із заголовком

```
int strcmp(const char* s1, const char* s2)
```

порівнює рядкові значення лексикографічно й повертає ціле значення – від'ємне, якщо `s1` менше, ніж `s2`; 0, якщо вони рівні, і додатне, якщо `s1` більше, ніж `s2`. Функція `strncmp(s1,s2,n)` робить те саме, тільки порівнює не більше ніж `n` перших символів рядків. Крім звичайного лексикографічного порівняння, у бібліотеці наявні функції, що забезпечують порівняння без урахування регістра символів та/або з урахуванням регіональної (локальної) специфіки.

Функція із заголовком

```
char* strcpy(char* s1, const char* s2)
```

записує вміст С-рядка `s2` на місце С-рядка `s1` і повертає адресу початку останнього, тобто `s1`. Аналогічно функції `strcat`, ця функція теж потенційно небезпечна. Її безпечніший аналог – функція `strcpy_s`. Зазначимо, що звичайне присвоювання С-рядків, заданих вказівниками, не копіює їх вміст і може мати непередбачувані наслідки (наприклад, помилки в обробці динамічних даних – див. розд. 4).

Операції введення з потоку `>>` і виведення в потік `<<` для С-рядків було розглянуто в підрозд. 2.1.

Обробка даних типу string

Довжину рядкового значення змінної типу `string` повертає функція `length()`. Наприклад, після означення рядків

```
string s0, s1("ABC");
```

вираз `s0.length()` має значення 0; `s1.length()` – значення 3.

- ✓ Якщо тип є класом, то виклик функції, означеної для типу, який застосовується до змінної-об'єкта, записується після імені змінної через крапку.

Для типу `string` означено операції присвоювання `=`, конкатенації `+`, дописування `+=`, порівнянь `==`, `!=`, `<`, `<=`, `>`, `>=`, уведення з потоку `>>`, виведення в потік `<<`, індексації `[]` та інші. Розглянемо деякі з них.

Присвоювання реалізує операцію копіювання рядків. Змінній типу `string` можна присвоїти або іншу змінну цього типу, або С-рядок, або символ. Наприклад, для змінних, означених вище, можливі вирази присвоювання `s0=s1`, `s0=cs`, `s0="QWE"`, `s0='A'`. Перші три присвоювання копіюють у змінну `s0` рядкові значення

виразів праворуч, і вони стають її рядковими значеннями, останнє робить рядковим значенням змінної `s0` рядок з одного символу.

Операція конкатенації `+` застосовна до змінних типу `string`, С-рядків і символьних значень; хоча б один із двох її операндів має бути об'єктом типу `string`. Наприклад, після ініціалізації змінних

```
string s1="ABC", s2="12"; char cs[100]="UVWX";
```

вираз `s1+s2` має рядкове значення `ABC12`, вираз `s1+cs` – `ABCUVWX`, `cs+s1` – `UVWXABC`, `cs[0]+s2+"MN"` – `U12MN`. Аналогічним є **дописування**: після `s1+=s2` об'єкт `s1` має рядкове значення `ABC12`, а після `s2+=cs[0]` об'єкт `s2` має рядкове значення `12U`.

Операції порівняння застосовуються до об'єктів типу `string` або С-рядків; хоча б один з операндів має бути об'єктом типу `string`. Результатом є лексикографічне порівняння відповідних рядкових значень. Наприклад, після ініціалізації

```
string s1="AB", s2="ABC"; char cs[100]="a";
```

вирази `s1<s2` і `s2<=cs` мають значення "істина", а вираз `s1+s2>cs` – "хибність".

Операції введення та виведення. Операція введення з потоку `>>` у змінну типу `string` пропускає в потоці порожні символи та створює рядкове значення з послідовності непорожніх символів до найближчого порожнього або до кінця потоку. Кількість символів, що потрапляють у рядок, *практично необмежена*.

Функція `getline` (аргументами в її виклику є потік і об'єкт-рядок) уводить із потоку в рядок усі символи до найближчого символу кінця рядка. Доступним у потоці стає символ, наступний після цього кінця рядка.

Операція `<<` виводить у потік рядкове значення виразу типу `string`.

Приклад. Якщо до кінця рядка потік `cin` містить символи `1 2 3`, то за інструкціями

```
cin >> s1; getline(cin,s2);
```

об'єкт-рядок `s1` отримує рядкове значення `1`, `s2` – `□2□3` (тут символ □ позначає пропуск). Після цього інструкція

```
cout << s1+s2+' '+s1;
```

виводить у потік `cout` послідовність символів `1□2□3□1`. ◂

Вираз вигляду `s[i]` з **операцією індексації** позначає символьну змінну (елемент об'єкта-рядка `s`) або її значення. Наприклад, після присвоювання `s2="012"` вираз `s2[0]` має значення `'0'`, а присвоювання `s2[2]='X'` надає `s2` значення `01X`.

✓ Програміст має стежити, щоб у виразі вигляду `s[i]` значення `i` було в межах від `0` до `s.length()-1`, інакше можливі непередбачувані наслідки.

Порівняно із С-рядками клас `string` має суттєві переваги:

– це *повноцінний тип*, який дозволяє записувати вирази типу `string` (зокрема повертати їх із функцій);

– оператори мови C++ (`=`, `+`, `>>` тощо), означені для цього класу, забезпечують *зручний запис операцій* з рядками;

– операції з рядками, означені для класу `string`, *убезпечують від "залізання" в інші змінні*.

Тип `string` має й певні недоліки, але їх розгляд виходить за межі цього посібника.

Вправи

2.3. Написати власні варіанти функцій: а) `strlen`; б) `strcmp` і `strncmp`; в) `strchr` і `strrchr`; г) `strcpy` і `strncpy`; д) `strcat` і `strncat`; е) `strstr`; ж) `strpbrk`; з) `strspn` і `strcspn`; и) `strcpy_s`.

2.4. Написати програму, яка виводить розмір змінних типу `string`, ініціалізованих рядковими константами `""`, `"0"`, `"01234567"`, `"0123456789012345678 90"`.

2.5. Написати функцію, що відсікає всі пропуски в кінці рядка[3].

2.6. Написати функцію, що дзеркально перевертає рядок.

2.7. Написати функцію визначення, чи є рядок *паліндромом*, тобто симетричною послідовністю символів.

2.8. Написати функцію визначення, чи є рядок подвоєним рядком, тобто має вигляд *ww*, де *w* – деяка послідовність символів, можливо, порожня.

2.9. Рядок містить слова, відокремлені одне від одного проміжками (у довільній кількості). Написати функцію, яка:
а) визначає кількість слів у рядку;

[3] У цій і наступних задачах можна скористатися як С-рядками, так і об'єктами типу `string`.

б) визначає довжину найдовшого слова рядка;
в) дзеркально перевертає кожне слово рядка;
г) вилучає з рядка всі слова, що містять менше п'яти літер.

2.10. Написати функцію, яка визначає множину символів, наявних у рядку.

2.11. Написати функцію, яка визначає множину символів, що входять у рядок рівно по одному разу.

2.12. Написати функцію, яка за двома рядками визначає:
а) чи збігаються множини символів, наявних у заданих рядках;
б) чи є множина символів першого рядка підмножиною множини символів другого.

Контрольні запитання

2.1. У чому полягає особливість зберігання послідовності символів, заданих рядковою константою?

2.2. Що таке С-рядок?

2.3. У чому полягає відмінність ініціалізації масиву символів рядковою константою й послідовністю символьних констант у фігурних дужках?

2.4. У чому полягає відмінність ініціалізації масиву символів рядковою константою та присвоювання цієї константи вказівнику типу `char*`?

2.5. Чи може головна функція мати параметри? Якщо так, то які?

2.6. Опишіть, як параметри функції `main` отримують значення.

2.7. Які оператори мови C++ застосовні до операндів типу `string`?

2.8. Чи можливе присвоювання С-рядків? Чи можливе присвоювання об'єктів типу `string`?

2.9. Чи може рядкове значення С-рядка містити символ `'\0'`?

2.10. Чи може рядкове значення об'єкта типу `string` містити символ `'\0'`?

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.

3.4. Написати клас точок площини `Point` із такими методами: уведення, виведення, порівняння, обчислення відстані між двома точками. Написати програму, яка за трьома точками площини обчислює периметр утвореного ними трикутника, його площу, радіуси вписаного й описаного кіл.

3.5. Написати клас прямих ліній на площині, заданих коефіцієнтами рівняння $Ax+By+C=0$ (коефіцієнти A та B не можуть одночасно бути нульовими). Методи описують дії: *ввести* й *вивести* дані про пряму; визначити, *чи збігаються* й *чи перетинаються* дві прямі (два окремі методи); *утворити* пряму за двома точками, через які вона проходить; обчислити *відстань* від прямої до точки; визначити, як розташовано дві точки відносно прямої (по один бік; по різні боки; одна на прямій, інша ні; обидві на прямій). Написати програму, яка вводить чотири точки й визначає, як розташовано четверту з них відносно трикутника, утвореного першими трьома (усередині, на контурі, зовні).

3.3. Конструктори й деструктор

Стандартні конструктори й деструктор

Конструктор – це метод класу, що неявно викликається для ініціалізації об'єкта відразу після його створення.

Деструктор – це метод класу, що неявно викликається для підготовки об'єкта до знищення.

Компілятор, створюючи машинний варіант класу, додає до методів, оголошених у класі, ще чотири – стандартний конструктор, стандартний конструктор копії, метод присвоювання та стандартний деструктор.

Стандартний конструктор жодних значень атрибутам об'єкта не надає. Якщо створюється статичний об'єкт (означений за межами функцій програми або зі специфікатором `static`), то всі його атрибути мають нульові значення. У нестатичних об'єктів значеннями атрибутів є "випадкове сміття".

Стандартний конструктор викликається *неявно*, коли виконується інструкція означення об'єкта без його ініціалізації, наприклад інструкція `Body p;`.

Стандартний конструктор копії копіює значення всіх атрибутів уже існуючого об'єкта в створюваний.

Конструктор копії викликається в таких ситуаціях:

1. Оголошений об'єкт ініціалізується за допомогою іншого об'єкта, наприклад `Body p2=p1, p3(p1);`.

2. Параметр-значення функції є змінною, яка після створення ініціалізується значенням аргументу, указаного у виклику. Якщо параметр є об'єктом, то цю ініціалізацію здійснює конструктор копії.

Ще одна ситуація, в якій викликається один з конструкторів, – створення динамічного об'єкта (див. розд. 4).

✓ Для кожного об'єкта один із конструкторів викликається тільки один раз – *коли створюється об'єкт*. Коли ж виконується присвоювання об'єктові, яке не є ініціалізацію, конструктор не викликається.

Стандартний деструктор викликається перед тим, як об'єкт знищується. Це відбувається, коли закінчується виклик функції, а об'єкт є локальним у ній, або коли закінчується виконання програми, а об'єкт є статичним. Ще одна ситуація – знищення динамічного об'єкта (див. розд. 4).

Запис власних конструкторів

У класі можна означати власні конструктори й описувати в них ініціалізацію атрибутів об'єкта. Дії, описані в конструкторі, виконуються відразу після створення об'єкта.

✓ Конструктор оголошується *як відкритий метод* класу. Його іменем є *ім'я класу*, а в заголовку немає типу значень, що повертаються. У класі можна означити кілька конструкторів з різними заголовками.

Приклад. Створений раніше клас `Body` забезпечував цілісність об'єктів при виконанні операцій над ними, але за наявності тільки стандартного конструктора щойно створені тіла могли містити неприпустимі дані, тобто порушувалася цілісність даних. Виправимо це, додавши до класу `Body`, означеного вище, чотири конструктори, що забезпечують різні форми ініціалізації об'єктів-тіл. Тепер клас `Body` відповідатиме принципу інкапсуляції, гарантуючи, що його об'єкти-тіла можуть мати невід'ємний вік і масу в межах від 1 до 100.

```
const int MMAX=100;
class Body {
private: int  mass, age;
```

```
public :
  Body();              /*1*/
  Body(int m);         /*2*/
  Body(int m,int a);   /*3*/
  Body(const Body&);   /*4*/
  ...
};
```

Перший конструктор дозволяє оголосити об'єкт без ініціалізації – як у інструкції вигляду `Body b;`, другий – ініціалізувати об'єкт цілим аргументом, наприклад `Body b(30);`, третій – двома цілими аргументами: `Body b(45,2);`. Четвертий є конструктором копії й дозволяє ініціалізувати об'єкт за допомогою іншого: `Body b1=b; Body b2(b1);`.

Реалізуємо наведені конструктори. У першому з них тіло отримує масу `MMAX/2` та вік 0.

```
Body::Body(): mass(MMAX/2), age(0){}      /*1*/
```

У другому конструкторі поле `mass` ініціалізується цілим значенням аргументу, а поле `age` – значенням 0. У тілі, якщо встановлена маса є недопустимою, то вона змінюється.

```
Body::Body(int m): mass(m), age(0){       /*2*/
  if( !(0<m && m<=MMAX) )
    mass=MMAX/2;
}
```

У третьому конструкторі цілі значення аргументів ініціалізують обидва поля, а в тілі, можливо, змінюються недопустимі значення.

```
Body::Body(int m,int a): mass(m), age(a){ /*3*/
  if( !(0<m && m<=MMAX) ) mass=MMAX/2;
  if(a<0)age=0;
}
```

Нарешті, четвертий конструктор *копіює* в створюваний об'єкт значення полів об'єкта, зображеного параметром-посиланням. Цей метод виконує ті самі дії, що й стандартний конструктор копії, тому тут наводиться лише для ілюстрації.

```
Body::Body(const Body& b2) :              /*4*/
       mass(b2.mass), age(b2.age) {};
```

Ініціалізацію всіх або деяких атрибутів задає *секція ініціалізації* в конструкторах (після двокрапки : через кому).

> Ім'я атрибута й ініціалізуючий вираз у дужках утворюють **ініціалізатор**.

Для ілюстрації в головній функції нижче оголошено чотири об'єкти-тіла. Під час створення об'єктів p1–p3 виконуються, відповідно, наведені конструктори, а об'єкта p4 – стандартний конструктор копії.

```
int main()
{ Body p1, p2(20), p3(33,-3), p4(p2);
  cout <<   "p1:"; p1.out();
  cout << ",\np2:"; p2.out();
  cout << ",\np3:"; p3.out();
  cout << ",\np4:"; p4.out();
  ...
}
```

За цими інструкціями програма виводить такий текст:

```
p1: Age:0 Mass:50,
p2: Age:0 Mass:20,
p3: Age:0 Mass:33,
p4: Age:0 Mass:20
```

◀

Замість перших трьох конструкторів у класі можна записати тільки один з таким заголовком:

```
Body(int m=MMAX/2,int a=0);
```

Якщо в класі оголошено цей конструктор, то в означенні

```
Body b1(15,3), b2(15), b3;
```

він насправді викликається з аргументами, відповідно, 15 і 3, 15 і 0, 50 (це значення MMAX/2) і 0. При цьому за межами класу має бути означено конструктор /*3*/, наведений вище.

Вирази, якими в заголовку функції ініціалізовано параметри, називаються **стандартними аргументами** (default arguments). Якщо у виклику функції не вказано аргументи, відповідні до цих параметрів, то їм надаються значення стандартних аргументів. Параметри зі стандартними аргументами можуть означатися *тільки в кінці переліку* параметрів функції.

Наявність конструкторів у класі Body дозволяє вилучити з класу метод init(). Цей метод має недолік – його можна за-

стосувати до об'єкта скільки завгодно разів, що не відповідає його призначенню. На відміну від нього, якийсь із конструкторів гарантовано викликається точно один раз відразу після створення об'єкта.

Ініціалізація в конструкторі дозволяє надати початкові значення атрибутам *незмінюваного* об'єкта. Окрім того, ініціалізація відбувається гарантовано *на початку виконання* конструктора, що може бути суттєвим для ініціалізації об'єктів з полями-об'єктами та в інших ситуаціях (див. розд. 5). Незалежно від порядку запису ініціалізуючих виразів ініціалізація відбувається в порядку запису атрибутів у заголовку конструктора в оголошенні класу.

Створюючи конструктори класу, бажано врахувати всі ситуації передавання даних в об'єкти класу, що *є природними й використовуються часто*. Не варто записувати конструктори класу, що реалізують малоймовірні способи створення об'єктів. Наприклад, конструктор класу `Body` з двома параметрами створює тіло будь-якого віку, що не є цілком природним.

Якщо в класі оголошено хоча б один конструктор, то *стандартний конструктор не створюється*. Звідси, якщо в класі немає конструктора без параметрів, то звичайне оголошення об'єкта, як `Body p;`, стає *помилковим*.

✓ У класі з власними конструкторами не забудьте про конструктор без параметрів.

Конструктор копії, наведений вище, як і стандартний конструктор копії, ініціалізує об'єкт значеннями атрибутів іншого об'єкта. У класі тіл цього достатньо, як і стандартних дій зі знищення об'єктів-точок. Проте в розд. 3.6 клас тіл буде модифіковано так, щоб під час створення й знищення об'єктів усі конструктори й деструктор виконували певні додаткові дії. Отже, розгляд деструктора відкладемо до розд. 3.6.

Вправи

3.6. Написати варіанти конструкторів для класів точок і прямих на площині (див. вправи до розд. 3.1).

3.7. Перевірити конструктори з попередньої вправи у простих програмах.

3.4. Об'єкти у функціях

Параметри-значення й параметри-посилання

Згадаймо: на початку виконання виклику функції для кожного її параметра-значення створюється локальна змінна й ініціалізується значенням аргументу у виклику. Якщо параметр-значення є об'єктом, то його ініціалізує конструктор копії. Коли закінчується виклик функції, об'єкти в пам'яті виклику знищуються, тому до параметра-значення застосовується деструктор.

Якщо об'єкт має величезні розміри й виклики функції виконуються багаторазово, то на ці створення, ініціалізації та знищення йде чимало часу.

Водночас параметр-посилання не є локальною змінною, яка створюється. Під час виконання виклику функції ім'я параметра позначає змінну, указану у виклику. Щоб передати адресу цієї змінної, потрібно набагато менше часу, адже ніякого створення нового об'єкта немає.

✓ Скрізь, де це можливо, для об'єктів використовуйте не параметри-значення, а *параметри-посилання*.

У багатьох ситуаціях параметр-посилання має не змінюватися в тілі функції. Можна явно указати на недопустимість змін, оголосивши параметр із модифікатором `const`. Тоді компілятор повідомить про спроби змінити параметр як про помилки.

Приклад. У класі `Body` був такий метод порівняння мас:

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

У тілі цього методу параметр `b` не змінюється, тому його можна зробити параметром-посиланням. Щоб запобігти його змінам у тілі функції, оголосимо його зі словом `const`. Отже, змінимо метод `compare` (а також його прототип у класі).

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

Під час виконання цього методу, на відміну від попереднього варіанта, локальна змінна, відповідна до імені b, не створюється, конструктор і присвоювання для неї не виконуються. ◂

Повернення об'єктів з функцій

Функція (метод класу або інша) може повертати об'єкт, указаний у тілі функції після слова return. У найпростішому випадку це може бути локальна змінна функції; інші можливості описано в розд. 4.

Приклад. До класу Body додамо метод heavier. Він повертає тіло, маса якого більше, ніж маса тіла – господаря методу, на задану кількість одиниць маси, але не більше ніж максимально можлива. Наведемо лише реалізацію методу.

```
Body Body:: heavier(int dMass)
{ Body temp;              //об'єкт, що повертається
  if(mass+dMass >MMAX)
    temp.mass=MMAX;
  else temp.mass=mass+dMass;
  temp.age=age;
  return temp;
}
```

Об'єкт, що його повертає виклик методу heavier, можна, наприклад, присвоїти, як у таких інструкціях:

```
Body a;
Body b; b=a.heavier(10);
```

Константний метод

Приклад. Об'єкт класу Body, якому належать виклики методів out і compare, під час їх виконання має залишатися без змін.

✓ Щоб явно вказати компілятору на недопустимість змін об'єкта, якому належить виклик методу, у кінці заголовка цього методу додають слово const.

Змінимо заголовки методів out і compare у класі Body.

```
void out() const;
bool compare(const Body& b) const;
```

У реалізації тіла цих методів залишаються без змін. ◂

> Метод, якому заборонено змінювати атрибути об'єкта-власника, називається **константним**.

Оголошення методів константними має задовольняти принцип *"усе або нічого"*. Це означає:

- ✓ усі методи, що викликаються в константному методі, теж мають бути константними.

Вправи

3.8. Змінити методи класів тіл, точок і прямих на площині так, щоб скрізь, де це можливо, їх параметри-об'єкти були параметрами-посиланнями.

3.9. Зробити, де це можливо, методи класів тіл, точок і прямих на площині константними.

3.5. Запис і використання класу

Файл заголовків і файл реалізації

Клас є оголошенням імені, а його інтерфейс – набором заголовків функцій. Для зберігання заголовків функцій мова С, в якій не було поняття класу, надавала спеціальний різновид файлів – файли заголовків.

Файл заголовків зазвичай має розширення імені `".h"` і містить оголошення імен – класів, інших типів, функцій, констант тощо. Найчастіше клас або кілька зв'язаних один з одним класів записують у файл заголовків, а реалізацію методів – в окремий `cpp`-файл. У файлах програми, де присутні об'єкти цих класів, указують включення файлу заголовків для того, щоб ці файли можна було скомпілювати.

Повернемося до класу тіл і запишемо клас `Body` (*без реалізації його методів*) у файл, наприклад `body.h`. Цей клас не містить імен із системних файлів заголовків, тому й директиви включення відсутні.

```
class Body {
  ... // див. вище
};
```

У файл `Body.cpp` запишемо реалізацію методів цього класу. У нього потрібно включити системні файли заголовків `<iostream>`, `<cstdlib>`, `<time.h>` і свій файл `body.h`. Ім'я біб-

ліотечного файлу заголовків записується в дужках < >, ім'я власного файлу – у лапках[4].

```
#include <iostream>
#include <cstdlib>
#include <time.h>
#include "body.h"
// реалізація методів класу тіл
Body::Body(): age(0), mass(MMAX/2){}
...
```

Частину програми з головною функцією запишемо у файл bodyUse.cpp, на початку вказавши включення файлу заголовків body.h.

```
#include <iostream>
#include <cstdlib>
#include <time.h>
#include "body.h"
using namespace std;
int main()
{ srand(unsigned(time(NULL)));
  Body a; a.out();
  cout<< " Press Enter"; cin.get();
  for(int i=1; i<=7; ++i){
    a.run(rand()*MMAX/RAND_MAX);  // бігає
    a.eat(rand()*MMAX/RAND_MAX);  // їсть
    a.out();                      //виведення даних
    cout<< " Press Enter"; cin.get();
  }
  return 0;
}
```

Отже, маємо три файли: body.h, body.cpp, bodyUse.cpp. Далі за допомогою засобів системи програмування створимо *проект* і додамо до нього ці три файли. Маючи відкритий проект, укажемо компіляцію файлу з програмою bodyUse.cpp. Компілятор має створити об'єктні файли з іменами bodyUse і body (з розширенням ".o" або ".obj"), а також кілька допоміжних файлів. Після компонування програма міститься у виконуваному файлі bodyUse.exe.

[4] Від того, записано ім'я в лапках чи дужках, залежить, з якої папки почне пошук відповідного файлу компілятор (детальніше див. документацію).

Означення імені для препроцесора

Включати файл заголовків необхідно в кілька місць програми, але повторне включення є помилкою, про яку повідомляє компілятор. Інтегровані середовища гарантують, що препроцесор додає кожен файл, указаний у директивах включення, тільки один раз. Проте позбутися повторних включень файлів можна за допомогою директив препроцесора незалежно від засобів інтегрованих середовищ.

За директивою вигляду

```
#define ім'я вираз
```

препроцесор має замінити кожну появу імені виразом, указаним у директиві. Незалежно від виразу після виконання цієї директиви *ім'я стає означеним* у препроцесорі. Умовою, що ім'я є означеним (або, навпаки, неозначеним) можна скористатися для керування діями препроцесора.

Директива вигляду `#ifndef ім'я` задає перевірку, чи є ім'я неозначеним[5], і якщо це так, – то обробку подальшого тексту файлу до директиви `#endif`. Якщо ж ім'я означено, то текст до директиви `#endif` пропускається.

Скористаємося описаними засобами у файлі `body.h` із класом тіл, текст якого має такий загальний вигляд:

```
#ifndef BODY_H
#define BODY_H
class Body { ... див. вище };
#endif
```

Виконуючи директиву `#include "body.h"` уперше, препроцесор означає ім'я `BODY_H`, тому повторні виконання цієї директиви не приводять до включення оголошень класу `Body` в програму.

Сучасні компілятори замість конструкцій, наведених вище, дозволяють на початку файлу заголовків записати єдину директиву.

```
#pragma once
```

[5] Тут наведено вигляд директиви для старих компіляторів. Новітні цей синтаксис підтримують, але, крім того, використовують `#if defined` і `#if !defined`.

Вправи

3.10. Написати класи точок і прямих на площині. Пряма на площині задається парою *різних* точок (p_1, p_2), через які вона проходить. Це зображення неявно містить ще й напрямок (*від* p_1 *до* p_2), тобто пари точок (p_1, p_2) і (p_2, p_1) визначають ту саму пряму, але з протилежними напрямками. Операції з прямими: *утворити* пряму за парою точок, *ввести* пряму, *вивести* пряму, *визначити відхилення* (*deviation*) точки від прямої (–1, якщо точка праворуч від напрямку прямої, 1, якщо ліворуч, і 0, якщо точка на прямій), *визначити рівність* двох прямих (без урахування їх напрямків).

3.11. Написати програму, яка вводить чотири точки й з'ясовує, чи утворюють перші три точки трикутник і, якщо так, то визначає, як розташована четверта відносно трикутника (усередині, на контурі, зовні).

3.6. Статичні атрибути й методи

Статичні атрибути

У класі можна оголосити атрибути, що належать не об'єктам класу, а самому класу. Їх спільно обробляють *усі об'єкти класу*, але за межами методів класу вони недоступні.

Статичний атрибут – це атрибут, оголошений зі специфікатором `static`.

Статичні атрибути існують в одному екземплярі незалежно від кількості створених об'єктів класу. Кожен об'єкт класу за допомогою методів класу може обробляти статичні атрибути класу, проте за межами методів статичні атрибути класу *недоступні*.

Оголошення та обробку статичних атрибутів розглянемо на прикладі класу тіл `Body` (див. попередні підрозділи).

По-перше, оголосимо константу `MMAX` як відкриту на початку класу `Body`.

```
class Body {
public: static const int MMAX=100;
...
}
```

Ця зміна в класі не потребує жодних змін у реалізації його методів.

По-друге, до класу Body додамо прихований статичний атрибут bodiesNumber, значенням якого має бути кількість об'єктів-тіл, що існують під час виконання програми.

```
class Body {
public: static const int MMAX=100;
private :
  int age, mass;
  static unsigned int bodiesNumber;
public: ...
};
```

Атрибути MMAX і bodiesNumber доступні в методах класу тіл, але не належать жодному з об'єктів класу.

Статичний атрибут не належить об'єктам класу, тому необхідно ще з'ясувати, *як його створити та ініціалізувати*.

✓ Статичний атрибут означається в програмі як *глобальна* змінна за межами функцій. Статичний атрибут належить класу, тому позначається з ім'ям класу та оператором розв'язання контексту :: попереду.

Статичний атрибут MMAX є незмінюваним, тому його можна ініціалізувати просто в оголошенні. Узагалі, статичні атрибути *ініціалізуються* за межами методів класу. Отже, ініціалізуємо атрибут BodiesNumber значенням 0.

```
unsigned int Body :: bodiesNumber = 0;
int main() {
  ...
}
```

За межами класу доступ до відкритих статичних атрибутів можливий за кваліфікованим ім'ям, наприклад Body::MMAX.

Методи класу можуть змінювати поточні значення статичних атрибутів. Нехай у класі Body кожен конструктор класу тіл збільшує значення bodiesNumber на 1. Укажемо це збільшення в тілі конструкторів, раніше означених у класі.

```
Body::Body(): age(0), mass(MMAX/2)
{ ++bodiesNumber; }
Body::Body(int m): age(0), mass(m)
{ if( !(0<m && m<=MMAX) )
    mass=MMAX/2;
  ++bodiesNumber;
}
```

```
Body::Body(const Body& b2) :
          age(b2.age), mass(b2.mass)
{ ++bodiesNumber; };
```

Зменшувати кількість існуючих об'єктів будемо в **деструкторі** (див. підрозд. 3.3), прототип якого додамо до класу `Body`.

```
~Body();
```

- ✓ Деструктор у класі може бути тільки один, причому без параметрів. Його заголовок – це знак ~, ім'я класу й дужки.
- ✓ Виконання деструктора – це *не знищення* об'єкта, а лише виконання дій, заданих у тілі деструктора.[6]

Деструктор неявно викликається перед знищенням об'єкта (див. підрозд. 3.3). Мова C++ дозволяє застосувати деструктор до об'єкта й явно, наприклад, якщо `p` – ім'я об'єкта класу `Body`: `p.~Body()`. Проте рекомендуємо цього уникати. Отже, у деструкторі зменшимо значення `BodiesNumber`, тільки якщо воно додатне.

```
Body::~Body()
{ if(bodiesNumber>0) --bodiesNumber; }
```

Статичний метод класу

У прикладі з об'єктами-тілами будемо виводити кількість утворених об'єктів-тіл, що є значенням прихованого статичного атрибута `BodiesNumber`. Щоб отримати це значення, до класу `Body` додамо *відкритий метод*, що повертає значення `BodiesNumber` й має такий заголовок:

```
unsigned int getBodiesNumber();
```

Цей метод застосовний до існуючих об'єктів, проте природно викликати його *незалежно від об'єктів у програмі*. Щоб викликати метод класу незалежно від об'єктів, необхідно оголосити його як статичний метод класу.

> **Статичний метод класу** – це метод, оголошений зі специфікатором `static` на початку.

Отже, у класі `Body` оголосимо новий метод як статичний.

```
static unsigned int getBodiesNumber();
```

У реалізації статичного методу за межами класу слово `static` у заголовку не додається.

```
unsigned int Body:: getBodiesNumber()
{ return bodiesNumber; }
```

[6] А також, можливо, певних "внутрішніх" дій, передбачених компілятором.

Виклик статичного методу містить ім'я класу й оператор ::.

```
cout << Body::getBodiesNumber();
```

- ✓ Статичний метод класу може обробляти *тільки статичні атрибути*. Використання нестатичних атрибутів у статичному методі є помилкою, хоча виклик статичного методу може належати як класу, так і об'єкту.

Приклад обробки статичних атрибутів

Проілюструємо обробку статичних полів об'єктів-тіл у програмі, яка створює тіла й виводить їх кількість. Вважатимемо, що клас записано у файлі body.h, а реалізацію його методів і неконстантних статичних атрибутів – у файлі body.cpp.

```
#include <iostream>
#include <cstdlib>
#include <time.h>
#include "body.h"
using namespace std;
void demo()
{ Body b0, b1(10), b2(20);
  cout<<"\nNBodies:"<<Body::.getBodiesNumber();
}
int main()
{ cout<<"NBodies:"<<Body::getBodiesNumber();
  demo();
  cout<<"\nNBodies:"<<Body::getBodiesNumber();
  return 0;
}
```

Під час виконання цієї програми відбувається ініціалізація статичних атрибутів, записана в файлі body.cpp, і статичний атрибут bodiesNumber ініціалізується значенням 0, яке потім виводиться за допомогою методу getBodiesNumber, застосовного до класу. Далі у виклику функції demo() конструктори збільшують статичний атрибут до значення 3. Коли виклик demo() закінчується, до трьох його локальних об'єктів застосовується деструктор, який зменшує лише додатні значення, тому остаточним значенням атрибута буде 0. Отже, виходом цієї програми будуть такі рядки:

```
NBodies:0
DemoNBodies:3
NBodies:0
```

3.7. Знайомство із означенням операторів

Операції з об'єктами класів можна позначати не лише викликами функцій, а й знаками операцій (*операторами*), наявними в мові C++, як, наприклад, у виразах `x+y` або `a=b-c`, де *операндами є об'єкти*, а не змінні або константи базових типів. Для цього в класі необхідно *означити відповідний оператор*.

У цьому підрозділі наведено загальні відомості про означення операторів у класах. Особливості та "підводне каміння" означень деяких операторів виходять за рамки цього посібника.

- ✓ Означити можна будь-який одномісний або двомісний оператор мови C++, окрім оператора `"."` (крапка) і `"::"` (уточнення області видимості). Тримісний оператор `"?:"` і оператори, яких немає в мові C++, *означати не можна*.

Означаючи оператори, бажано зберігати їх сутність у мові C++, наприклад, щоб оператор `+` позначав дії, пов'язані з додаванням, `=` – з присвоюванням, а `>>` – з уведенням з потоку. Старшинство та кількість операндів залишаються такими самими, як у C++.

Оголошення оператора в класі зазвичай має такий вигляд:

```
тип operator знак (параметри);
```

Тут *тип* – це тип значень, що повертаються, *знак* – оператор мови C++, наприклад `==`, `+`, `=` тощо.

Приклад. Метод, що реалізує оператор порівняння в деякому класі `C`, має такий прототип:

```
bool operator == (const C & obj) const;
```

Першим операндом порівняння є об'єкт, до якого застосовується цей метод, другим – аргумент у виклику, представлений параметром-посиланням `obj`. Порівняння не змінює операнди, тому в заголовку записано специфікатори `const`. Сам метод зазвичай означається за межами класу.

```
bool C :: operator == (const C & obj) const
{ ... залежно від атрибутів класу }
```

Ці означення дозволяють порівняти об'єкти класу, наприклад `x` і `y`, у виразі `x==y`. ◂

У прикладах класів, наведених нижче, буде потрібно означати оператори присвоювання, тому розглянемо схематично два варіанти їх означення. Нехай далі `C` – ім'я класу.

Перший варіант. За правилами мови C++, значенням виразу присвоювання є значення, присвоєне змінній ліворуч від знака `=`. Це дозволяє записувати вирази з ланцюгами присвоювань, наприклад, `a=b=c`, з яких спочатку виконується `b=c`, потім – `a=b`. Звідси метод для операції присвоювання має *повертати об'єкт, якому присвоєно значення*.

Природно вимагати, щоб лівий операнд присвоювання не можна було відразу змінити іншим методом. Тому типом методу присвоювання буде `const C`. У тілі методу об'єкт має змінюватися, тому в кінці заголовка методу *не буде* слова `const`. Отже, до класу `C` додається такий прототип:

```
const C operator = (const C & r);
```

У тілі методу лівий операнд присвоювання ідентифікується й повертається за допомогою вказівника `this`. Присвоювати об'єкт самому собі немає сенсу, тому значення полям лівого операнда присвоюються, тільки якщо він не збігається з правим операндом.

```
const C C::operator = (const C & r)
{ if(this != &r)
    { ... залежить від атрибутів класу }
  return *this;
};
```

Ця функція дозволяє присвоїти одному об'єкту інший, можливо, заданий виразом, наприклад, таким чином:

```
C a, b, c;    // присвоюються:
a=C(...);     // об'єкт, створений конструктором,
c=b=a;        // об'єкт a
```

Другий варіант. Нагадаємо: **посилання на змінну** – це додаткове ім'я певної змінної. Посилання оголошується зі знаком `&` перед іменем, а ім'я змінної, яку надалі позначає посилання, є обов'язковим ініціалізуючим виразом. Наприклад, означимо змінну `a` й посилання `r` на неї.

```
int a = 2;   // a – ім'я змінної
int & r = a; // посилання на змінну a
```

У зоні дії цих оголошень імена `a` і `r` позначають одну й ту саму змінну, тобто, наприклад, `r=4` присвоює значення змінній `a`.

Якщо вираз *T* позначає деякий тип, то вираз *T* `&` позначає **тип посилань** на змінні типу *T*, а вираз `const` *T* `&` – тип посилань на *незмінювані змінні* (типу *T*).

В усіх стандартних і бібліотечних класах оператори присвоювання `=`, `+=` тощо реалізовано методами, які повертають не значення, присвоєне об'єкту, а *посилання на об'єкт* (при цьому кажуть: "повертається об'єкт як змінна"). Отже, метод присвоювання можна оголосити так:

```
const C & operator = (const C & r);
```

Тіло методу залишається без змін.

Контрольні запитання

3.1. Що таке структура? Як виглядає означення типу структури?
3.2. Чи можуть елементи структури мати різні типи?
3.3. Як здійснюється доступ до окремих елементів структури?
3.4. Чим з погляду математики є значення структури як змінної?
3.5. Чи є поля структури рівнодоступними?
3.6. Чи заповнюється неініціалізована структура якими-небудь значеннями?
3.7. У чому полягає присвоювання структур?
3.8. Чи можна значення типу структур повертати з функцій?
3.9. Чи можуть структури мати "поля-функції"?
3.10. У чому полягає відмінність між оголошеннями структур і класів?
3.11. Сформулювати, що таке клас, об'єкт, атрибут, метод, приховане поле, відкрите поле, інтерфейс класу.
3.12. Для чого використовуються зарезервовані слова `public` і `private`? Чи можна використовувати їх в оголошенні типу структур?
3.13. Що таке інкапсуляція й у чому полягає принцип інкапсуляції?
3.14. Чи обов'язково атрибути класу оголошувати прихованими?
3.15. Чи може метод класу бути прихованим?
3.16. Чим виклик методу класу відрізняється від виклику функції, яка не є методом?
3.17. Чи містять об'єкти класу власні копії полів-методів цього класу?
3.18. Що позначає зарезервоване слово `this`?
3.19. Як за допомогою слова `this` позначити об'єкт і його поле?
3.20. У чому полягає стандартне присвоювання об'єктів?
3.21. Чи можуть об'єкти бути параметрами функцій або повертатися з функцій?

3.22. Що таке конструктор? Для чого він призначений?
3.23. Які стандартні конструктори створює компілятор?
3.24. Чи ініціалізує стандартний конструктор поля об'єкта якими-небудь значеннями?
3.25. Коли викликається конструктор без параметрів?
3.26. Чим стандартний конструктор копії відрізняється від стандартного конструктора? Коли він викликається?
3.27. Що таке деструктор і для чого він призначений? Коли він викликається?
3.28. Чим конструктор, оголошений у класі, синтаксично відрізняється від інших методів?
3.29. Чи можна переозначити конструктор?
3.30. Що таке ініціалізатор? Де він записується?
3.31. Чим ініціалізація полів у секції ініціалізації конструктора відрізняється від присвоювання значень цим полям у тілі конструктора?
3.32. Який метод називається константним?
3.33. Які обмеження накладає константний метод на методи, що викликаються в ньому?
3.34. Назвіть синтаксичні особливості конструктора копії.
3.35. Скільки конструкторів копії може бути в класі?
3.36. Назвіть синтаксичні особливості деструктора.
3.37. Скільки деструкторів може бути в класі?
3.38. Чому бажано уникати явних викликів деструктора?
3.39. У чому полягає головна особливість статичного поля класу?

РОЗДІЛ 4.
ДИНАМІЧНІ ДАНІ

4.1. Дані у вільній пам'яті

Вільна пам'ять і динамічні дані

Пам'ять процесу виконання програми складається з кількох частин:

– код програми (містить команди програми й бібліотечних функцій, які викликаються у програмі);

– статична пам'ять (зберігає статичні дані, що існують протягом усього процесу виконання програми);

– автоматична пам'ять, або програмний стек (містить змінні, локальні у викликах функцій);

– вільна пам'ять (містить дані, що створюються під час виконання програми за вказівками останньої).

Розмір статичної пам'яті програми не змінюється протягом виконання програми. Програмний стек має певний, відносно невеликий розмір, у межах якого створюються й знищуються змінні, локальні у викликах функцій. На відміну від них, розмір **вільної пам'яті** обмежений тільки наявною пам'яттю комп'ютера, яку не зайнято іншими програмами. Це дозволяє створювати й обробляти масиви значно більшого розміру, ніж у статичній або автоматичній пам'яті.

Спосіб організації даних у вільній пам'яті називається **купою** й реалізується диспетчером пам'яті операційної системи.

Змінні у вільній пам'яті називаються **динамічними**. Вони не мають імен у програмі й позначаються за допомогою встановлених на них вказівників.

Виділення та звільнення ділянок вільної пам'яті здійснюється шляхом звернень до диспетчера пам'яті у складі операційної системи й називається **керуванням купою**.

Створення й знищення динамічної змінної

Операція new з ім'ям типу як операндом створює динамічну змінну цього типу; значенням операції є адреса змінної.

Приклади

1. Інструкції

```
int *p; p = new int;
```

або ініціалізація

```
int *p = new int;
```

оголошують вказівник на цілі, створюють динамічну змінну цілого типу й установлюють на неї вказівник p. Далі цю змінну ідентифікує вираз *p.

Створювану динамічну змінну можна ініціалізувати значенням виразу в дужках.

```
int *p = new int(123);
```

Значенням змінної *p стає 123.

2. Розглянемо інструкції.

```
int *pi = new int(123);
int **ppi = new int*(pi);
```

Перша створює динамічну цілу змінну зі значенням 123 і встановлює на неї вказівник pi. Друга створює динамічний вказівник, ініціалізує його значенням вказівника pi (адресою змінної *pi) і встановлює на нього вказівник ppi (рис. 4.1). ◂

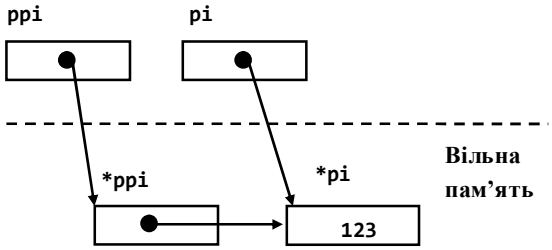

Рис. 4.1. Створення динамічних змінних

✓ Коли створюється динамічна змінна, незайнятої ділянки потрібного розміру може не бути. Тоді виконання операції new призводить до аварійного закінчення програми.

Операція delete, операндом якої є ім'я вказівника, наприклад delete p, звільняє ділянку пам'яті, на яку встановлено вказівник (*знищує динамічну змінну*).

✓ Операція delete p знищує не вказівник p, а змінну, на яку він указує. Після звільнення *вказівник продовжує вказувати* на неї.

Залежно від компілятора спроба доступу до вже знищеної змінної може бути допустимою або помилковою, або аварійно закінчувати програму (як і спроба звільнити вже звільнену змінну).

Приклад. Інструкції, наведені нижче, у деяких системах програмування виконуються, але, починаючи з третього рядка, *є дуже небезпечними*.

```
int *ip = new int (123);
delete ip;   // знищення змінної *ip
*ip=46;      // це небезпечно!
cout << *ip; // виведення 46
delete ip;   // повторне знищення небезпечне!
cout << *ip; // виведення "сміття"
```

◂

✓ Уникайте спроб доступу до звільнених динамічних змінних та їх повторного звільнення.

Клас об'єктів із вказівниками

Кожен клас має два стандартні конструктори: перший дозволяє оголосити об'єкт, другий (конструктор копії) – ініціалізувати об'єкт за допомогою іншого. Перший не надає полям об'єкта жодних значень, другий копіює в створений об'єкт значення полів іншого об'єкта.

Якщо об'єкти містять поля-вказівники, то стандартних конструкторів і стандартного методу присвоювання зазвичай недостатньо. Адже, коли стандартний конструктор створює об'єкт, значенням вказівника в об'єкті є якась випадкова адреса. Спроба скористатися нею дає непередбачувані наслідки. Копіювання ж полів з іншого об'єкта призводить до того, що вказівники в різних об'єктах указують на одну й ту саму змінну, що бажано далеко не завжди. Отже, якщо об'єкти класу містять вказівники, то зазвичай *необхідні власні конструктори*.

Для ілюстрації розглянемо дуже простий клас: атрибутом є вказівник на цілу змінну, а методи дозволяють установити й отримати її значення, а також перевірити, чи існує взагалі ця змінна. У класі означимо також три конструктори й деструктор.

```
class PInt { int * p;
public:
  PInt();
  PInt(int i);
```

```
  PInt(const PInt& old);   //конструктор копії
  ~PInt();                 //деструктор
  bool isDefined();        //чи існує змінна
  void setValue(int i);    //встановити значення
  bool getValue(int &x);   //повернути значення
  const PInt& operator=(const PInt& old)
  // нижче додамо ще один прихований метод
};
```

Реалізація методів

Спочатку реалізуємо два конструктори – з цілим параметром і без параметрів. *Конструктор з цілим параметром* створює динамічну цілу змінну, ініціалізує вказівник у створеному об'єкті її адресою й надає їй початкове значення.

```
PInt(int i) : p(new int) { *p=i; }
```

Конструктор без параметрів ініціалізує вказівник в об'єкті значенням `NULL`.

```
PInt():p(NULL){}
```

Отже, ненульовий вказівник в об'єкті стає ознакою того, що ціла змінна отримала значення, нульовий – що не отримала.

Метод перевірки `isDefined` визначає, чи встановлено вказівник на деяку змінну.

```
bool PInt::isDefined()
{ return p!=NULL; }
```

Конструктор копії має забезпечити кожному об'єкту власний екземпляр цілої змінної. У новому об'єкті він присвоює вказівнику значення `NULL`, якщо базовий об'єкт не має змінної, інакше створює нову цілу змінну, ініціалізує її значенням змінної, що належить іншому об'єкту, і встановлює на неї вказівник в об'єкті.

```
PInt::PInt(const PInt& old);
{ if (!old.isDefined()) p = NULL;
  else { p = new int(*(old.p)); }
}
```

Конструктор копії дозволяє ініціалізувати об'єкт іншим об'єктом, наприклад як `b` і `c` в такій інструкції:

```
PInt a(5), b(a), c=a;
```

Вказівники в об'єктах `a`, `b`, `c` встановлено на три різні цілі змінні, кожна зі значенням `5`.

✓ Стандартні конструктори копії здійснюють копіювання значень полів об'єкта, яке називається **неглибоким**. Для об'єк-

тів з полями-вказівниками зазвичай потрібно **глибоке копіювання** – копіювання всіх або деяких даних, досяжних з об'єкта за допомогою вказівників.

Стандартний деструктор виконується перед знищенням об'єкта. У класі `PInt` деструктор має звільнити пам'ять, зайняту даними, на які встановлено поле-вказівник у об'єкті. Звільняти пам'ять і встановлювати вказівник "у нікуди" будемо допоміжним методом із заголовком `void dispose()`, який знадобиться в ще одному методі. Оголосимо його прихованим, щоб уникнути неконтрольованого використання за межами методів класу. Його реалізація має такий вигляд:

```
void PInt::dispose()
{ if(p!=NULL) {delete p; p=NULL;} }
```

Деструктор класу `PInt` лише викликає цей метод.

```
PInt::~PInt()
{ dispose(); }
```

Без цього деструктора ціла змінна, на яку посилається вказівник в об'єкті, після знищення об'єкта залишається в пам'яті, але є недоступною, оскільки на неї не вказує жоден вказівник.

✓ Зайняту, але недоступну пам'ять називають "*сміттям*". Накопичення "сміття" небезпечно тим, що місця для потрібних даних у вільній пам'яті може не вистачити. Одним із джерел "сміття" є динамічні дані, досяжні з об'єктів. Знищення цих даних у деструкторі протидіє накопиченню "сміття".

Метод установки значення `setValue` присвоює цілій змінній значення, задане параметром. При цьому, якщо вказівник у об'єкті нульовий, то метод спочатку створює змінну. Якщо створити змінну не вдалося, то програма аварійно закінчується.

```
void PInt::setValue(int i)
{ if(p==NULL) p = new int;
  *p=i;
}
```

Цей метод дозволяє надати значення вже існуючому об'єкту.

```
PInt a; a.setValue(10);
```

Метод отримання значення `getValue` присвоює значення цілої змінної, на яку встановлено вказівник в об'єкті, своєму параметру-посиланню, але тільки тоді, коли ця змінна є. Це переві-

ряється методом isDefined. Метод getValue повертає ознаку того, що значення присвоєно параметру.

```
bool PInt::getValue(int &x)
{ if(!isDefined()) return false;
  x=*p; return true;
}
```

Цей метод дозволяє отримати ціле значення з об'єкта, наприклад:

```
PInt a; a.setValue(10);
int v;
if(a.getValue(v)) cout << v;
else cout << "No value";
```

Метод присвоювання має надати цілій змінній, на яку встановлено вказівник в об'єкті, ціле значення з іншого об'єкта. Якщо в іншому об'єкті значення немає, то таке присвоювання призводить до того, що й поточний об'єкт втрачає значення (за допомогою методу dispose).

```
PInt& PInt::operator=(const PInt& old)
{ if(old.getValue(*p)==NULL) dispose(); }
```

Створення й знищення динамічного масиву

Створити динамічний масив можна за допомогою операції new. Нехай T – тип елементів масиву, p – вказівник типу T*. Присвоювання

```
p = new T [n]
```

створює масив елементів типу T з індексами 0, 1, …, n-1 і встановлює p на його перший елемент. Після цього вирази *(p+i) й p[i] позначають i-й елемент масиву. Як n можна записати довільний вираз із додатним цілим значенням.

Операція delete звільняє пам'ять з-під динамічного масиву. Якщо вказівник p встановлено на початок масиву, то масив знищується так:

```
delete [] p;
```

Приклад. Програма створює динамічний масив з п'яти цілих елементів, присвоює їм значення від 0 до 4, виводить їх і знищує створений масив.

```
int main(){
  int * p = new int[5];        // створення масиву
  int i;
  for(i=0; i<5; p[i]=i, ++i);
```

```
  for(i=0; i<5; ++i)
    cout << *(p+i) << ' ';
  delete [] p;                  // знищення масиву
  return 0;
}
```

Докладніше роботу з динамічними масивами розглянемо в наступному розділі.

Вправи

4.1. Імітувати виконання програми з такою головною функцією.

```
int main(){
  int * p1 = new int(0), *p2 = new int(2),
      * p3 = new int;
  *p3=*p1; *p1=*p2; *p2=*p3;
  delete p3; p3=p1;
  cout << *p3 << *p1 << *p2;
  delete p1; delete p2;
  return 0;
}
```

4.2. Написати функцію з двома параметрами, що є вказівниками на `int`. Функція обмінює місцями значення динамічних змінних, на які встановлено ці вказівники.

4.3. До класу `PInt` додати метод обміну місцями цілих значень, на які вказують вказівники в об'єктах.

4.4. До класу `PInt` додати методи введення значення в об'єкт за допомогою клавіатури й виведення з об'єкта на екран.

4.2. Абстрактний тип "стек" і його реалізація

Поняття абстрактного типу даних

Дамо спрощений погляд на одне з фундаментальних понять програмування – абстрактний тип даних. Згадаймо: тип даних – це множина значень плюс певний набір операцій з ними. У програмуванні значення зображуються даними, що мають певні розміри й внутрішню організацію, а операції реалізуються підпрограмами. Деякі типи в мовах програмування є вбудованими – для їх операцій сама мова забезпечує позначення й реалізацію.

Якщо замість конкретних даних розглядати абстрактні елементи, то виникає ще одне поняття – **абстрактний тип даних**

(АТД). Дуже спрощено, АТД – це набір операцій, що мають певні властивості, з деякими абстрактними елементами. Що це за елементи, АТД не показує.

На основі АТД можна створити **реалізацію АТД** – вибрати конкретну множину значень, тобто даних, організованих певним чином, і написати підпрограми, що реалізують операції. Реалізації АТД можуть бути різними, залежно від того, як організовано дані та їх обробку в підпрограмах. Проте набір підпрограм та їх заголовки мають бути однаковими в усіх реалізаціях, щоб ними можна було користуватися, не знаючи нічого про організацію даних і особливості їх обробки.

Набір підпрограм АТД називається його **інтерфейсом**, а конкретні реалізації АТД – **структурами даних**.

Абстрактний тип даних "стек"

Проілюструємо поняття АТД і його реалізації на прикладі стека. **Стек** – це спосіб обробки даних, що додаються до деякого набору й вилучаються з нього за принципом LIFO (*Last In, First Out* – "останнім прийшов, першим пішов"). Опишемо неформально АТД "стек" і наведемо його можливу реалізацію.

Набір даних у стеку розглядається як послідовність елементів (неважливо, якого саме типу), організована за принципом LIFO: елементи додаються й вилучаються на початку. Початок послідовності назвемо **верхівкою (вершиною) стека**, кінець – **дном**. Зафіксуємо набір операцій зі стеком:

– створити порожній стек;
– додати елемент (заштовхнути елемент у стек);
– вилучити елемент із верхівки стека й повернути його (виштовхнути елемент зі стека);
– знищити стек.

Отже, набір операцій зі стеком як послідовністю однотипних елементів, вимоги до цих операцій (додавати й вилучати елементи за принципом LIFO) і фіксація їх вигляду утворюють неформальний опис АТД "стек".

АТД "стек" зобразимо класом `TStack`, тип елементів стека позначимо ім'ям `T`. Спочатку в класі запишемо лише заголовки методів, не уточнюючи атрибутів. Операції "створити порожній

стек" і "знищити стек" реалізуємо конструктором і деструктором класу, інші операції – методами. До них додамо конструктор копії та оператор присвоювання.

```
class TStack
{ public:
  TStack();                 // створити порожній стек
  ~TStack();                // знищити стек
  TStack(const TStack &);
  const TStack& operator=(const TStack &);
  bool push(T elem);        // заштовхнути елемент
  bool pop(T & elem);       // виштовхнути елемент
  ...                       // атрибути (див. нижче)
};
```

Операції "заштовхнути" й "виштовхнути" можуть бути успішними або неуспішними, тому методи `push` і `pop` повертають булеву ознаку успішності. Вилучений елемент повертається за допомогою параметра-посилання.

Реалізація стека в динамічному масиві

Наведемо приклад реалізації АТД "стек". По-перше, уточнимо тип елементів `T`: нехай це буде `char`.

```
typedef char T;
```

По-друге, зафіксуємо спосіб зображення послідовності елементів. Спочатку розглянемо динамічний масив, на який указує вказівник `stack` в об'єкті. Дно стека розташуємо в першому елементі масиву – тоді додавання елементів та їх вилучення збільшує або зменшує довжину зайнятого початку масиву. Цю довжину, що водночас є індексом першого незайнятого елемента масиву, наступного після верхівки стека, будемо зберігати в окремому атрибуті `len`. Ще один атрибут `lenMax` зберігає максимально можливу довжину стека.

Створюючи динамічний масив, необхідно вказати його довжину. Можна було б передавати її як аргумент у виклику конструктора, проте конструктор, оголошений у класі `TStack`, не має параметрів. Отже, оголосимо ім'я константи `S_SIZE` (стартовий розмір), що визначає початкову довжину масиву.

Бажано, щоб ця константа була недоступною за межами класу та його методів. Проте оголошення вигляду

```
private: const int S_SIZE = 100;
```

є помилковим, оскільки *ініціалізувати атрибути в оголошенні класу не можна*. Отже, оголосимо `S_SIZE` як статичну константу, що розміщується в статичній пам'яті програми й жодному об'єкту не належить (див. підрозд. 3.6).

```
private: static const int S_SIZE = 100;
```

Отже, запишемо клас `TStack`.

```
class TStack {
public:
  TStack();               // створити порожній стек
  ~TStack();              // знищити стек
  bool push(T elem);      // заштовхнути елемент
  bool pop(T & elem);     // виштовхнути елемент
private:
  static const int S_SIZE = 100;
  T * stack;
  unsigned len;
  unsigned lenMax;
};
```

Нарешті, напишемо методи, що реалізують операції.

Створити порожній стек. Ініціалізуємо вказівник в об'єкті адресою першого елемента створеного динамічного масиву, максимальну довжину стека – константою `S_SIZE`, а довжину зайнятої частини стека – нулем.

```
TStack::TStack():
  stack(new T[S_SIZE]), lenMax(S_SIZE), len(0) {}
```

Додати елемент у стек. Перевіримо, чи є місце в стеку для ще одного елемента. Якщо місця немає, то повернемо ознаку неуспішності, інакше додамо елемент, збільшивши довжину зайнятої частини стека.

```
bool TStack::push(T elem)
{ if(len==lenMax) return 0;
  stack[len++] = elem;
  return 1;
};
```

Виштовхнути елемент зі стека. Якщо стек порожній, то повернемо ознаку неуспішності, інакше вилучимо елемент, зменшивши довжину зайнятої частини стека.

```
bool TStack::pop(T & elem)
{ if(len == 0) return 0;
  elem = stack[--len];
  return 1;
};
```

Знищити стек. Перед тим, як знищити стек, що містить вказівник на динамічний масив, необхідно звільнити пам'ять, зайняту масивом, інакше вона стане "сміттям".

```
TStack::~TStack()
{ delete [] stack; stack=NULL; }
```

Приклад. Розглянемо головну функцію програми, що отримує від клавіатури непорожні символи й записує їх у стек, а після закінчення введення виводить їх на екран у зворотному порядку.

```
int main() {
  TStack stack; char c;
  while(cin >> c)
    if(!stack.push(c))
      { cout << "Stack Overflow\n"; break; }
  cout << "Symbols in reversal\n";
  while(stack.pop(c))
    cout << c;
  return 0;
}
```

Вправи

4.5. До класу стеків додати метод визначення, чи є стек порожнім.

4.6. Реалізувати конструктор копії та оператор присвоювання класу стеків.

4.7. Змінити метод додавання елемента: якщо масив заповнено, то створюється новий масив подвоєної довжини, у нього копіюється поточний стек, і після цього додається елемент. Попередній масив знищується.

4.8. До класу стеків додати конструктор, цілий додатний параметр якого задає кількість елементів у створюваному масиві. Якщо значення аргументу у виклику конструктора не є додатним, то довжиною масиву стає `S_SIZE`.

4.9. Послідовність дужок (,), [,], {, }, <, > називається *правильною*, якщо це пара дужок (), [], {}, <> або пара дужок, між якими записано правильну послідовність дужок, або кілька правильних послідовностей, записаних підряд. Наприклад, послідовності (), ([()]), {}<()[]> є правильними, а ()), (([()]) – ні. Написати програму перевірки, чи є послідовність дужок у тексті правильною. Символи, відмінні від дужок, ігноруються.

4.10. Серед літер тексту особливу роль відіграє знак #; його поява в тексті означає відміну попередньої літери тексту; *k* знаків # підряд відміняють *k* попередніх літер (якщо вони є). Вивести текст, виправлений з урахуванням ролі знака #, наприклад, текст RT#E##HELO#LO виводиться як HELLO.

4.11. З'ясувати, чи є фраза з вхідного текстового файлу паліндромом, тобто такою, що читається з обох боків однаково, наприклад: "А роза упала на лапу Азора" або "Баритон є ноти раб". Великі й малі літери у фразі не розрізняються, пробіли й розділові знаки ігноруються.

4.12. З файлу вводиться не менше двох ненульових чисел. Вивести в інший файл усі числа від першого найбільшого до останнього найменшого або від першого найменшого до останнього найбільшого у зворотному порядку. Наприклад,
3 5 6 10 4 10 1 7 1 5 8 → 1 7 1 10 4 10,
3 5 6 1 4 1 7 10 5 8 → 10 7 1 4 1.

4.3. Абстрактний тип "черга" та його реалізація

Поняття черги

Черга (*англ.* queue) у загальному розумінні – це послідовність, до якої додаються та з якої вилучаються елементи, тільки вилучаються в її початку (з голови), а додаються в кінці (у хвіст). Таке вилучення й додавання означає: елемент, що потрапив у чергу першим, першим же й вилучається. Звідси й інша назва черги: FIFO (*First In, First Out* – "першим прийшов, першим пішов").

Базові операції з чергою:

– створити порожню чергу;
– додати елемент у кінець черги;
– вилучити елемент у початку черги й повернути його;
– знищити чергу.

Набір операцій із чергою як послідовністю однотипних елементів, вимоги до цих операцій (додавати й вилучати елементи за принципом FIFO) і фіксація їх вигляду утворюють неформальний опис АТД "черга".

Реалізація черги в динамічному масиві

Послідовність елементів черги зобразимо в динамічному масиві, на який указує вказівник в об'єкті. На відміну від стека, для роботи з чергою потрібні два індекси: `tail` (хвіст) указує на перший вільний елемент масиву після, `head` (голова) – на перший елемент черги. Додавання елемента збільшує `tail`, вилучення – `head`. Атрибут `lenMax` зберігає максимально можливу довжину черги. Як і в класі стеків (див. підрозд. 4.2), оголосимо приховану статичну константу `S_SIZE`, за допомогою якої створюється динамічний масив. Для потреб реалізації методів додамо ще один атрибут – кількість `count` елементів у черзі.

```
private: static const int S_SIZE = 100;
```

Означимо тип черг у вигляді класу `TQueue`. Нехай елементи черги мають тип `T`. Заголовки методів класу `TQueue` аналогічні заголовкам у класі `TStack` (див. підрозд. 4.2).

```
class TQueue {
public:
  TQueue();               // створити порожню чергу
  ~ TQueue();             // знищити чергу
  TQueue (const TQueue &);
  const TQueue& operator=(const TQueue &);
  bool add(T elem);       // додати елемент
  bool del(T & elem);     // вилучити елемент
private:
  static const int S_SIZE = 100;
  T * queue;
  unsigned head, tail;
  unsigned count;
  unsigned lenMax;
};
```

Наведемо приклад реалізації АТД "черга", типом `T` елементів якої є `char`, тобто перед класом `TQueue` означимо ім'я `T`.

```
typedef char T;
```

Перейдемо до методів, що реалізують операції. Чергу в масиві зазвичай реалізують як *кільцеву*. Така реалізація використовується дуже часто, наприклад **кільцевий буфер** символів у буферизованому введенні даних (див. підрозд. 6.4), черги в операційних системах тощо.

Елементи додаються в кінець черги, тому її хвіст може досягти кінця масиву. У той самий час, коли елементи вилучаються, зсувається голова черги й перші елементи масиву звільняються. Тому, коли кінець масиву заповнено, елементи додаються до черги на звільнені місця в початку масиву (звісно, якщо такі є). Для того, щоб з'ясувати, що вільні місця для нових елементів є, достатньо порівняти поточну кількість елементів з її максимально можливою довжиною. Так само, коли початок черги досягає кінця масиву, подальші вилучення продовжуються з його початку.

Створити порожню чергу. Вказівник в об'єкті встановлюється на початок створеного динамічного масиву, черга порожня, а її максимальна довжина становить `S_SIZE`.

```
TQueue::
TQueue():queue(new T[S_SIZE]), head(0),
    tail(0), count(0), lenMax(S_SIZE) {}
```

Додати елемент у кінець черги. Якщо черга займає весь масив, то додати елемент неможливо. Інакше на вільне місце, указане індексом `tail`, записуються дані, їх кількість та індекс `tail` збільшуються. Якщо при цьому значення `tail` виходить за межі масиву, то воно стає нульовим, тобто наступне додавання буде в початку масиву.

```
bool TQueue::add(T elem) {
  if(count==lenMax)          // додавання неможливе
    return false;
  queue[tail] = elem;        // додавання
  ++count; ++tail;
  if (tail == lenMax) tail = 0;
  return true;
};
```

Вилучити елемент із початку черги й повернути його. Якщо черга порожня, то вилучити елемент неможливо. Інакше зберігається значення з початку черги, указаного індексом `head`, кількість елементів у черзі зменшується, а її початок зсувається. Якщо після

цього значення head виходить за межі масиву, то воно стає нульовим, тобто наступне вилучення буде з початку масиву.

```
bool TQueue::del(T & elem) {
  if (count == 0)            // вилучення неможливе
    return false;
  elem = queue[head];
  --count; ++head;           // вилучення
  if (head == lenMax) head = 0;
  return true;
}
```

Знищити чергу. Перед знищенням об'єкта, що містить вказівник на динамічний масив, звільнимо зайняту ним пам'ять і встановимо його "у нікуди".

```
TQueue::~TQueue()
  { delete [] queue; queue=NULL; }
```

Приклад. Розглянемо головну функцію програми, що отримує за допомогою клавіатури символи й записує їх у чергу. Якщо запис символу в чергу неможливий, то з черги вилучаються й виводяться два символи, лише після цього символ додається до черги.

```
int main(){
  TQueue queue;
  char c;
  while (cin >> c)
    if (!queue.add(c))
      { char c1; queue.del(c1);
        cout << "DEL:" << c1 << '\n';
        if(queue.del(c1))
          cout << "DEL:" << c1 << '\n';
        queue.add(c);
      }
  while(queue.del(c))
    cout << c << '\n';
  return 0;
}
```

Вправи

4.13. До класу черг додати метод визначення, чи є черга порожньою.

4.14. Реалізувати конструктор копії та оператор присвоювання класу черг.

4.15. До класу черг додати конструктор, цілий додатний параметр якого задає кількість елементів у створюваному масиві. Якщо значення аргументу у виклику конструктора не є додатним, то довжиною масиву стає `S_SIZE`.

4.16. Змінити метод додавання елемента: якщо масив заповнено, то створюється новий масив більшої довжини, у нього копіюється поточна черга, після чого додається елемент. Попередній масив знищується.

4.17. На вхід програми даються цілі числа в діапазоні від -1 до $2 \cdot 10^9$, розділені порожніми символами. Кількість чисел не обмежено. Ведеться список чисел: невід'ємне число додається в кінець списку, а число -1 означає, що зі списку вилучається перший елемент (якщо він є). Після введення виводяться числа, що є в списку. Наприклад:
`-1 3 0 5 -1 6` → `0 5 6.`

4.4. Матриці й динамічні масиви

Вказівник `p`, установлений на перший елемент масиву, дозволяє позначати елемент масиву виразом вигляду `p[i]`. Ім'я `p` визначає адресу першого елемента масиву, вираз `i` – розташування потрібного елемента відносно першого (з індексом `0`). Індекси елемента двовимірного масиву `m[i][j]` теж визначають зміщення відносно `m[0][0]` – якщо рядки містять `c` елементів, то це зміщення дорівнює `i*c+j`. Отже, якщо вказівник `p` встановлено на початок двовимірного масиву `m`, то елемент `m[i][j]` можна позначити виразом `p[i*c+j]`. Це дозволяє розглядати двовимірний масив з `r` рядками й `c` стовпчиками як одновимірний з довжиною `r*c`. Одновимірному індексу `k` відповідають двовимірні `[k/c]` і `[k%c]`.

Замість двовимірного масиву можна утворити сукупність динамічних одновимірних масивів так, що їх елементи позначаються звичними виразами вигляду `m[i][j]`.

Приклад. Припустимо, що розміри матриці цілих чисел (`r` рядків і `c` стовпчиків) уводяться під час виконання програми. Рядки матриці зобразимо одновимірними динамічними масивами по `c` цілих елементів у кожному. На початок кожного рядка встановимо вказівник типу `int*`, що є елементом додаткового

масиву, індексованого номерами рядків. На початок додаткового масиву встановимо вказівник `m` типу `int**`, тобто вказівник на вказівники на `int` (рис. 4.2). Тоді вираз `m[i]` позначає *i*-й вказівник у масиві, а вираз `m[i][j]` – *j*-й елемент у масиві цілих, на який указує `m[i]`.

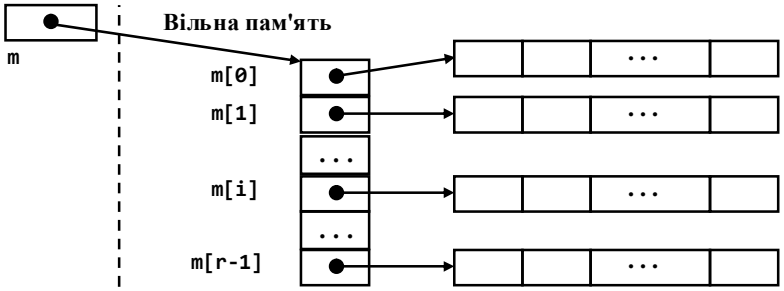

Рис. 4.2. Зображення матриці

Вправи

4.18. Розробити клас, що зображує матриці цілих чисел в описаний спосіб та реалізує введення й виведення матриці.

4.19. Додати до класу матриць метод заповнення матриці псевдовипадковими числами.

4.5. Поняття віртуальної пам'яті

Технологія віртуальної пам'яті

Комп'ютер виконує програму під керуванням операційної системи (ОС), яка зберігає певні дані про програму, записану в оперативній пам'яті (ОП). Програма разом із цими даними називається **процесом** (**задачею**). Сучасні ОС є **багатозадачними**: вони дозволяють виконувати одночасно багато програм.

Насправді під керуванням ОС час роботи процесора розподіляється між процесами невеличкими порціями. Віддавати всю фізичну пам'ять одному процесу в умовах багатозадачності нераціонально, тому вона також розподіляється між процесами. Крім того, існують програми, які цілком не вміщуються в ОП, через що мають завантажуватися лише частинами.

Отже, однією із задач ОС є керування розподілом ОП між процесами. Основою цього керування зазвичай є **технологія**

віртуальної пам'яті (*virtual memory*), запропонована ще в 1959 р. Її суть полягає в тому, що кожен процес має уявний (віртуальний) простір адрес, не залежний від інших процесів і від справжнього розташування даних у ОП. Адреси віртуального простору називаються **логічними**. Простір адрес є неперервним і доступним незалежно від того, якими є реальні фізичні адреси даних процесу й чи є вони взагалі. Фізична ОП ділиться на окремі блоки, що динамічно виділяються процесам. Коли програма або її частина завантажується в блок, логічні адреси певним чином відображаються у **фізичні адреси** в межах блоку, і це дозволяє процесору за командами програми обробляти дані, розташовані за справжніми фізичними адресами.

Технологія віртуальної пам'яті:

– підвищує ефективність використання пам'яті програмами, що виконуються одночасно,

– забезпечує захист їх пам'яті від інших програм,

– дозволяє виконувати програми, які цілком не вміщуються в оперативну пам'ять комп'ютера.

✓ Технологія віртуальної пам'яті значно спрощує створення програми, оскільки в ній не потрібно враховувати, що ОП обмежена, й узгоджувати її використання з іншими процесами.

Сторінки та сегменти

Системи віртуальної пам'яті традиційно поділяють на три класи:

– системи з фіксованим розміром блоку, що називається сторінкою (*сторінкова* організація пам'яті);

– системи з різними розмірами блоків, що називаються сегментами (*сегментна* організація пам'яті);

– комбіновані системи (*сегментно-сторінкова* організація пам'яті).

За **сторінкової** організації як фізична, так і віртуальна пам'ять розглядаються у вигляді набору **сторінок** – блоків однакового розміру, що не перекриваються. Передавання даних між пам'яттю й диском, де записана програма або її частини, здійснюється цілими сторінками.

Процес може виконуватися, якщо його поточна сторінка завантажена в оперативну пам'ять, інакше її необхідно переписати з диска. Нова сторінка може бути записана в довільну незайняту сторінку фізичної пам'яті. Для того, щоб перетворювати логічні адреси на фізичні, процес має таблицю сторінок, яка під час його виконання постійно зберігається в пам'яті ОС і зв'язує логічні сторінки з фізичними. Кожна сторінка має номер, ознаку, чи завантажено її в ОП, адресу її початку у фізичній пам'яті й атрибути, пов'язані з можливостями її обробки. Логічна адреса даних або команд програми має два поля – *номер сторінки* й *зміщення* всередині сторінки. Відповідна фізична адреса обчислюється як адреса початку фізичної сторінки, в яку відображена логічна, плюс зміщення.

Головним у сторінковій організації пам'яті є те, що процесу доступні лише виділені йому фізичні сторінки, а пам'ять за їх межами не доступна.

За **сегментної** організації віртуальна пам'ять програми є набором **сегментів** – ділянок, що можуть мати різні розміри. Сегменти призначені для того, щоб уміщати певні *логічно цілісні частини програми*, наприклад код програми або набори бібліотечних підпрограм. Процес містить таблицю сегментів, в якій кожен сегмент має ім'я (номер), адресу його початку, розмір, ознаку, чи завантажено сегмент у ОП, а також атрибути, пов'язані з можливостями його обробки. Логічна адреса складається з номера сегмента та зміщення всередині сегмента. Якщо сегмент записано у фізичну пам'ять, то фізична адреса даних у ньому утворюється як фізична адреса початку сегмента з таблиці сегментів, плюс зміщення.

У таблицях сегментів кількох різних процесів можуть міститися адреси однієї й тієї самої ділянки ОП, яку займає деякий сегмент. Завдяки цьому кілька процесів *спільно використовують єдиний екземпляр сегмента*.

Зазвичай сторінки мають відносно невеликі розміри порядку кількох кбайт, а розміри сегментів можуть досягати кількох Мбайт і більше.

Сегментна організація пам'яті в чистому вигляді не зустрічається, оскільки зберігання в ОП величезних сегментів так само незручне, як і процес у єдиному блоці. Натомість сучасні обчислювальні системи реалізують **сегментно-сторінкову** організацію пам'яті, за якої сегмент утворюється як набір сторінок. Віртуальна адреса має три поля: номер сегмента, номер сторінки всередині сегмента й зміщення всередині сторінки. Отже, тут відбувається дворівневе перетворення віртуальної адреси на фізичну. За цієї організації процес містить таблицю сегментів, яка містить адресу таблиці сторінок кожного сегмента.

У 32-розрядних ОС програма може адресувати $2^{32} = 4294697296$ байтів, тобто 4 Гбайти віртуальної пам'яті. За сегментно-сторінкової організації пам'яті логічна адреса має три поля: номер сегмента, номер сторінки, зміщення в межах сторінки. Довжина цих полів відповідно 10, 10 і 12 біт. Це забезпечує $2^{10} = 1024$ сегменти по $2^{10} = 1024$ сторінок у кожному, тобто всього сторінок може бути $2^{20} = 1048576$. 12-бітове зміщення визначає розмір сторінки $2^{12} = 4096$ байтів, тобто 4 кбайти.

Вказівник займає 4 байти пам'яті, які містять номер сегмента (10 бітів), сторінки (10 бітів) і зміщення (12 бітів), але завдяки технології віртуальної пам'яті замислюватися над цими подробицями нам не доводиться.

Контрольні запитання

4.1. З яких частин складається пам'ять процесу виконання програми?

4.2. Чим змінні у вільній пам'яті відрізняються від змінних у статичній або автоматичній пам'яті з погляду програми мовою високого рівня?

4.3. Чому змінні у вільній пам'яті називаються динамічними?

4.4. Які операції мови C++ створюють і знищують динамічні змінні? Що є їхніми операндами й результатами?

4.5. Чи можна обробляти динамічну змінну, пам'ять якої звільнено?

4.6. Чи можна повторно знищити одну й ту саму динамічну змінну?

4.7. Опишіть синтаксичні особливості створення й знищення динамічних масивів за допомогою операцій `new` та `delete`.
4.8. Яке копіювання об'єктів називається неглибоким? Чому його зазвичай недостатньо для роботи з об'єктами, що містять вказівники?
4.9. Чим глибоке копіювання об'єктів відрізняється від неглибокого?
4.10. Що таке деструктор?
4.11. У чому полягають синтаксичні особливості нестандартного деструктора?
4.12. Назвіть ситуації, в яких під час виконання програми об'єкт: а) створюється; б) знищується.
4.13. Опишіть поняття "стек".
4.14. Опишіть поняття "черга".
4.15. Чи можна за допомогою двох стеків проімітувати роботу з чергою?

РОЗДІЛ 5.
Успадкування класів

5.1. Відкрите успадкування

Познайомимося з успадкуванням на простому прикладі. Розглянемо клас, що зображує живий організм – тіло (Body). Прихованим атрибутом тіла є ціла маса (mass). Тіло створюється з певною масою, яка передається через параметр конструктора. Припустимо, що маса тіла не може бути більше 100. Це число зобразимо прихованим константним атрибутом класу MMAX. Тіло може їсти (метод eat), збільшуючи масу на задану величину, але маса не може стати більше MMAX. Тіло може втрачати масу (метод drop), але, якщо маса стає недодатною, то тіло їсть, щоб його маса досягла MMAX/2. Масу тіла можна також вивести на екран (метод out).

```
class Body {
  static const int MMAX=100;
  int mass;                    // прихований атрибут
public :                       // відкриті методи
  Body(int m);
  void eat(int d);
  void drop(int d);
  void out();
};
Body::Body(int m) : mass(m)
{ if(mass>MMAX) mass=MMAX; }
void Body::eat(int d)
{ mass+=abs(d);
  if(mass>MMAX) drop(mass-MMAX);
}
void Body::drop(int d)
{ mass-=abs(d);
  if(mass<=0)
    { mass=0; eat(MMAX/2); }
}
void Body::out()
{ cout << "Mass: " << mass;}
```

Іншим різновидом об'єктів є особа (Person) – це тіло з ідентифікуючими даними ID (Identification Data), які є цілим числом і задаються під час створення особи. Отже, атрибути особи – mass та ID. Особа поводиться як тіло, тобто їсть, втрачає вагу й виводить масу на екран, але додатково може діяти (метод act) і виводити не тільки свою масу, а й ID.

Отже, у класі "особа" мають бути як власні атрибути й методи, так і всі ті, що є в класі "тіло". *Проте повторно означати їх у новому класі не потрібно*. Достатньо оголосити новий клас Person, указавши на початку оголошення, що він *успадковує* клас Body.

```
class Person : public Body {
  private: int ID;                // доданий атрибут
public:             // додані й переозначені методи
  Person(int ID_,int m);
  void act(int d);
  void out();
};
Person::Person(int ID_,int m):ID(ID_),Body(m){}
void Person::act(int d)
{ cout<<ID<<" IS ACTING\n"; drop(d); }
void Person::out()
{cout<<"ID: "<<ID<<' '; Body::out(); }
```

Успадкування класів полягає в тому, що в оголошенні нового класу (**нащадка**, або **похідного класу**) використовують інший, раніше оголошений клас (**батько**, або **базовий клас**), до якого додають атрибути й методи.

Успадкування класу Body вказано після імені класу Person і двокрапки. Завдяки йому кожен об'єкт класу Person містить поле mass від класу Body й додане поле ID.

Слово public перед іменем класу-батька – це один із трьох **специфікаторів доступу**, які позначають три різні можливості доступу до полів базового класу за межами похідного.

Специфікатор public позначає **відкритий доступ** – за межами класу-нащадка відкриті поля класу-батька доступні так само, як і відкриті поля нащадка.

Інші способи доступу до полів базового класу за межами похідних класів представлено нижче.

Отже, за межами класу Person доступні його методи act і out, а також методи класу Body (eat, drop і Body::out), тобто неформально *особа – це тіло (і ще дещо)*.

- ✓ Приховані поля класу-батька (атрибут mass) залишаються прихованими й для класу-нащадка, тому в методах класу Person атрибут mass недоступний. Безпосереднє використання імені mass було б помилкою, на яку вказав би компілятор.

Звернімо увагу на дві синтаксичні особливості в класі Person. По-перше, ініціалізація в конструкторі містить явний виклик конструктора класу Body, тобто створення об'єкта-особи включає створення об'єкта-тіла. По-друге, до об'єктів класу Person застосовні обидва методи out – власний і успадкований. Їх виклики мають вигляд, наприклад, p.out() і p.Body::out().

Проілюструємо роботу з об'єктами класу Person за допомогою як його власних методів, так і методів базового класу Body.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{ Person man(2014,365);  // об'єкт-особа
  // виклики успадкованих методів
  man.Body::out(); cout << '\n'; // "Mass: 100"
  man.drop(40);
  man.eat(17);
  // виклики власних методів
  man.act(14); // "2014 IS ACTING"
  man.out();   // "ID: 2014 Mass: 63"
  return 0;
}
```

- ✓ Клас-батько *сумісний за присвоюванням* із класом-нащадком. Присвоювання об'єкту класу-батька – це копіювання полів, наявних у класі-батьку.
- ✓ Клас-нащадок *не* сумісний за присвоюванням із класом-батьком, оскільки це присвоювання залишає невизначеними власні поля об'єкта класу-нащадка.

5.2. Захищені поля

Поля, недоступні за межами методів класу, можна зробити доступними в методах його класів-нащадків.

У класі Body оголосимо атрибут mass, раніше прихований, зі специфікатором protected – "*захищений*".

```
class Body {
  ...
  protected : int mass;          // захищене поле
public :
  ... усе, як означено раніше
};
```

> Поля, оголошені після слова protected, називаються **захищеними**.

Захищене поле класу-батька доступне в методах класів-нащадків (якщо успадкування в них не є прихованим – про це нижче), але не доступне за їх межами.

Отже, тепер захищений атрибут mass класу Body доступний у методах класів-нащадків класу Body, але не за їх межами. Наприклад, у класі Person можна змінити метод out так, щоб не викликати метод Body::out.

```
void Person::out()
{ cout << "ID: " << ID << " Mass: " << mass; }
```

Атрибут ID у класі Person також зробимо захищеним, щоб він був доступним у можливих похідних класах.

```
class Person : public Body {
  protected: int ID;              // захищене поле
  ...
}
```

✓ Якщо під час розробки класу невідомо, чи матиме він нащадків, яким буде потрібен доступ до його атрибута, то краще оголосити цей атрибут захищеним, а не прихованим.

5.3. Ієрархії класів

У класу може бути скільки завгодно класів-нащадків, а клас-нащадок може бути базовим для інших класів. Це дозволяє створювати ланцюги й розгалуження класів, в яких наступні є нащадками попередніх. Такі системи класів називаються *ієрархіями*.

Продовжимо приклад із класами тіл і осіб. Успадкуємо клас осіб Person у двох нових класах – студентів Student і робітників Worker.

Студент (Student) – це особа, яка має додатковий дійсний атрибут mark (середній бал за навчання). Студент учиться – втрачає масу й підвищує свій середній бал (метод study). Виведення даних про студента – це виведення його даних як особи й середнього балу.

```
class Student : public Person {
  protected: double mark;
public:
  Student(int ID_,int m);
  void study(int d);
  void out();
};
Student::Student(int ID_,int m)
    : Person(ID_,m), mark(0) {}
void Student::study(int d)
{ cout<<ID<<" IS STUDYING\n"; drop(d); mark+=0.1*d; }
void Student::out()
{ cout << "Student: "; Person::out();
  cout << " Mark: " << mark;
}
```

Робітник (Worker) є особою, яка не має додаткових атрибутів і працює (метод work), інтенсивно витрачаючи масу.

```
class Worker : public Person {
public:
  Worker(int ID_, int m);
  void work(int d);
  void out();
};
Worker::Worker(int ID_, int m) : Person(ID_, m){}
void Worker::work(int d)
{ cout<<ID<<" IS WORKING\n"; drop(3*d); }
void Worker::out()
{ cout << "Worker: "; Person::out(); }
```

Отже, класи Student і Worker відкрито успадковують клас Person, а він, у свою чергу, – клас Body. Завдяки цьому до об'єк-

тів-студентів і об'єктів-працівників застосовні відкриті методи як їхніх класів, так і класів-предків Body й Person.

```
int main()
{ Student studdy(2013, 70);
  studdy.act(20); studdy.eat(16);
  studdy.study(11);
  studdy.out(); cout << '\n';
  Worker worky(1989, 120);
  worky.out(); cout << '\n';
  worky.eat(20); worky.work(20);
  worky.Person::out(); cout << '\n';
  system("pause");
}
```

Вихід цієї функції (з наведеними вище класами) буде таким:

```
2013 IS ACTING
2013 IS STUDYING
Student: ID: 2013 Mass: 55 Mark: 1.1
Worker: ID: 1989 Mass: 120
1989 IS WORKING
ID: 1989 Body: 80
```

Отже, класи тіл Body, осіб Person, студентів Student і робітників Worker утворюють ієрархію, в якій є клас-батько й кілька класів-нащадків. Зв'язки між класами в ієрархії позначають стрілками, що ведуть *від похідних класів до базових* (рис. 5.1).

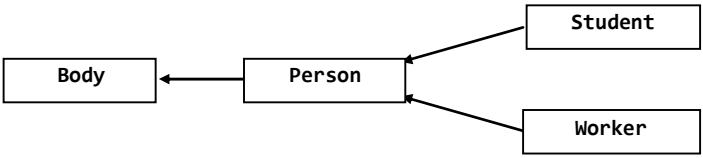

Рис. 5.1. Приклад ієрархії класів

✓ Успадкування полегшує створення нових класів і збільшує можливості повторного використання коду. Воно дозволяє утворювати ієрархії класів, які зображують ієрархії понять реального світу, звичні для людини.

5.4. Приховане й захищене успадкування

Приховане успадкування задається специфікатором private у класі-нащадку й означає: відкриті й захищені поля класу-батька недоступні за межами класу-нащадка, як у "зовнішньому світі", так і в методах класів, похідних від цього нащадка.

Приклад. Запишемо заголовок класу Person:

```
class Person:private Body //приховане успадкування
{ ... усе, як було вище };
```

Виклики методів класу Body, які тепер не можна застосувати до об'єкта-особи, "заховано" в коментарі.

```
int main()
{ Person man;
  man.setPerson(2009,50);    // метод класу Person
  // man.Body::out(); - недоступний
  // man.eat(16);     - недоступний
  man.act(11);               // методи класу Person
  man.out();                 // 50 – 11 = 39
}
```

Як бачимо, особа, приховуючи своє "тілесне походження", тепер не може навіть поїсти. ◂

Захищене успадкування задається специфікатором protected і полягає в тому, що відкриті й захищені поля класу-батька доступні в методах усіх класів, похідних від класу-нащадка, але недоступні за межами цих класів (у "зовнішньому світі").

Продовжимо приклад із класами тіл і осіб і змінимо заголовок класу Person.

```
class Person:protected Body //захищене успадкування
{ ... усе, як вище };
```

Тепер відкриті й захищені поля класу-батька Body доступні в методах класів Student і Worker, похідних від Person, проте за їх межами (у "зовнішньому світі") ці поля недоступні. Неформально для класів-нащадків особа залишається тілом, а для "зовнішнього світу" – ні.

5.5. Виклики конструкторів за умов успадкування

Розглянемо дуже простий приклад: базовий клас BC має "сина" D1 і "онука" D2. В усіх цих класах є конструктори й деструктори, що виводять відповідні повідомлення про своє виконання. У головній функції лише оголошено об'єкт класу D2.

```
#include <iostream.h>
class BC {
public :
  BC() { cout << "BC Con\n"; }
  ~BC(){ cout << "BC Des\n"; }
};
class D1 : public BC {
public:
  D1() { cout << "  D1 Con\n"; }
  ~D1(){ cout << "  D1 Des\n"; }
};
class D2: public D1 {
public:
  D2() { cout << "    D2 Con\n"; }
  ~D2(){ cout << "    D2 Des\n"; }
};
int main()
{ D2 d;
  return 0;
}
```

Виконання цієї програми дасть такий вихід:

```
BC Con
  D1 Con
    D2 Con
    D2 Des
  D1 Des
BC Des
```

✓ Коли створюється об'єкт похідного класу, виконуються конструктори всіх класів-попередників, починаючи з базового класу й закінчуючи класом об'єкта. Перед знищенням об'єкта деструктори цих класів виконуються у зворотному порядку.

Ще один приклад. У попередніх розділах фігурували клас-батько Body та його нащадок Person. У заголовку конструктора Person явно викликався конструктор Body:

```
Person(int ID_, int mass_)
  : ID(ID_), Body(mass_) {}
```

Це означення явно вказує, що конструктор класу Body виконується перед початком виконання тіла конструктора Person.

✓ У конструкторі класу-нащадка виклик конструктора класу-батька з параметрами записується *явно у вигляді ініціалізатора*, лише замість імені атрибута вказується ім'я класу.

5.6. Вказівники та посилання на об'єкти базових і похідних класів

✓ Вказівник на клас-батько можна встановити на об'єкт будь-якого похідного класу й за допомогою вказівника позначати поля, наявні в класі-батьку. Проте такий вказівник не дає доступу до полів, доданих у похідному класі.

Приклад. Розглянемо функцію з вказівниками на об'єкти класів Body й Student (з відкритим успадкуванням).

```
int main()
{ Person * pPr = new Person(2010,80);
  Body * pBd = pPr;
  pBd->eat(120);                // метод класу Body
  // pBd->act(5); - метод класу Person: помилка
  delete pBd;       // після цього
  pPr->out();       // наслідки непередбачувані
  return 0;
}
```

Після того, як за допомогою вказівника pBd на клас Body знищено об'єкт-тіло, значення поля mass в об'єкті-особі *pPr стає "сміттям". Поле pPr->ID залишається без змін, але подальша робота з об'єктом *pPr небезпечна. ◂

✓ Вказівник на похідний клас можна встановити на об'єкт класу-батька, якщо *адресу об'єкта класу-батька перетворити до типу адрес похідного класу*. Проте поля даних, додані в похідному класі, насправді відсутні в об'єкті, тому їх використання за допомогою вказівника на похідний клас має *непередбачувані наслідки*.

Приклад. Створимо об'єкт-тіло *pBd й установимо на нього вказівник Person *pPr, попередньо перетворивши адресу об'єкта до типу Person*. Вказівник pPr дозволяє працювати з об'єктом-особою *pPr, проте використовувати поле ID, відсутнє в об'єкті *pBd, небезпечно, оскільки імені ID відповідає чужа пам'ять за межами об'єкта-тіла *pBd.

```
int main()
{ Body *pBd = new Body(99);
  //  pPr=pBd; - це було б помилкою
  Person *pPr=(Person *)pBd;
  pPr->out();          // значення ID - ???
  // ДАЛІ НЕБЕЗПЕЧНО!
  delete pPr;          //звільняється чужа пам'ять
  return 0;
}
```

Арифметичні операції з вказівниками змінюють їх значення на величину, кратну розміру типу, базового для вказівника (sizeof Person для вказівника типу Person*, sizeof Body для Body* тощо). Звідси вказівник на деякий клас можна встановити, наприклад, на елемент масиву об'єктів іншого класу, але збільшення або зменшення цього вказівника матиме *непередбачувані й небезпечні наслідки*.

Як і вказівник, посилання, типом якого є клас-батько, дозволяє отримати доступ до об'єкта будь-якого похідного класу.

Зокрема, у функції типом параметра-посилання може бути клас-батько, а аргументами у викликах – об'єкти похідних класів. Проте тип параметра визначає, що далі у функції доступні лише поля об'єкта-аргументу, наявні в класі-батьку.

5.7. Віртуальні функції та поліморфізм

Віртуальна функція

Розглянемо три класи: батько BC, його син D1 і онук D2. Клас BC має методи обробки process() і виведення out(). У класах-нащадках успадковано від класу-батька метод обробки й означено власні методи виведення. Метод виведення out() кожного

з класів виводить назву свого класу, а метод обробки викликає метод виведення. Запишемо ці класи у простій програмі.

```
#include <iostream>
using std::cout;
class BC {
public :
  void process();
  void out();
};
class D1 : public BC {
public : void out();
};
class D2 : public D1 {
public : void out();
};
void BC::process(){ out(); }
void BC::out(){cout << "BC ";}
void D1::out(){cout << "D1 ";}
void D2::out(){cout << "D2 ";}

int main()
{ BC bc; D1 d1; D2 d2;
  bc.out(); d1.out(); d2.out();
  bc.process(); d1.process(); d2.process();
  return 0;
}
```

Виходом цієї програми є рядок `BC D1 D2 BC BC BC`. Для кожного з об'єктів виконується метод виведення того класу, якому належить об'єкт. Метод `process()` означено в класі-батьку так, що він викликає метод виведення класу-батька, тому результат обробки об'єктів не залежить від їх класів.

Змінимо головну функцію так, щоб вона вводила ціле число, залежно від нього встановлювала вказівник на динамічний об'єкт одного з класів `BC`, `D1`, `D2` і далі взаємодіяла з об'єктом за допомогою вказівника. Вказівник на клас-батько можна встановити на об'єкт похідного класу, тому скористаємося вказівником типу `BC*`.

```
int main()
{ BC * p;
  int r=0;
```

```
  cout << "Enter 0, or 1, or 2: ";
  cin >> r;
  if(r==1) p = new D1;
  else if(r==2) p = new D2;
  else p = new BC;
  p->process();            //метод класу-батька
  delete p;
  return 0;
}
```

Аналогічно першій програмі, яким би не було вхідне число, програма виводить BC, тобто виконується тільки метод out() класу-батька, оскільки у виразі p->process() *компілятор визначає* потрібний метод process() *за типом вказівника* BC*.

УВАГА! Змінимо клас-батько BC – на початку заголовка методу out() запишемо слово virtual.

```
class BC {
  public :
    void process();
    virtual void out();
};
```

З цим методом out() перша з програм, указаних вище, виводить рядок BC D1 D2 BC D1 D2. Це означає: у методі process() викликається метод out() *того класу, якому належить оброблюваний об'єкт*. Друга програма після введення 0 виводить BC, після 1 – D1, після 2 – D2, тобто у виразі p->out() викликається метод out() *того класу, на об'єкт якого встановлено вказівник* p типу BC*.

> Метод класу, оголошений у класі-батьку з ключовим словом virtual, називається **віртуальним**. Методи похідних класів з таким самим прототипом, що й у віртуального методу класу-батька, теж називаються віртуальними.

Отже, якщо метод класу-батька містить виклик віртуального методу й застосовується до об'єкта класу-нащадка, то, на відміну від звичайних методів, викликається віртуальний метод класу, якому належить об'єкт, а не метод класу-батька. Аналогічно, якщо на об'єкт класу-нащадка встановлено вказівник на клас-батько й за допомогою вказівника позначено виклик віртуального методу, то викликається метод того класу, якому належить об'єкт.

У методі process() класу-батька BC нічого не відомо про клас, якому належить оброблюваний об'єкт. Аналогічно в другій програмі наперед невідомо, якому класу належить створений динамічний об'єкт. Звідси, який саме метод out() необхідно викликати, в обох ситуаціях визначається за типом об'єкта *під час виконання програми*, а не компіляції.

Аналогічно вказівникам, посилання, типом якого є базовий клас, можна встановити на об'єкт похідного класу. Якщо до посилання застосувати метод, віртуальний у класі-батьку, то виконується метод того класу, якому належить об'єкт, базовий для посилання. Наприклад, виконання інструкцій

```
D1 d1; D2 d2;
BC & r1=d1, & r2=d2;
r1.process(); r2.process();
```

дає вихід D1 D2.

Віртуальний метод, означений у класі-батьку, успадковується так само, як інші методи. Якби, наприклад, у класі D2 метод out() не був оголошений, то до об'єкта класу D2 застосовувався б метод D1::out() класу, найближчого з ієрархії класів, що мають цей метод.

- ✓ Записувати слово virtual в оголошенні віртуальних методів у похідних класах необов'язково. У заголовку реалізації віртуальної функції за межами класу слово virtual відсутнє.
- ✓ Деструктор класу може бути віртуальним, конструктор – ні.

Поняття поліморфізму

Методи out() у класах BC, D1, D2 (див. початок підрозділу), як віртуальні, так і звичайні, мають однакові прототипи й різні реалізації. *Одна й та сама сутність* операції out (виведення імені класу об'єкта) має *різні форми*, тому ця операція називається **поліморфною** (буквально – багатоформною).

> **Поліморфізм** – це існування різних реалізацій однієї операції для об'єктів різних типів.

Існує погляд, згідно з яким поліморфними вважаються операції, реалізовані лише віртуальними функціями (**динамічний поліморфізм**, або **поліморфізм часу виконання програми**). Клас, що містить хоча б один віртуальний метод, називають **по-**

ліморфним класом. Головне, що поліморфні класи, завдяки динамічному зв'язуванню, надають можливість обробляти об'єкти, тип яких стає відомим лише під час виконання програми.

Проте частіше поліморфізм розуміють не лише як динамічний. Зокрема, переозначення функцій та операторів у мові C++ реалізує **статичний поліморфізм** (**поліморфізм часу компіляції**).

Поліморфізм втілює принцип "*один інтерфейс, кілька реалізацій*". Він дозволяє використовувати спільний інтерфейс різних класів і не замислюватися над відмінностями в реалізації операцій, прихованих у класах. Це сприяє створенню ясного, короткого й гнучкого коду.

Статичне й динамічне зв'язування

Дамо спрощений погляд на те, як виклику функції (будь-якої, а не лише методу класу) ставиться у відповідність сама функція. Компілятор перетворює функцію на код (послідовність команд) і записує в пам'ять за деякою адресою. Також компілятор створює таблицю, яка кожному імені функції програми ставить у відповідність адресу початку її коду (*таблиця імен*). За допомогою таблиці імен визначається адреса початку коду функції й записується в місці її виклику.

> Запис адреси початку коду функції (адреси функції) у місці її виклику називається **зв'язуванням виклику функції** (з її кодом).

Виклики невіртуальних функцій зв'язує *компонувальник* (*редактор зв'язків*). Для віртуальних функцій зв'язування *відкладається до часу виконання* програми.

> Зв'язування під час побудови програми називається **раннім**, або **статичним**, а під час виконання програми – **пізнім**, або **динамічним**.

Якщо клас має хоча б одну віртуальну функцію, то, щоб забезпечити динамічне зв'язування, компілятор створює **таблицю віртуальних методів** (ТВМ) класу, яка містить адреси всіх віртуальних методів. Якщо в класі-нащадку віртуальна функція не оголошена, а успадкована з класу-попередника, то в ТВМ класу-

нащадка записується адреса функції класу-попередника. ТВМ кожного класу додається до коду програми.

Коли створюється об'єкт класу, в якому оголошено хоча б один віртуальний метод, до об'єкта додається вказівник, що встановлюється на ТВМ його класу. Якщо до об'єкта застосовується віртуальний метод, то за допомогою вказівника в об'єкті й ТВМ класу визначається адреса коду методу. Виклик методу зв'язується з цією адресою й виконується.

✓ Динамічне зв'язування уповільнює виконання програми, хоча зазвичай це не є суттєвим. Важливо, що динамічне зв'язування дозволяє обробляти об'єкти, тип яких невідомий під час компіляції. Завдяки цьому програмування в багатьох ситуаціях спрощується й стає гнучкішим.

5.8. Суто віртуальна функція та абстрактний клас

Суто віртуальна функція – це віртуальна функція, яка не має реалізації.

Продовжимо приклад із трьома класами `BC`, `D1`, `D2` й оголосимо метод `out()` у базовому класі `BC` як суто віртуальний. У кінці заголовка суто віртуального методу записується текст `"=0"`, а реалізації він не має.

```
class BC { public :
  void process();
  virtual void out()=0;    //суто віртуальний метод
};
```

Абстрактний клас – це клас, що має хоча б одну суто віртуальну функцію.

Щойно наведений клас `BC` є абстрактним. Головна особливість абстрактного класу – об'єктів цього класу *не може бути*, адже його суто віртуальні функції не означені. Проте абстрактний клас можна успадковувати й використовувати для оголошення вказівників і посилань.

У нашому прикладі похідні класи `D1` і `D2` мають реалізації методу `out()`, тому можна створювати об'єкти цих класів і працювати з ними. Розглянемо створення об'єктів цих класів і доступ

до них за допомогою вказівника на абстрактний клас-батько в такій головній функції:

```
int main()
{ BC * p;           //вказівник на абстрактний клас
  int r=0;
  cout << "Enter int: "; cin >> r;
  if(r%2==1) p = new D1;   //new BC було б помилкою
  else p = new D2;
  p->out();
  delete p;
  return 0;
}
```

Якщо під час виконання програми ввести непарне число, то виходом буде `D1`, а якщо парне – то `D2`.

✓ Основне призначення абстрактного класу – *зафіксувати інтерфейс*, який мають реалізувати похідні класи.

Класи, похідні від абстрактного, також можуть бути абстрактними. Вони зазвичай призначені для того, щоб розширити інтерфейс абстрактних класів-попередників.

Вправи

5.1. Написати абстрактний клас із двома методами. Суто віртуальний метод має заголовок `int f(int)`, а інший метод аналогічний функції `fTab` з підрозд. 1.5.

5.2. Написати два нащадки класу з попередньої вправи, що містять реалізації абстрактного методу `f`. Написати програму, в якій створюються об'єкти класів-нащадків з попередньої вправи, а їх функції табулюються на заданому цілочисловому проміжку.

5.3. Написати класи, аналогічні класам з попередніх вправ, і скористатися ними для розв'язання вправ з підрозд. 1.5.

Контрольні запитання

5.1. Що таке успадкування й яка від нього користь?

5.2. Чи можна в похідному класі оголошувати власні поля даних і власні методи?

5.3. Чим захищені поля відрізняються від прихованих?

5.4. Чи сумісні за присвоюванням похідний і базовий класи?

5.5. Які функції класу-батька не успадковуються в класі-нащадку?

5.6. До яких полів дає доступ вказівник на базовий клас, установлений на об'єкт похідного класу?
5.7. Як позначається метод класу-попередника, коли він застосовується до об'єкта класу-нащадка?
5.8. Які конструктори виконуються під час створення об'єкта похідного класу й у якому порядку?
5.9. Які деструктори виконуються перед знищенням об'єкта похідного класу й у якому порядку?
5.10 Які три різновиди успадкування є в мові C++ і як вони позначаються?
5.11. Що визначає специфікатор доступу в оголошенні успадкування?
5.12. Чим відкрите успадкування відрізняється від захищеного, а захищене – від прихованого?
5.13. Який метод називається віртуальним?
5.14. Синтаксичні й семантичні особливості віртуального методу.
5.15. Чи успадковуються віртуальні функції?
5.16. Чи можуть конструктор і деструктор бути віртуальними?
5.17. Що таке зв'язування? Яка програма здійснює зв'язування викликів невіртуальних функцій?
5.18. Чим статичне зв'язування відрізняється від динамічного?
5.19. Як впливають віртуальні функції на розмір класу?
5.20. Навіщо потрібні віртуальні функції?
5.21. Чим об'єкт класу, що має віртуальні функції, відрізняється від об'єкта класу без віртуальних функцій?
5.22. Поняття поліморфізму. Що дає поліморфізм?
5.23. Які два різновиди поліморфізму реалізовані в C++?
5.24. Який клас називають поліморфним?
5.25. Що є засобом реалізації статичного поліморфізму?
5.26. Поясніть різницю між успадкуванням інтерфейсу та успадкуванням реалізації.
5.27. Як оголошується суто віртуальний метод?
5.28. Який клас називається абстрактним і яким є його головне призначення?
5.29. Чи можна успадковувати суто віртуальну функцію?
5.30. Чи можна оголосити деструктор як суто віртуальний?

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 7.
Зв'язані структури даних

7.1. Реалізація стека зв'язаними структурами

У розд. 4 розглядалися абстрактний тип даних (АТД) "стек" і його реалізація в динамічному масиві. Розглянемо іншу реалізацію АТД "стек" – на основі *динамічних зв'язаних структур*, за допомогою яких зображують послідовності та інші, складніші структури.

Послідовність зв'язаних структур

Спочатку розглянемо зв'язану послідовність, або **список структур**, що мають такий вигляд:

Значення	Адреса наступної структури

Структури, тобто елементи, утворюють послідовність, зображену на рис. 7.1. Перший елемент називається *головним елементом*, або *головою*. Поле-вказівник кожного елемента списку (крім останнього) установлено на наступний елемент, тобто наступний "прив'язано" до попереднього. За останнім елементом списку наступного немає, тому його вказівник має значення NULL.

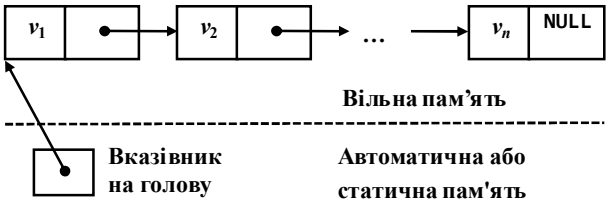

Рис. 7.1. Зв'язаний список рядків

- ✓ Забезпечити доступ до елементів списку можна за допомогою вказівника, який має ім'я в програмі й розташований у статичній або автоматичній пам'яті.
- ✓ Якщо змінити значення вказівника в деякому елементі списку, то можна втратити адресу наступного після нього елемента. Якщо адресу наступного елемента втрачено, то він і всі подальші елементи стають недоступними, тобто "сміттям".

На практиці дані, зображені зв'язаним списком, дуже часто зберігаються в динамічній пам'яті окремо, а елементи списку містять вказівники на ці дані (рис. 7.2).

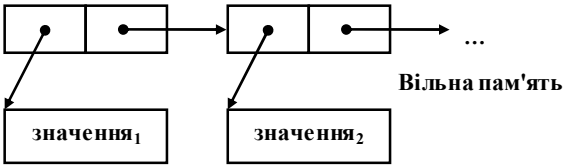

Рис. 7.2. Список вказівників на дані

Структура в списку й клас стеків

Позначимо тип поля даних у структурі, описаній вище, ім'ям `T`, не уточнюючи цього типу. Вказівник у структурі має вказувати на таку саму структуру.

```
struct ListElem            //тип елементів списку
{ T data;
  ListElem * next;
};
```

Елементи стека створюються у вільній пам'яті, тому ніяких обмежень на кількість елементів у стеку немає. Стек зображується вказівником на верхівку – структуру `ListElem`. Методами класу є конструктор без параметрів, деструктор, додавання й вилучення елемента.

```
class TStack {             //тип стеків
private:
  ListElem * top;          //вказівник на верхівку
public:
  TStack();
  ~TStack();
  bool push(T elem);
  bool pop(T & elem);
};
```

Перейдемо до реалізації методів.

Конструктор класу створює порожній стек, установлюючи вказівник на верхівку "у нікуди".

```
TStack::TStack() : top(NULL){};
```

Метод додавання створює новий елемент списку за допомогою операції `new` (за неуспішного створення програма аварійно закінчується). Інакше метод:

– у поле даних нового елемента записує задане значення `elem`,
– "прив'язує" верхівку стека до нового елемента,
– робить його верхівкою, установлюючи на нього вказівник `top` в об'єкті.

```
bool TStack::push(T elem)
{ ListElem * p = new ListElem(elem);
  p->data = elem;
  p->next = top;
  top = p;
  return 1;
};
```

Метод вилучення перевіряє, чи не є стек порожнім. Якщо є, то метод повертає ознаку неуспішності. Інакше:

– зберігає в параметрі значення поля даних верхівки стека,
– установлює на верхівку допоміжний вказівник `t`,
– пересуває вказівник `top` в об'єкті на наступний елемент списку,
– елемент, що був верхнім, знищує за допомогою вказівника `t`.

```
bool TStack::pop(T & elem)
{ if(top == NULL) return 0;
  elem = top->data;
  ListElem * t = top;
  top = top->next;
  delete t;
  return 1;
};
```

Деструктор звільняє пам'ять від усіх елементів стека, для чого викликає метод вилучення, поки вказівник в об'єкті ненульовий.

```
TStack::~TStack()
{ T elem;
  while(top) pop(elem);
}
```

Приховане оголошення імені типу

Для закінчення реалізації уточнимо тип `T` як тип символів. Оголосимо його перед класом `TStack`.

Оголошення типу елементів списку `ListElem` бажано приховати в класі `TStack`, щоб за межами методів класу ім'я типу структур було недоступним. Для цього оголосимо тип `ListElem` *усередині класу* `TStack` як прихований (звісно, перед оголошенням поля `ListElem * top`).

```
typedef char T;
class TStack {
private:
  struct ListElem            //тип елементів списку
  { T data;
    ListElem * next;
  };
  ListElem * top;
  ...                        //решта класу TStack
};
```

Завдяки цьому оголошенню ім'я `ListElem` доступне в класі `TStack` і його методах, але не доступне за їх межами.

7.2. Реалізація черги зв'язаними структурами

Черга – це послідовність, елементи до якої додаються в її кінці, а вилучаються на початку (див. підрозд. 4.3). Реалізуємо клас черг аналогічно класу стеків на основі зв'язаних структур.

Зобразимо чергу кільцевим списком, останній елемент якого вказує на перший, а вся черга представлена вказівником на останній елемент (рис. 7.3). Це дозволяє легко додавати елементи в кінці черги й вилучати на початку.

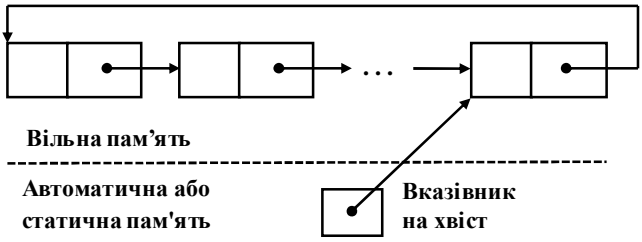

Рис. 7.3. Кільцевий список з вказівником на хвіст

Нехай типом елементів `T` буде тип `double`.

```
typedef double T;
```

У класі оголосимо тип структур `ListElem`, що утворюють зв'язаний список (див. підрозд. 7.1), і вказівник `last` на останній елемент списку. Інтерфейс залишається без змін (див. підрозд. 4.3).

```
class TQueue {
  struct ListElem
  { T data;
    ListElem * next;
  };
  ListElem * last;
public:
  TQueue();
  ~TQueue();
  bool add(T elem);
  bool del(T & elem);
};
```

Конструктор установлює вказівник на кінець черги "у нікуди".

```
TQueue::TQueue() : last(NULL){};
```

Метод додавання створює новий елемент списку (за неуспішності створення програма аварійно закінчується) і далі:

– записує задане значення в поле даних нового елемента,
– якщо черга порожня, то робить новий елемент першим і водночас останнім, тобто прив'язує його до самого себе,
– інакше прив'язує до нового елемента перший елемент черги, а новий елемент – до останнього елемента,
– робить новий елемент останнім.

```
bool TQueue::add(T elem)
{ ListElem * p = new ListElem;
  p->data = elem;
  if(!last)              //якщо список був порожнім,
    p->next = p;         //то зациклюється один,
  else {                 //інакше елемент
    p->next=last->next; //додається перед першим
    last->next = p;     //після останнього
  }
  last = p;            //новий елемент стає останнім
  return 1;
}
```

Метод вилучення перевіряє, чи не є черга порожньою. Якщо є, то метод повертає ознаку неуспішності. Інакше він:

– отримує доступ до першого елемента черги,
– значення поля даних цього елемента присвоює параметру,
– якщо перший елемент був останнім, то встановлює вказівник в об'єкті "у нікуди",
– інакше до останнього елемента прив'язує елемент, наступний за першим,
– знищує елемент, що був першим.

```
bool TQueue::del(T & elem)
{ if (last==NULL)
    return 0;
  ListElem * p = last->next;
  elem = p->data;
  if(last==last->next)   //елемент був останнім
    last=NULL;
  else                   //елемент не був останнім
    last->next = p->next;
  delete p;
  return 1;
}
```

Деструктор звільняє пам'ять від усіх елементів черги, для чого викликає метод вилучення, поки вказівник в об'єкті ненульовий.

```
TQueue::~TQueue()
{ T elem;
  while(last) del(elem);
}
```

Приклад. Програма вводить із файлу дійсні числа, зберігає в двох чергах від'ємні та додатні числа, ігноруючи нулі, і виводить на екран спочатку від'ємні числа (у порядку введення), а потім – додатні.

```
int main(){
  TQueue qNegat, qPosit;
  T n;
  ifstream f("inQReals.txt");
  if (f.fail())
    { cout << "input file is absent\n";
      return 1;
    }
```

```
  while (f >> n)
  { if(n<0)
      qNegat.add(n);
    else if(n>0)
      qPosit.add(n);
  }
  cout << "NEGATIVE NUMBERS\n";
  while(qNegat.del(n))
    cout << n << ' ';
  cout << "\nPOSITIVE NUMBERS\n";
  while(qPosit.del(n))
    cout << n << ' ';
  return 0;
}
```

7.3. Поняття зв'язаного списку

Зв'язаний список рядків

Задача. Текст містить прізвища, які можуть повторюватися. Потрібно прочитати текст і вивести прізвища по одному разу в порядку їх першої появи в тексті.

Позначимо потік, зв'язаний з текстом, ім'ям `f`, новий рядок – `s`, а послідовність збережених рядків – `list` (*list* – список). Спочатку `list` порожній – позначимо це символами `<>`. Алгоритм роботи очевидний:

```
підготувати f до обробки;
list = <>;
while (f >> s)
  if (!(list містить s))
    додати s до list;
вивести рядки з list.
```

З цього алгоритму неважко виділити операції зі списком:

- створити порожній список,
- визначити, чи містить список заданий рядок,
- додати рядок до списку,
- вивести рядки зі списку.

Проект класу списків і його використання

Умова задачі не обмежує кількості елементів у списку `list`, тому зобразимо його за допомогою зв'язаних структур (див. підрозд. 7.1) і розробимо клас `TList`. Створити порожній список при-

родно конструктором, інші операції реалізуємо як відкриті методи. Додамо деструктор, що звільняє пам'ять із-під елементів списку.

Даними в об'єкті класу `TList` є два вказівники – на перший і останній елементи списку. Вказівник на перший елемент дає доступ до початку списку, а вказівник на останній дозволяє додати елемент у кінці списку. Додавати потрібно саме в кінці, адже, за умовою, рядки виводяться в порядку їх появи в тексті.

У файлі, наприклад `TList.h`, оголосимо ім'я типу даних `T`, що зберігаються в елементах списку. Як і в розробці класів стеків і черг, ім'я типу елементів списку `ListElem` приховаємо всередині класу `TList`.

```
#include <string>
using namespace std;
typedef string T;
class TList {
  struct ListElem  //прихований тип елементів списку
  { T data;
    ListElem * next;
  };
  ListElem * first, * last;
public:
  TList();
  ~TList();
  bool contains(T & s); //чи є рядок у списку
  bool append(T & s);   // додати рядок
  void output();        // вивести рядки списку
};
```

Реалізуємо методи цього класу нижче. Проте зафіксований інтерфейс класу `TList` уже дозволяє записати розв'язання задачі.

Напишемо головну функцію, яка реалізує алгоритм розв'язання задачі за допомогою методів класу `TList`. Припустимо, що вхідні дані беруться з файлу з ім'ям `inSL.txt`.

```
#include <iostream>
#include <fstream>
#include "TList.h"
int main(){
  TList list;               // об'єкт-список
  T s;                      // рядок, що вводиться
  ifstream f("inSL.txt");   // вхідний потік
  if (f.fail())
    { cout << "input file is absent\n";
```

```
      return 1;
    }
  while (f >> s)
    if( !(list.contains(s)) )
      if( !(list.append(s)) )
        { cout << "append failed\n";
          return 1;
        }
  list.output();
  return 0;
}
```

Реалізація операцій із списком

Запишемо реалізацію методів класу `TList` у файлі `TList.cpp`. Спочатку включимо необхідні файли.

```
#include <iostream>
#include <fstream>
#include "TList.h"
```

Конструктор. Конструктор `TList` дуже простий – вказівники на перший і останній елементи списку отримують значення `NULL`.

```
TList::TList()
  : first(NULL), last(NULL) {};
```

Додавання. Метод додавання повертає, фактично, кількість доданих елементів, тобто `1` (за неуспішного створення елемента програма аварійно завершується). Спочатку створюємо новий елемент і встановлюємо на нього допоміжний вказівник `pElem` (див. рис. 7.4, *а*). Можливі дві різні ситуації – список перед додаванням або порожній, або ні. Якщо порожній, то новий елемент стає і першим, і останнім у списку – на нього встановлюються вказівники `first` і `last` (рис. 7.4, *б*). Інакше "прив'язуємо" його до елемента, який поки що є останнім (його ідентифікує вказівник `last` в об'єкті, а його поле `next` – вираз `last->next`). Після цього вказівник `last` установлюємо на новий елемент (рис. 7.4, *в*).

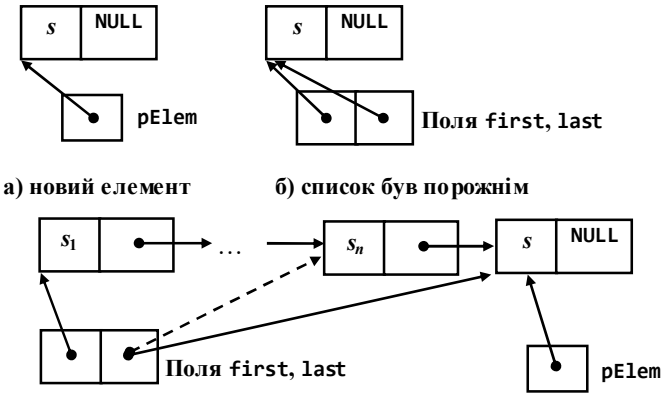

Рис. 7.4. Додавання елемента до списку

Метод append реалізує описані дії.

```
bool TList::append(T & s)
{ ListElem * pElem = new ListElem;
  pElem->data=s; pElem->next=NULL;
  if(last==NULL)              // список порожній
    { last = first = pElem; }
  else                        // список не порожній
    { last->next = pElem;
      last = pElem;
    }
  return 1;
};
```

Визначення, чи містить список заданий рядок. Щоб визначити, чи містить список заданий рядок, починаємо з голови списку й рухаємося елементами списку, поки список не закінчився й не знайшовся елемент, що представляє заданий рядок. Якщо елемент знайшовся, то результатом є "так". Інакше список закінчується, і результатом є "ні".

Розглянемо рух елементами списку. Якщо вказівник p установлено на елемент списку, то перехід до наступного елемента задає вираз p=p->next. Умовою того, що вказівник установлено

на якийсь елемент списку, є вираз p!=NULL або просто p. Заданий рядок є значенням параметра s, тому вираз

```
p && p->data != s
```

є умовою того, що *списком можна й треба рухатися*. Рух списком припиняється, коли або p==NULL, або значення p->data й заданий рядок збіглися, тому умова p!=NULL є ознакою того, що рядок у списку є.

```
bool TList::contains(T & s)
{ ListElem * p = first;
  while(p && p->data != s)
      p = p->next;   //перейти на наступний елемент
  return (p != NULL);
};
```

Виведення. Вивести рядки, представлені послідовними елементами списку, можна, рухаючися від початку списку.

```
void TList::output()
{ ListElem * p = first;
  while(p != NULL)
    { cout << p->data << '\n';
      p = p->next;   //перейти на наступний елемент
    }
  return;
};
```

Деструктор. Деструктор має знищити всі елементи списку. Для цього знищимо перший елемент списку, зробимо наступний за ним першим, так само знищимо його і так далі, поки список не стане порожнім. Щоб знищити перший елемент списку, необхідно:

– установити на нього допоміжний вказівник, наприклад t,
– переставити вказівник first на наступний елемент,
– знищити перший елемент, використовуючи t.

```
TList::~TList()
{ while(first != NULL)      // поки є що знищувати
    { ListElem * t = first;
      first = first->next;
      delete t;             // знищити елемент
    }
  last=NULL;
};
```

Вправи

7.1. Написати метод класу `TList`, аналогічний `append`, який додає елемент не в кінці списку, а на його початку.

7.2. Додати до класу `TList` метод уведення рядків з файлу, параметризований шляхом до файлу у файловій системі.

7.3. Написати клас елементів зв'язаного списку, що містять вказівники на структуру з даними про студента (прізвище та цілий сумарний рейтинг) і на наступний елемент списку. Конструктор класу, параметризований даними про студента, створює динамічну структуру з даними й установлює на неї вказівник в об'єкті. Деструктор знищує структуру, на яку встановлено вказівник в об'єкті. Метод виведення виводить дані про студента в потік виведення, заданий параметром.

7.4. Написати клас `ListStud`, аналогічний класу `TList`, для зв'язаних списків, елементи яких є об'єктами класу з попередньої вправи. Конструктор створює порожній список. Деструктор знищує всі елементи списку. Метод додавання, параметризований даними про студента, додає новий елемент у кінці списку. Метод вилучення знищує головний елемент списку й повертає вказівник на структуру з даними про студента, на яку вказував знищений елемент (або повертає `NULL`, якщо список порожній). Метод уведення, параметром якого є шлях до файлу, уводить дані про студентів і додає елементи до списку (без перевірки, чи вже містяться ці дані в списку). Метод виведення виводить дані про студентів рядками в потік виведення, заданий параметром.

7.4. Вилучення зі списку

Окремо розглянемо один з кількох можливих варіантів вилучення елемента зі списку – знищити елемент, що зображує задане значення. Розглянемо цю операцію на прикладі списку, що зображує рядки, і реалізуємо її методом `delStr`, який додамо до класу `TList`. Метод має вилучити елемент, що зображує рядок, заданий параметром методу (якщо такий елемент є).

Розглянемо алгоритм операції вилучення.

1. Порожній список залишаємо без змін.

2. Якщо поле `data` в голові списку має задане рядкове значення, то потрібно *вилучити голову списку*. Для цього встановлюємо на голову додатковий вказівник `t` й пересуваємо вказівник `first` в об'єкті на наступний елемент (рис. 7.5, *а*, *б*). Якщо наступного елемента немає, то список стає порожнім, тому змінюємо вказівник `last` в об'єкті (рис. 7.5, *в*). Нарешті, знищуємо елемент, на який указує `t`.

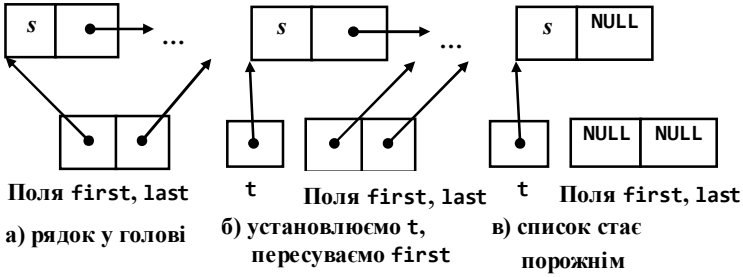

Рис. 7.5. Знищення голови списку

3. Якщо в голові списку задане значення відсутнє, то за допомогою вказівника `p` рухаємося елементами списку, поки або не пройдемо список, або не натрапимо на елемент *P*, *наступний* за яким елемент *T* представляє заданий рядок. Якщо цього *T* немає, то список залишається без змін. Якщо цей елемент *T* є, то встановимо на нього допоміжний вказівник `t`, а елемент, наступний за *T*, прив'яжемо до *P* (рис. 7.6, *а*). Тільки після цього знищимо *T* (за допомогою вказівника `t`). Якщо знищуваний елемент *T* був останнім у списку, то останнім стає *P* (рис. 7.6, *б*).

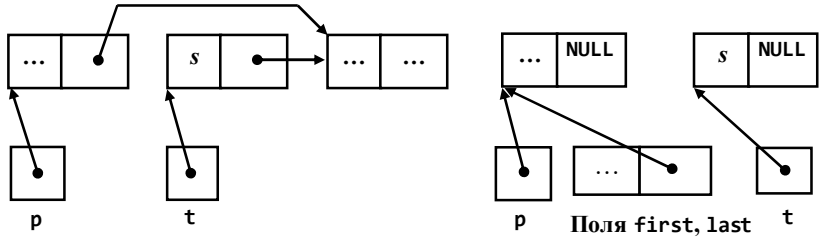

Рис. 7.6. Зв'язаний список рядків

Реалізуємо наведений алгоритм у методі del, який повертає ознаку того, що елемент дійсно вилучався. Додамо його прототип до класу TList:

```
bool del(T & s);
```

Реалізуємо його за межами класу.

```
bool TList::del(T & s) {
  if(first == NULL) return 0;
  if(first->data == s)    //вилучаємо голову списку
    { ListElem * t = first;
      first = first->next;
      if(first == NULL) last = NULL;
      delete t;
      return 1;
    }
  //шукаємо елемент, наступний за яким
  //містить заданий рядок
  ListElem * p = first;
  while(p->next && p->next->data != s )
    p = p->next;
  if(p->next==NULL)     //наступного не знайшли -
    return 0;           //список не змінюється
  // знайшли наступний – елемент *(p->next)
  ListElem * t=p->next;
  p->next = t->next;
  if(p->next == NULL) last = p;
  delete t;
  return 1;
}
```

Вправи

7.5. До класу ListStud (див. вправи до попереднього підрозділу) додати метод вилучення елемента, що представляє студента із заданим прізвищем.

7.6. До класу ListStud (див. вправи до попереднього підрозділу) додати метод, що вилучає елемент списку, заданий його адресою, і повертає адресу елемента, наступного за вилученим. Написати програму, яка створює список студентів за даними з файлу, викреслює з нього студентів із середнім балом менше заданого й виводить решту в новий файл.

7.5. Структури з двома зв'язками

На практиці інколи використовуються списки зв'язаних структур, що містять не один, а два вказівники: на наступний і на попередній елементи списку. Такі списки називають **двобічно зв'язаними** (рис. 7.7).

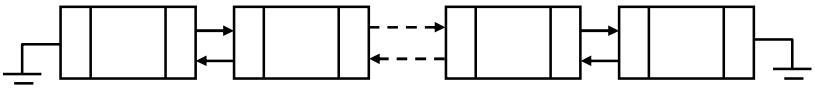

Рис. 7.7. Двобічно зв'язаний список

Двобічно зв'язані списки зручні, коли умови задачі вимагають руху списком як від його початку до кінця, так і від кінця до початку. Також двобічно зв'язаний список природно реалізує тип даних **дек** (англ. *deq – double-ended queue*), або чергу з двома кінцями. Це послідовність, в якій елементи додаються й вилучаються з обох кінців.

Техніка роботи з двобічно зв'язаними списками аналогічна такій зі списками з одним зв'язком, лише код громіздкіший, оскільки необхідно додатково відстежувати значення полів із другими зв'язками. Докладніше ці списки тут не розглядаються.

Кореневі бінарні дерева та арифметичні вирази

Структури з двома зв'язками використовуються для зображення кореневих бінарних дерев, що мають надзвичайно широке практичне застосування.

Кореневе орієнтоване впорядковане дерево складається з **вузлів** і **дуг** (рис. 7.8). Один з вузлів є **коренем**; від нього ведуть дуги до інших вузлів (**дітей** кореня), від них – до інших тощо. **Нащадки** вузла – це всі вузли піддерева, коренем якого є цей вузол. **Листок** – це вузол, з якого не виходять дуги; інші вузли називаються **проміжними**.

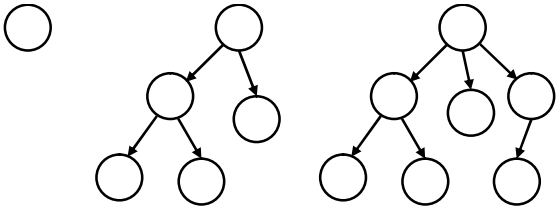

Рис. 7.8. Приклади кореневих орієнтованих упорядкованих дерев (діти вузлів упорядковані зліва направо)

✓ Діти вузла вважаються *упорядкованими*: зміна порядку дітей будь-якого вузла дає інше дерево.

Кореневі орієнтовані впорядковані дерева мають *рекурсивну структуру*: дно рекурсії – це дерево, корінь якого є листком, а будь-яке інше дерево має корінь, від якого ведуть дуги до менших дерев. Обробка дерев описується переважно також рекурсивними підпрограмами.

Бінарне дерево – це дерево, кожен вузол якого має не більше двох дітей.

Розглянемо роботу з кореневими бінарними деревами в елементарній задачі, набагато складніші варіанти якої розв'язуються на практиці під час трансляції.

Задача. Простий арифметичний вираз – це або цифра, яка задає натуральне число від 0 до 9, або запис, утворений двома арифметичними виразами зі знаком операції +, – або * між ними й узятий у дужки. Пробіли між символами виразу не допускаються. Наприклад, записи `1`, `(1+2)`, `((2–7)*(9+4))` є простими арифметичними виразами, `(1)`, `1+2`, `(1+2-3)` – ні. У текстовому файлі записано деякі символи. Якщо вони утворюють простий арифметичний вираз, то потрібно вивести значення цього виразу, а інакше повідомити про помилку.

На початку розв'язання задачі опишемо структуру простих арифметичних виразів за допомогою такої системи БНФ:

```
<Expr> ::= <Digit> | '('<Expr> <Oper> <Expr> ')'
<Digit> ::= '0' | '1' | … | '9'
<Oper> ::= '+' | '-' | '*'
```

На основі цієї системи БНФ напишемо функцію, яка розпізнає, чи має послідовність символів у рядку потрібну структуру, і будує бінарне дерево, що зображує вираз.

Виразу відповідає **семантичне дерево**, вузли якого позначені цифрами або знаками операцій. Найменшому виразу, тобто цифрі, відповідає листок, позначений цифрою; виразу, утвореному двома підвиразами зі знаком операції між ними, – дерево, корінь якого має знак операції як мітку й дуги до коренів дерев, що відповідають підвиразам (рис. 7.9).

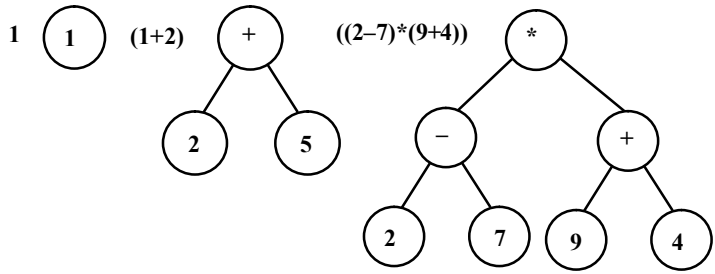

Рис. 7.9. Вирази та бінарні семантичні дерева

Загальний алгоритм розв'язання. Спочатку за виразом у файлі будуємо семантичне дерево. Якщо вираз правильний, то дерево має хоча б один вузол, інакше робимо його порожнім. Якщо дерево побудовано й у тексті більше немає символів, то вираз правильний, і тоді за допомогою дерева обчислюємо значення виразу, інакше повідомляємо про помилку у виразі.

Ієрархія класів для вузлів дерева

Реалізуємо наведений алгоритм за допомогою системи класів для вузлів дерева. Вузли є *різнотипними*: листки зберігають цілі числа, проміжні вузли – знаки операцій. Типи вузлів зобразимо за допомогою *ієрархії класів*.

Базовий клас. Клас `TreeNode` є *абстрактним*: він лише задає інтерфейс, спільний для всіх класів вузлів, а конкретні типи вузлів уточнюються в класах-нащадках. Означимо такі спільні операції:

- вивести мітку вузла,
- вивести вираз, зображений деревом з коренем у вузлі,
- обчислити вираз, зображений деревом з коренем у вузлі,
- знищити вузол (операція-деструктор).

У розв'язанні задачі об'єкти похідних класів мають бути вузлами дерева, які обробляються за допомогою вказівників на базовий клас, тому методи інтерфейсу оголосимо *віртуальними*.

До вказаних методів додамо конструктор копії та оператор присвоювання. Зробимо їх прихованими, щоб запобігти неглибокому копіюванню об'єктів похідних класів.

```
class TreeNode {
  TreeNode(const TreeNode&);
  TreeNode& operator = (const TreeNode&);
public:
  virtual void output() = 0;
  virtual void outputObject() = 0;
  virtual int calculateObject() = 0;
  virtual ~TreeNode();
};
```

Деструктор не є суто віртуальним методом, тому означимо його. В абстрактному класі його дія порожня.

```
TreeNode:: ~TreeNode(){}
```

Клас для листків. Для листків дерева означимо клас `IntNode`, об'єкти якого містять поле `number` для цілого значення.

```
class IntNode: public TreeNode {
  int number;
public:
  IntNode(int n);
  void output();
  void outputObject();
  int calculateObject();
};
```

Реалізувати методи цього класу можна таким чином:

```
IntNode::IntNode(int n): number(n) {}
void IntNode::output() { cout<<number; }
void IntNode::outputObject() { output(); }
int IntNode::calculateObject() { return number; }
```

Об'єкти цього класу не містять вказівників, тому деструктор залишається стандартним.

Класи для проміжних вузлів. Для вузлів з бінарними операціями означимо *абстрактний клас* `BinOpNode` і успадкуємо його в *конкретних похідних класах* `PlusBinOpNode`, `MinusBinOpNode`, `MultBinOpNode`, відповідних до операцій `+`, `–`, `*`

із цілими числами. Ці операції є допоміжними в обчисленні виразу, тому для них у класі BinOpNode оголосимо *прихований суто віртуальний* метод calculate і реалізуємо його в похідних класах як додавання, віднімання та множення.

Кожен проміжний вузол має два піддерева; відповідно, об'єкт *кожного* із зазначених похідних класів містить два вказівники на об'єкти, що зображують корені піддерев. Оголосимо вказівники в базовому класі BinOpNode і успадкуємо їх у похідних класах. Об'єкти, що зображують корені піддерев, можуть належати будь-якому з класів, похідних від TreeNode, тому типом вказівників є TreeNode*.

```
class BinOpNode: public TreeNode {
  TreeNode *left, *right;
  virtual int calculate(int arg1, int arg2) = 0;
public:
  BinOpNode(TreeNode *l, TreeNode *r);
  void outputObject();
  int calculateObject();
  virtual ~BinOpNode();
};
```

Реалізуємо методи цього класу, спільні для похідних класів. Конструктор установлює вказівники нового об'єкта на об'єкти, задані параметрами – вказівниками на TreeNode. Виведення виразу, обчислення виразу та знищення вузла в деструкторі разом з усіма вузлами-нащадками відбуваються рекурсивно.

```
BinOpNode::BinOpNode(TreeNode *l, TreeNode *r):
                   left(l), right(r) {
  //піддерева немає - додаємо листок для виразу 0
  if (left==NULL) left=new IntNode(0);
  if (right==NULL) right=new IntNode(0);
}
int BinOpNode::calculateObject() {
  return calculate(left->calculateObject(),
                   right->calculateObject());
}
void BinOpNode::outputObject() {
  cout<<"("; left->outputObject(); cout<<" ";
  output(); cout<<" ";
  right->outputObject(); cout<<")";
}
```

```
BinOpNode::~BinOpNode() {
  if (left != NULL)
    {delete left; left = NULL;}
  if (right != NULL)
    {delete right; right = NULL;}
}
```

Класи `PlusBinOpNode`, `MinusBinOpNode`, `MultBinOpNode`, похідні від `BinOpNode`, відрізняються лише конструкторами, тому наведемо тільки один з них.

```
class PlusBinOpNode: public BinOpNode{
  int calculate(int arg1, int arg2);
public:
  PlusBinOpNode(TreeNode *l, TreeNode *r);
  void output();
};
```

Класи `MinusBinOpNode`, `MultBinOpNode` замість конструктора

```
PlusBinOpNode(TreeNode *l, TreeNode *r);
```

мають, відповідно, такі методи:

```
MinusBinOpNode(TreeNode *l, TreeNode *r);
MultBinOpNode(TreeNode *l, TreeNode *r);
```

Операції `output` і `calculate` у похідних класах виконуються по-різному, тому реалізуються окремо по класах. Наведемо лише конструктор і методи класу `PlusBinOpNode`.

```
PlusBinOpNode::PlusBinOpNode(
  TreeNode *l, TreeNode *r): BinOpNode(l,r){}
int PlusBinOpNode::calculate(int arg1, int arg2) {
  return arg1+arg2;
}
void PlusBinOpNode::output() { cout<<"+"; }
```

Однойменні методи класів `MinusBinOpNode` і `MultBinOpNode` на місці знака + мають знаки – і *.

Для прикладу за допомогою класів наведеної ієрархії створимо дерева з рис. 7.9.

```
TreeNode *root1 = new IntNode(1);
TreeNode *root2 = new PlusBinOpNode(
  new IntNode(2),new IntNode(5));
TreeNode *root3 = new MultBinOpNode(
  new MinusBinOpNode(new IntNode(2),new IntNode(7)),
  new PlusBinOpNode(new IntNode(9),new IntNode(4)));
```

Клас для створення семантичного дерева

Розглянемо, як за арифметичним виразом, записаним у текстовому файлі, створити відповідне семантичне дерево. Ця задача є типовим прикладом лексичного та синтаксичного аналізу. Для створення дерева розробимо клас `TreeConstructor`, конструктор якого відкриває потік для введення з файлу, а метод `makeTree` зчитує файл і створює дерево. Передбачимо перевірку синтаксичних помилок у вхідному записі й додамо метод `getErrMessage`, що діагностує помилки. Об'єкт класу працює з потоком, тому додамо деструктор, що закриває потік.

Поточний стан уведення та аналізу тексту з виразом зобразимо кількома атрибутами. Текст уводимо за допомогою вхідного потоку `f`, а поточна позиція в потоці є значенням цілого поля `curPos`. Ознаку того, що помилок у виразі ще не було, зберігає булеве поле `isOK` (поява помилки має блокувати подальшу роботу). Повідомлення про помилку утворюється в рядковій змінній `errMessage`.

Для наочності лексичного аналізу вхідні символи поділимо на типи, яким дамо відповідні імена.

`Digit` – цифра	`LBracket` – ліва дужка
`PlusSign` – знак +	`RBracket` – права дужка
`MinusSign` – знак -	`Eof` – кінець тексту
`MultSign` – знак *	`Other` – інший символ
`White` – пробіл	

Означимо ці імена в переліку `enum SymbolTypes`.

Додамо методи, допоміжні до аналізу виразу та створення дерева. Метод із заголовком

```
void readNext(char &c, SymbolTypes &ct)
```

уводить із потоку наступний символ і визначає його тип. Метод із заголовком

```
SymbolTypes getSymbolType(char c)
```

визначає й повертає тип символу. Метод із заголовком

```
bool isBinOpSign(SymbolTypes ct)
```

перевіряє, чи відповідає тип символу знаку бінарної операції.

Основну роботу зі створення дерева проводить допоміжна рекурсивна функція з таким прототипом:

```
TreeNode* makeTree0();
```

Вона повертає адресу створеного дерева, або NULL, якщо вираз помилковий.

Отже, клас для побудови дерева має вигляд

```
class TreeConstructor{
  std::ifstream f;
  int curPos;
  bool isOK;
  std::string errMessage;
  enum SymbolTypes {Other, Digit, LBracket, RBracket,
       White, PlusSign, MinusSign, MultSign, Eof};
  bool isBinOpSign(SymbolTypes ct);
  SymbolTypes getSymbolType(char c);
  void readNext(char &c, SymbolTypes &ct);
  TreeNode* makeTree0();
public:
  TreeConstructor(char *fName);
  ~TreeConstructor();
  TreeNode* makeTree();
  std::string getErrMessage();
};
```

Реалізація методів

Конструктор отримує ім'я файлу та ініціалізує атрибути. Якщо при цьому потік не відкрито, то формується ознака помилки та повідомлення про неї.

```
TreeConstructor::TreeConstructor(char *fName):
  f(fName), curPos(0), isOK(true), errMessage("") {
  if (f.fail()) {
    isOK=false;
    errMessage="Couldn't open file " + string(fName);
  }
}
```

Деструктор закриває відкритий потік.

```
TreeConstructor::~TreeConstructor() {
  if (f.is_open()) f.close();
}
```

Метод створення дерева повертає адресу дерева або NULL, якщо трапилася помилка. Аналіз виразу й формування дерева цей метод перекладає на прихований метод makeTree0(), а по-

тім перевіряє, чи немає зайвого символу після виразу. Якщо він є, то дерево знищується й метод повертає NULL.

```
TreeNode* TreeConstructor::makeTree(){
  char c; SymbolTypes ct;
  TreeNode *res=makeTree0();
  readNext(c,ct);
  if (ct!=Eof) {
    delete res; res=NULL;
    errMessage="Extra symbol: position " +
               int2str(curPos);
    isOK=false;
  }
  return res;
}
```

У методі makeTree0() реалізовано наведений нижче алгоритм.

Отримуємо з тексту поточний символ. За умовою, вираз може починатися або цифрою, або дужкою. Якщо це цифра, то створюємо вузол-листок і повертаємо його адресу. Якщо це дужка, то далі мають бути вираз (*лівий підвираз*), знак операції та ще один вираз (*правий підвираз*).

Лівий підвираз обробимо рекурсивно й збережемо адресу кореня утвореного дерева (*лівого піддерева*) у вказівнику left.

Знак операції визначає тип кореневого вузла дерева, який має створюватися після обробки правого підвиразу й правої дужки, тому після зчитування збережемо його та його тип.

Правий підвираз також обробимо рекурсивно й збережемо адресу кореня *правого піддерева* у вказівнику right.

Вираз має закінчуватися дужкою – після її зчитування створюємо кореневий вузол дерева, синами якого стають корені лівого й правого піддерев, побудованих рекурсивно.

Якщо який-небудь з отриманих символів не відповідає синтаксису виразів, то знищуємо всі раніше створені піддерева й повертаємо NULL.

```
TreeNode* TreeConstructor::makeTree0(){
  char c; SymbolTypes ct,ctOp;
  readNext(c,ct);        // має бути цифра або '(':
  if (!(ct==LBracket || ct==Digit)) {
    isOK=false;
    errMessage="Digit or ( was expected: position "
               + int2str(curPos);
    return NULL;
  }
```

```
  if (ct==Digit)                // отримано цифру:
    return new IntNode(c-'0');
  TreeNode *left=makeTree0();   // отримано '(':
  if (left==NULL) return NULL;
  readNext(c,ctOp);       // має бути знак операції
  if (!isBinOpSign(ctOp)) {
    delete left;
    isOK=false;
    errMessage="+, -, * was expected: position " +
               int2str(curPos);
    return NULL;
  }
  TreeNode *right = makeTree0();
  if (right==NULL)
    {delete left; return NULL;}
  readNext(c,ct);                  // має бути ')'
  if (ct!=RBracket) {
    delete left; delete right;
    isOK=false;
    errMessage=") was expected: position " +
               int2str(curPos);
    return NULL;
  }
  switch (ctOp){
    case PlusSign:
      return new PlusBinOpNode(left,right);
    case MinusSign:
      return new MinusBinOpNode(left,right);
    case MultSign:
      return new MultBinOpNode(left,right);
    default:
      isOK=false; errMessage="strange ...";
      delete left; delete right;
      return NULL;
  }
}
```

У цьому методі використовуються допоміжні функції.

Прихована функція введення наступного символу `readNext` уводить символ з потоку й повертає його та його тип за допомогою параметрів-посилань. Якщо трапилася помилка введення, то

символ і тип отримують значення '\0' і Other, а поле isOK в об'єкті – false.

```
void TreeConstructor::
readNext(char &c, SymbolTypes &ct) {
  if (!isOK) return;
  ++curPos;
  if (f.get(c)) {
    ct=getSymbolType(c);
  } else {
    if (f.eof()) ct=Eof;
    else { ct=getSymbolType('\0'); isOK=false;}
  }
}
```

Допоміжна функція getSymbolType за символом визначає й повертає його тип – константу з переліку SymbolTypes. Типом стороннього символу стає Other.

```
TreeConstructor::SymbolTypes
TreeConstructor::getSymbolType(char c) {
  if (isdigit(c)) return Digit;
  if (c=='(') return LBracket;
  if (c==')') return RBracket;
  if (c==' ' || c==13 || c==10) return White;
  if (c=='+') return PlusSign;
  if (c=='-') return MinusSign;
  if (c=='*') return MultSign;
  return Other;
}
```

Допоміжна функція isBinOpSign отримує тип символу й повертає ознаку того, що це один з типів знаків операцій.

```
bool TreeConstructor::isBinOpSign (SymbolTypes ct) {
  return ct==PlusSign || ct==MinusSign ||
         ct==MultSign;
}
```

Нарешті, допоміжна функція int2str за цілим числом утворює рядок з цифрами й повертає його. Вона не є членом класу й записується окремо.

```
string int2str(int n) {
  const int bufsize=32;
  char buffer[bufsize];
  sprintf_s(buffer, bufsize, "%d", n);
  return string(buffer);
}
```

Метод getErrMessage() повертає повідомлення про помилку, сформоване в об'єкті, або повідомлення, що все гаразд, якщо помилки не було.

```
string TreeConstructor::getErrMessage() {
  if (isOK) return "All is OK";
  return errMessage;
}
```

Розв'язання задачі обчислення виразу

Запишемо клас treeNode та похідні від нього класи у файл заголовків з ім'ям treeNode.h, клас makeTree – у файл makeTree.h у папці з програмою, яку створимо для розв'язання задачі. Реалізацію методів цих класів запишемо у відповідні файли treeNode.cpp та makeTree.cpp. У текстовий файл з ім'ям tree1.txt запишемо арифметичний вираз. До проекту в середовищі програмування включимо вказані вище файли заголовків, файли реалізації та файл із такою головною функцією:

```
#include "treeNode.h"
#include "makeTree.h"
#include <iostream>
using namespace std;
int main() {
  TreeConstructor tc("tree1.txt");
  TreeNode *tr=tc.makeTree();
  cout<<tc.getErrMessage()<<endl;
  if (tr!=NULL){
    cout << "Object: ";
    tr->outputObject();
    cout << endl;
    cout<<"Result: ";
    cout << tr->calculateObject() << endl;
    delete tr;
  }
  return 0;
}
```

Якщо під час уведення виразу та побудови дерева трапилася помилка, то програма виводить відповідне повідомлення. Якщо дерево успішно побудовано, то у виклику tr->outputObject() виводиться вираз, а потім значення виразу, отримане з виклику tr->calculateObject().

Вправи

7.7. Описати АТД "дек" і реалізувати його в класі `DEQue` на основі двобічно зв'язаних списків. За його допомогою розв'язати таку задачу. Текст містить слова та пробіли між ними. Увести текст і визначити, чи збігається його перше слово з останнім, друге – з передостаннім тощо.

7.8. Написати та реалізувати класи `MinusBinOpNode` і `MultBinOpNode`.

7.9. Модифікувати класи вершин, додавши можливість глибокого копіювання семантичних дерев.

7.10. Модифікувати клас `TreeConstructor`, відокремивши помилки введення-виведення від синтаксичних помилок.

7.11. Модифікувати клас `TreeConstructor`, дозволивши зчитувати дерево не тільки з текстового дискового файлу, але й з консолі та рядка. *Вказівка.* Зменшити дублювання коду можна за рахунок ієрархічних зв'язків класів вхідних потоків і використання вхідного потоку класу `istream`.

7.12. Модифікувати клас `TreeConstructor`, дозволивши пробіли в запису дерева.

7.13. Змінити клас `TreeConstructor`, дозволивши пробіли у виразах і довільні числа типу `int` як операнди.

7.14. Розв'язати задачу обчислення значення виразу, додавши до можливих операцій у виразі операції ділення `/` та `%`. *Вказівка*. Варто змінити прототипи `calculate` і `calculateObject`, щоб забезпечити діагностику помилок.

7.15. До попередньої задачі обчислення виразу додайте як можливі операції піднесення до квадрата та "унарний мінус". *Вказівка*. Аналогічно класу вузлів для бінарних операцій додайте загальний клас вузлів для унарних операцій.

Контрольні запитання

7.1. Як зв'язуються між собою послідовні структури?

7.2. Опишіть відмінності в реалізації черги та стека за допомогою зв'язаних послідовностей.

7.3. Чому циклічний однозв'язний список зазвичай ідентифікують вказівником не на початку, а в кінці?

7.4. Опишіть операції, які краще (за швидкістю виконання) реалізовуються за допомогою двозв'язних списків порівняно з однозв'язними.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Вступ до програмування мовою С++. Організація обчислень / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К.: ВПЦ "Київський університет", 2012.

2. Прата С. Язык программирования С++ (С++11). Лекции и упражнения / С. Прата. – М.: Вильямс, 2014.

3. Саттер Г. Стандарты программирования на С++ : серия "C++ In-Depth" / Г. Саттер, А. Александреску. – М.: Вильямс, 2008.

4. Седжвик Р. Алгоритмы на С++. Фундаментальные алгоритмы и структуры данных на С++ / Р. Седжвик. – М.: Вильямс, 2013.

5. Страуструп, Б. Программирование: принципы и практика использования С++ / Б. Страуструп. – М.: Вильямс, 2012.

6. Шилдт Г. С++: базовый курс / Г. Шилдт. – М.: Вильямс, 2014.

7. Шилдт Г. С++: методики программирования Шилдта / Г. Шилдт. – М.: Вильямс, 2008.

8. International Standard ISO/IEC 14882:2014(E) – Programming Language C++ : [Електронний ресурс]. – Режим доступу: https://isocpp.org/std/the-standard.

9. ISO International Standard ISO/IEC 14882:2012(E) – Programming Language C++: [Електронний ресурс]. – Режим доступу: http://en.wikipedia.org/wiki/C++11.

ПРЕДМЕТНИЙ ПОКАЖЧИК

ЗМІСТ

Навчальне видання

Карнаух Тетяна Олександрівна
Коваль Юрій Віталійович
Потієнко Михайло Валерійович
Ставровський Андрій Борисович

ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ С++

ОРГАНІЗАЦІЯ ДАНИХ

Навчальний посібник

Редактор Н. Земляна

Оригінал-макет виготовлено Видавничо-поліграфічним центром "Київський університет"

Формат 60x84$^{1/16}$. Ум. друк. арк. 8,83. Наклад 100. Зам. № 215-7423.
Гарнітура Times New Roman. Папір офсетний. Друк офсетний. Вид. № К5.
Підписано до друку 25.06.15

Видавець і виготовлювач
Видавничо-поліграфічний центр "Київський університет"
01601, Київ, б-р Т. Шевченка, 14, кімн. 43
☎ (38044) 239 32 22; (38044) 239 31 72; тел./факс (38044) 239 31 28
e-mail: vpc_div.chief@univ.kiev.ua
http: vpc.univ.kiev.ua
Свідоцтво суб'єкта видавничої справи ДК № 1103 від 31.10.02