КИЇВСЬКИЙ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Р. А. Веклич
Т. О. Карнаух
А. Б. Ставровський

ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ С++

СТРУКТУРИ ДАНИХ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Р. А. Веклич
Т. О. Карнаух
А. Б. Ставровський

ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ С++

СТРУКТУРИ ДАНИХ

Навчальний посібник

КИЇВСЬКИЙ УНІВЕРСИТЕТ 2018

УДК 004.43(075.8)
В26

Рецензенти:
канд. фіз.-мат. наук С. В. Єфіменко,
канд. фіз.-мат. наук Є. О. Іванов

Рекомендовано до друку вченою радою
факультету комп'ютерних наук та кібернетики
(протокол № 2 від 10 жовтня 2017 року)

Ухвалено науково-методичною радою
Київського національного університету імені Тараса Шевченка
(протокол № 3-17/18 н. р. від 19 березня 2018 року)

Веклич Р. А.

В26 Вступ до програмування мовою C++ . Структури даних : навч. посіб. / Р. А. Веклич, Т. О. Карнаух, А. Б. Ставровський. – К. : ВПЦ "Київський університет", 2018. – 99 с.

На прикладах реальних задач оброблення наборів однорідних даних висвітлено питання організації складних структур даних, завантаження даних у них, проектування власних і використання бібліотечних класів шаблонних контейнерів, продемонстровано застосування певних шаблонів та інших загальноприйнятих прийомів проектування.

Для студентів університетів, які вивчають комп'ютерні науки та прикладну математику.

УДК 004.43(075.8)

Зміст

Передмова 5

1. Винятки 7
1.1. Обробка помилкових та неочікуваних ситуацій 7
1.2. Генерування та обробка винятків у програмі 8
1.3. Специфікація винятків 14
1.4. Бібліотечні класи винятків 15
1.5. Методи класів, здатні генерувати винятки 16
Контрольні запитання 17

2. Контейнери та підхід до їх проетування 18
2.1. Поняття контейнера 18
2.2. Проектування класів згори донизу 18
2.3. Класи та безпечне керування ресурсами 20
2.4. Приклад контейнера з прямим доступом до елементів 21
2.5. Конструктори копії/переміщення та оператори присвоювання копіюванням/переміщенням 28
2.6. Неявні конструктори та оператори присвоювання у стандарті C++14 34
Контрольні запитання 36
Вправи 37

3. Контейнер з послідовним доступом до елементів 38
3.1. Зв'язані списки 38
3.2. Клас, що зберігає множину цілих 42
3.3. Ітератори 45
3.3.1. Поняття ітератора 45
3.3.2. Ітератор для класу IntList 47
3.3.3. Використання ітераторів 48
3.3.4. Ітератор, що обходить елементи за певною умовою 51
Контрольні запитання 55
Вправи 56

4. Шаблони 58
4.1. Шаблон функції 58
4.2. Шаблон класу на прикладі контейнера 61

Контрольні запитання 66
Вправи 67

5. Задача про обробку тексту 69
5.1. Постановка задачі 69
5.2. Загальна структура розв'язку 70
5.3. Завантаження даних із файлу в контейнер 71
5.4. Обгортки для контейнерів 79
Контрольні запитання 87
Вправи 88

6. Використання стандартних асоціативних контейнерів 90
6.1. Реалізація класу Students за допомогою стандартного бібліотечного контейнера set 90
6.2. Реалізація класу Students за допомогою стандартного бібліотечного контейнера map 91

Задачі для самостійного розв'язання 95

Післямова 98

Бібліографічний список 99

Передмова

Пропонований навчальний посібник створено на основі нормативного курсу "Програмування" для студентів факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка. Він є третім у серії посібників із цієї дисципліни. Якщо перші два розглядали питання організації обчислень і прості випадки організації даних, то в цій книзі представлено організацію складних структур даних, що виникають у реальних задачах, і доволі складні для студентів питання завантаження даних із зовнішніх носіїв у такі структури.

Посібник висвітлює питання проектування власних і використання бібліотечних класів шаблонних контейнерів засобами мови програмування C++. На конкретних прикладах розглядаються питання: як створити власний шаблонний клас або шаблон функції, як їх використовувати для конкретної задачі, як додати специфічну функціональність до загального шаблонного класу, як використовувати стандартні бібліотечні шаблонні класи для розв'язання конкретної задачі. Також розглядаються контейнери, елементами яких є інші контейнери, і організація додавання даних у них.

Разом з висвітленням можливостей мови C++ на прикладі конкретної задачі демонструється застосування певних шаблонів проектування та інших загальноприйнятих прийомів.

Посібник містить численні підготовчі вправи до комплексу лабораторних робіт і задачі для самостійного розв'язання. Робота з посібником передбачає знання основ програмування в цілому й, зокрема, мови C++ в обсязі, наприклад, посібників [1, 2].

Для подальшого вдосконалення знань із C++ і техніки програмування рекомендуємо монографії [3–9] та останній на час завершення роботи над рукописом стандарт мови C++ [10].

Код прикладів розділу 1 підготовлено за допомогою компілятора мови C++ від фірми Microsoft. Код прикладів у розділах 2–6 також цілком сумісний з компілятором MinGW 6.3.0. Деякі директиви та using-інструкції в тексті посібника могли опускатися. Повну версію коду прикладів див. на сайті книги за адресою https://sites.google.com/site/vstupdocplusplus/.

У посібнику розглядаються задачі обробки наборів однорідних даних. До таких задач належать, наприклад, обробка екзаменаційних відомостей з переліком студентів та їхніх оцінок або касових чеків, що містять перелік товарів з кількістю, ціною та вартістю тощо. Узагалі такого роду задачі можна розв'язувати засобами баз даних. Проте розгортати програмне забезпечення керування базами даних (яке може бути й не безкоштовним) заради кількох задач може виявитися недоцільним.

Продемонструємо роботу зі структурами даних на конкретній задачі.

Задача про лабораторні роботи. Текстовий файл містить дані про виконання лабораторних робіт студентами. У кожному рядку через роздільник указується:

– прізвище, ім'я та по батькові студента;

– кодова назва лабораторної роботи;

– кількість набраних балів.

При цьому студенти можуть здавати лабораторні роботи кілька разів, оскільки остаточним за лабораторну роботу стає бал найкращої спроби. Необхідно для кожного студента вивести суму набраних балів з розшифровкою по лабораторних роботах із зазначенням бала найкращої спроби та кількості спроб.

У розв'язанні цієї задачі, наведеному в розділах 5 та 6, використовується певний набір мовних і бібліотечних засобів мови C++ – винятки, контейнери, ітератори, шаблони. Знайомлять із ними розділи 1–4. На початку розділу 5 сформульовано кілька важливих принципів, утілених під час проектування й розробки програмного розв'язання задачі.

1. ВИНЯТКИ

Сучасні бібліотечні функції та класи, що реалізують структури даних, використовують винятки. У цьому розділі розглянемо механізм винятків на простих прикладах. У наступних розділах вивчатимемо створення класів, що виконують обробку помилкових ситуацій з використанням механізму винятків.

1.1. Обробка помилкових та неочікуваних ситуацій

Помилкова ситуація (від англ. *error*) виникає тоді, коли виконання програми за певних обставин не може продовжуватися звичайним шляхом. Наприклад, під час виконання програма повинна обробити файл, але не має доступу до нього, або повинна поділити ціле число на нуль.

Неочікувана ситуація (від англ. *unexpected*) зазвичай виникає тоді, коли програма отримує дані й не відомо, як їх у ній інтерпретувати. Наприклад, програма має отримати від користувача команду з певного переліку, а користувач вводить щось зовсім інше.

В обох випадках звичайне виконання програми стає неможливим. Узагальнимо описані помилкові та неочікувані ситуації поняттям виняткової ситуації (англ. *exception*).

Виняткова ситуація, або просто **виняток,** – це певна ситуація, яка не дозволяє продовжити звичайне виконання програми.

Розрізняють синхронні та асинхронні винятки. **Синхронні винятки** виникають тоді, коли їх генерують у певній частині коду програми за допомогою мовних конструкцій. **Асинхронні винятки** можуть виникнути будь-коли, вони надходять ззовні, наприклад від апаратного забезпечення, зокрема процесора, у випадку, коли ціле число ділиться на нуль.

Використання синхронної моделі винятків дозволяє суттєво зменшити код, що обслуговує механізм винятків. Зокрема, саме синхронну модель винятків використовує стандартно налаштований компілятор C++ фірми Microsoft.

У мові C++ обробка виняткових ситуацій має такий вигляд. Спочатку бібліотечна функція або власний код повідомляє про наявність виняткової ситуації. Це називають **киданням винятку** (англ. *throwing an exception*), інколи – **генеруванням винятку**. В іншому місці програми має бути код, що реагує на виняток (наприклад запобігає виконанню недопустимих дій). Його виконання називають **обробкою винятку** (англ. *handling an exception*).

Механізм винятків дозволяє зібрати обробку помилок в одному місці програми. Це порушує структурованість коду, проте за іншого підходу "розповзання" логіки обробки помилок і неочікуваних ситуацій по різних частинах коду робить його малозрозумілим. Отже, використання винятків є далеко не найгіршим способом створити зрозумілий і гнучкий код.

1.2. Генерування та обробка винятків у програмі

Приклад. Розглянемо генерування та обробку винятків у задачі обчислення квадратного кореня з дійсного числа, що його вводить користувач.

У цій задачі можливі нестандартні ситуації. Користувач може, по-перше, замість десяткового зображення дійсного числа ввести щось інше, наприклад слово `qwerty`, по-друге, ввести від'ємне число. Опишемо розв'язання задачі у функції `calculation`, яка використовує допоміжні функції `inputDouble` та `mySqrt` введення дійсного числа й обчислення квадратного кореня. Обробку помилок, поява яких можлива під час виконання викликів допоміжних функцій, реалізуємо за допомогою винятків.

Функція `inputDouble` повертає введене значення. За невдалого введення вона кидає виняток, але перед цим "прибирає за собою", тобто скидає необроблений залишок входу та помилку в потоці. Кидання винятку виділено шрифтом:

```
double inputDouble(){
  double x;
  cout<<"Input some double..."<<endl;
  cin>>x;
  if (cin.fail()) {
    cin.clear();
```

```
    cin.ignore(numeric_limits<streamsize>::max(),
               '\n');
    throw exception("no input");
  }
  return x;
}
```

Функція `mySqrt` повертає корінь квадратний зі свого аргументу. За від'ємного аргументу вона кидає виняток:

```
double mySqrt(double x){
  if (x<0)
    throw exception("sqrt from negative");
  return sqrt(x);
}
```

Дії з генерування й передачі даних про виняткові ситуації (винятки) задаються спеціальним виразом вигляду **`throw` *`вираз`***, що міститься в окремій інструкції. Спочатку обчислюється значення виразу, записаного після зарезервованого слова `throw` (англ. *кинути*). Цим значенням ініціалізується тимчасова змінна-виняток (англ. *exception object*), після чого починається пошук обробника, якому, у разі успішного пошуку, передається керування порядком обчислень. Значення змінної-винятку інколи не дуже точно називають значенням винятку, або просто винятком (так само неточно, як і цю змінну, і саму виняткову ситуацію). З контексту зазвичай зрозуміло, про що йдеться.

✓ Значення виразу після `throw` може мати довільний тип (окрім `void`); воно визначає тип змінної-винятку. Значенням винятку, зокрема, може бути об'єкт деякого класу.

Мова C++ дозволяє не брати до уваги значення виразу – тоді утворюється так звана інструкція виразу (англ. *expression statement*), тобто вираз із `;`. Саме такими є інструкції кидання винятків, записані вище у функціях.

✓ Типи винятків, які може кидати функція, ніяк не зв'язані з типом значень, що повертаються з неї.

У наведених функціях під час кидання винятку створюється об'єкт *стандартного типу винятків* `std::exception`. Об'єкт ініціалізується рядком, указаним у дужках, і надалі зберігає цей рядок. (Ком-

пілятор Microsoft дозволяє ініціалізувати об'єкт класу `std::exception` С-рядком, хоча це й суперечить стандарту.) Далі починається пошук обробника для згенерованого винятку. Функції `inputDouble` та `mySqrt` обробників винятків не містять, тому в процесі пошуку їх виклик буде передчасно завершений, а дані виклику вилучені з програмного стеку. Для того, щоб виняткова ситуація могла бути оброблена, необхідно написати код, який це буде виконувати.

Наступна функція `calculation` містить код, що описує виявлення та обробку винятків (початок і кінець частин коду, зв'язаних із цим, виділені шрифтом):

```
void calculation(){
try{
  double x=0., y=0.;
  x=inputDouble();
  y=mySqrt(x);
  cout<<"for x=" <<x<<" y=" << y<< endl;
}
catch(exception &e){
  cout<<e.what()<<endl;
  cout<<"unable to calculate"<<endl;
}
catch(...){
  cout<<"smth wrong"<<endl;
}
}
```

Для виявлення та обробки винятків у цій функції використовується **інструкція try**, складена з `try`-блоку й непорожньої послідовності `catch`-блоків після нього (основними значеннями англ. слова *try* є "випробовувати", "перевіряти", а *catch* – "ловити", "спіймати", "пастка").

Склад `try`-блоку: зарезервоване слово `try` і блок після нього. Про код у `try`-блоці кажуть, що він **охороняється** (англ. *the guarded section of code*): під час його виконання перевіряється поява винятків. Якщо з'являється виняток, то він може бути перехоплений і оброблений.

Кожен `catch`-блок (пастка) починається зарезервованим словом `catch`, далі йде оголошення винятку в круглих дужках і блок. Код у блоці пастки призначений для обробки винятків типу, указаного в оголошенні. Інколи замість терміна "обробляє" (англ. *handle*) уживають "ловить" (англ. *catch*).

У наведеній функції catch-блок

```
catch(exception &e){
  cout<<e.what()<<endl;
  cout<<"unable to calculate"<<endl;
}
```

ловить винятки типу exception& і тих типів, що приводяться до нього, зокрема типу exception і похідних від нього класів. Для того, щоб мати доступ до значення винятку, використовують **іменований виняток**. Ім'я винятку, яке не є обов'язковим, указується після його типу. Воно доступне в межах блоку пастки та при її активації ініціалізується кинутим винятком. Блок пастки містить послідовність інструкцій, які обробляють зловлений виняток.

Областю дії оголошення імені винятку є блок пастки. Водночас змінні, означені в try-блоці, недоступні в catch-блоці. За потреби передати дані в catch-блок використовують значення винятку, наприклад, вираз e.what() повертає рядок, указаний при киданні винятку.

✓ Одному try-блоку може відповідати кілька catch-блоків, що ловлять винятки різних типів. Якщо заздалегідь невідомо, виняток якого типу може бути кинутий, або не потрібно розбиратися, якого саме типу виняток було згенеровано, то можна перехопити виняток будь-якого типу. Довільний тип винятку вказується в пастці знаком "...".

✓ Якщо виняток оброблено яким-небудь catch-блоком, то на цьому виконання try-інструкції завершується й решта catch-блоків цього try-блоку вже не виконуються.

✓ Якщо виконання try-блоку завершується звичайним способом, тобто не припиняється через появу винятків, то на цьому виконання try-інструкції завершується. Жодна її пастка керування не отримує.

У нашому прикладі, якщо користувач вводить нечислове або від'ємне значення, то завершується не тільки виконання відповідного виклику, але й виконання try-блоку у функції calculation. Далі виконується пастка для винятків типу exception&. На цьому виконання try-інструкції закінчується. Вона є останньою в тілі функції calculation, тому виконання її виклику завершується звичайним чином.

Нарешті, запишемо головну функцію:

```
int main(){
  calculation();
  system("pause"); return 0;
}
```

Порядок виконання. Коли кинуто виняток, його обробляє найближча пастка, яка відповідає його типу. Розглянемо детальніше, як відбувається пошук обробника.

Нагадаємо: під час виконання програми утворюється послідовність викликів функцій і вкладених інструкцій, виконання яких розпочато, але не завершено. Коли закінчується виконання блоку, означені в ньому змінні вилучаються з програмного стеку, а завершення виклику функції звільняє стек від усіх даних виклику. При цьому виконання інструкцій і викликів функцій завершується в порядку, зворотному до порядку початків їх виконання.

Після кидання винятку починається процес послідовного припинення виконання вкладених інструкцій і викликів функцій. Дані інструкцій і викликів вилучаються з програмного стеку. Цей процес називають **розмотуванням стеку** (англ. *stack unwinding*).

Як тільки припиняється виконання `try`-блоку тієї `try`-інструкції, що містить `catch`-блок винятків кинутого типу, розмотування стеку зупиняється. Далі починають виконуватися інструкції цього `catch`-блоку (кажуть, що виняток перехоплено пасткою, або *активовано обробник* винятку). Якщо виконання інструкцій блоку пастки добігає кінця, то виконання `try`-інструкції, чия пастка перехопила виняток, завершується, і далі звичайним шляхом виконуються інструкції програми після цієї `try`-інструкції.

✓ Під час визначення, чи може пастка обробити виняток, до уваги не беруться `const` та `&`.

✓ Пастки `try`-інструкції перевіряються в порядку запису. Пастка для винятків базового типу може обробляти винятки похідних типів, тому пастку для винятків базового класу завжди слід записувати після пасток для похідних класів (якщо такі пастки є), а пастку для винятків усіх типів – останньою.

✓ Якщо під час розмотування стеку пастку для винятку не знайдено, то програма аварійно завершується.

За необхідності перехоплений пасткою виняток можна кинути далі. Для цього в блоці пастки достатньо записати просто `throw;`. Можна кинути також інший виняток. При цьому код `catch`-блоку, що генерує виняток, міститься після `try`-блоку поточної `try`-інструкції, тому її пастки перевірятися вже не будуть.

Приклад. Розглянемо складнішу ситуацію. Нехай є такі функції f та g:

```
void g(int n){
  int a;
  if (n==1) throw 'c';
  if (n==2) throw int(3);
  if (n==3) throw exception();
}
void f(int n){
  try {                      // 1
    g(n);
  }
  catch (char) {cout<<"cought char in f()\n";} // 2
  cout<<"end of f\n";              // 3
}
```

Розглянемо дії під час виконання такої `try`-інструкції:

```
try{                        // 4
  f(1);                      // 5
  f(2);                      // 6
  f(3);                      // 7
}
catch (char){cout<<"cought char in sample\n";} // 8
catch (int) {cout<<"cought int in sample\n";} // 9
catch (...) {cout<<"cought smth in sample\n";} //10
```

Зобразимо схематично передачу керування між частинами коду в результаті виконання цієї інструкції (послідовні дії розміщено на одному рівні, вкладені – з відступом). У дужках указано номери рядків, записані вище в коментарях:

```
try-блок try-інструкції      (4)
  виклик f(1)
    try-блок try-інструкції   (1)
      виклик g(1)
        кидається виняток 'c'
    catch-блок          (2)
    повернення з виклику f(1)
```

```
  виклик f(2)
    try-блок try-інструкції   (1)
      виклик g(2)
        кидається виняток int(3)
catch-блок            (9)
```

За наведених дій на екран виводяться такі рядки:

```
cought char in f()
end of f
cought int in sample
```

Зауважимо, що після того, як кинуто виняток `'c'`, із програмного стеку послідовно вилучаються дані виклику `g(1)` та `try`-блоку `try`-інструкції (1). Після кидання винятку `int(3)` зі стеку вилучаються дані виклику `g(2)`, `try`-блоку `try`-інструкції (1) та виклику `f(2)`. Після виконання `catch`-блоку (9), що зловив виняток, виконання всієї `try`-інструкції (4) завершується, тому виклик `f(3)` узагалі не виконується. □

✓ Обробку винятків можна, але зазвичай не варто записувати в головній функції. Через розмотування програмного стеку кидання винятків здатне передавати керування дуже віддаленим частинам коду, тому обробку виняткових ситуацій необхідно забезпечувати якомога раніше.

1.3. Специфікація винятків

Якщо під час виконання функція генерує й не перехоплює виняток, то її виклик передчасно завершується. У такій ситуації кажуть, що **функція кидає виняток**. Для організації обробки помилок важливо знати, винятки яких типів можуть кидати використовувані функції та чи можуть кидати взагалі, оскільки тоді можна контролювати тільки появу винятків певних типів.

Хоча мова C++ має засоби специфікації типів винятків, які може кидати функція, поточний стандарт рекомендує специфікувати винятки за принципом "або все, або нічого". Наприклад, вважається, що функція з оголошенням

```
int f();
```

може кинути виняток довільного типу (згідно зі своїм кодом), а функція з оголошенням

```
int g() noexcept;
```
не може кидати жодних винятків. Якщо виклик функції g завершиться в результаті появи винятку, то це може призвести до аварійного завершення програми.

Водночас, користуючись функцією, програміст повинен знати, винятки яких типів вона кидає, тому типи винятків, що їх може кидати функція, часто вказують у коментарях.

1.4. Бібліотечні класи винятків

У мові C++ бібліотечний клас винятків `exception` є базовим для низки інших, що відповідають різним типам помилок. Усі помилки поділяють на дві групи: логічні та помилки часу виконання (англ. *runtime*).

Логічні помилки визначаються внутрішньою логікою програми – некоректне значення індексу, значення не з області визначення функції тощо. Такі помилки теоретично можна передбачити й уникнути їх під час виконання.

Помилки часу виконання виникають через події за межами коду програми. Наприклад, арифметичне переповнення під час обчислень або відсутність файлу на диску.

Логічним помилкам у цілому відповідає клас `logic_error`, який має низку похідних класів: `domain_error`, `out_of_range`, `length_error`, `invalid_argument`. Передбачається, що винятки типу `domain_error` мають сповіщати про некоректний аргумент під час математичних обчислень; винятки типу `out_of_range` найчастіше сигналізують про вихід значення за межі індексного діапазону, типу `length_error` – про спробу побудувати об'єкт, розмір якого перевищує дозволений, типу `invalid_argument` – про некоректне значення в решті випадків. Наведені класи винятків оголошені в стандартному бібліотечному файлі `stdexcept`.

Помилкам часу виконання в цілому відповідає клас `runtime_error` (треба включати стандартний заголовний файл `stdexcept`), який також має низку похідних класів. Серед них зазначимо тільки клас `ios_base::failure`, призначений сповіщати про помилки, що виникають під час операцій з буфером потоку.

Операції new та new[] у випадку неможливості виділення динамічної пам'яті кидають винятки класу bad_alloc, похідного безпосередньо від exception.

Клас exception має віртуальний метод what(), що повертає С-рядок з повідомленням щодо проблеми. Вміст цього повідомлення стандартом не регламентується, але природно очікувати, що стандартні бібліотечні функції кидають винятки похідних класів, для яких це повідомлення відображає суть проблеми.

За стандартом, об'єкти всіх наведених класів винятків, окрім exception та bad_alloc, під час створення треба ініціалізувати С-рядком або рядком типу string, вміст якого далі можна отримати через виклик методу what(). На відміну від стандарту, компілятор Microsoft дозволяє ініціалізувати об'єкт класу std::exception С-рядком.

1.5. Методи класів, здатні генерувати винятки

Методи класу можуть кидати винятки. Якщо виняток генерується у звичайному методі (не конструкторі й не деструкторі), то його обробка нічим не відрізняється від обробки винятку, кинутого функцією. Так само відбувається розмотування стеку, тобто змінні блоків і викликів функцій вилучаються зі стеку доти, доки не виконується catch-блок для відповідного типу винятків або не вилучається головна функція, тобто програма завершується аварійно. При вилученні об'єктів для них викликається деструктор.

Обробка винятку, кинутого й не перехопленого в конструкторі, дещо відрізняється. Якщо для об'єкта жоден конструктор до кінця ще не відпрацював, то об'єкт не вважається сконструйованим. Тому при розмотуванні стеку для такого об'єкта *деструктор не викликається*. Проте, якщо, у свою чергу, поля об'єкта є об'єктами або масивами об'єктів і вони вже сконструйовані, то для них їхні деструктори викликаються. Якщо базовий об'єкт було сконструйовано, то для нього конструктор теж викликається.

Деструктори також можуть кидати винятки, але ці *винятки мають перехоплюватися в тілі деструктора*. Інакше можлива ситуація, коли програма завершиться аварійно, оскільки, за стандартом, виняток, кинутий з деструктора під час розмотування стеку, призводить до аварійного завершення програми.

Контрольні запитання

1. Що таке виняток?
2. Що таке кидання винятку?
3. Що таке обробка винятку?
4. Якого типу виняток генерується за інструкцією `throw`?
5. Що означає, що код охороняється?
6. Що таке розмотування стеку?
7. Які дії виконуються за `try`-інструкцією?
8. Чи може `try`-інструкція мати кілька `catch`-блоків?
9. Чи доступні в `catch`-блоці змінні, означені у відповідному `try`-блоці?
10. Як передати дані про помилку в обробник?
11. Які стандартні класи винятків наявні в мові C++?
12. У якому випадку активується обробник винятку?
13. Що відбудеться, якщо кілька пасток `try`-інструкції потенційно здатні перехопити виняток?
14. Чи можна в обробнику зловлений виняток кинути далі?
15. Чи можна в обробнику замість зловленого винятку далі кинути інший?
16. Що станеться, якщо жоден обробник для винятку не активується?
17. Чи має функція перехоплювати винятки, згенеровані під час її виконання?
18. Що означає, що функція може кидати винятки?
19. Як описати в коді те, що функція винятки не кидає?
20. Чи може конструктор кидати винятки?
21. Чи може деструктор кидати винятки?

2. КОНТЕЙНЕРИ ТА ПІДХІД ДО ЇХ ПРОЕКТУВАННЯ

2.1. Поняття контейнера

Структури даних мають конкретну організацію та реалізують погляд на дані з боку розробника, а не користувача. Серед структур даних виділяють контейнери. Під **контейнером** (англ. *container*), або **колекцією** (англ. *collection*), розуміють структуру даних, що зберігає набір значень одного чи різних типів (але зазвичай похідних від спільного базового) і надає доступ до цих значень. Як правило, кількість елементів даних у контейнері може бути довільною й за відсутності явних обмежень значення елементів можуть повторюватися. Найтиповіші контейнери дозволяють додавати та вилучати дані, а також надають доступ до даних, у тому числі реалізують пошук даних у контейнері. Найчастіше контейнери застосовуються для зображення однотипних даних, що обробляються однаковим способом. Серед контейнерів виділяють такі, що зберігають послідовності значень (англ. *sequence container*), наприклад масиви та списки [2]. Деякі інші різновиди контейнерів розглядаються в розд. 6.

Крім операцій над даними, контейнери зазвичай містять операції, що дозволяють перевірити, чи є контейнер порожнім, визначити кількість елементів, які він зберігає, та деякі інші.

2.2. Проектування класів згори донизу

Подробиці технічної реалізації є невід'ємною складовою структур даних, проте починати проектування саме з них не варто. Будемо вести проектування *згори донизу* (див. [1]), поступово уточнюючи пов'язані із задачею поняття та потрібні для обробки дії. Проектуючи контейнер згори донизу, спочатку визначимо дані, що їх зберігає контейнер, та обмеження на них, наприклад, чи можуть дані повторюватися. Також визначимо допустимі операції над вмістом контейнера, наприклад додавання, пошук,

вилучення даних, і тільки потім перейдемо до уточнення подробиць реалізації такого контейнера. Згори донизу будемо проектувати не тільки контейнери, але й код проектів у цілому.

Проектувати структури даних будемо у вигляді *класів* (див. [2]), оскільки класи дозволяють інкапсулювати всі необхідні операції з даними й забезпечити цілісність останніх. При цьому будемо намагатися *відділяти інтерфейс від реалізації.*

До **інтерфейсу класу** входять відкриті методи класу (у широкому розумінні сюди ж додають загальні коментарі стосовно призначення класу). Приховані та захищені члени класу й означення методів становлять **реалізацію** класу (англ. *implementation*). Суть відділення інтерфейсу від реалізації полягає в тому, що для використання класу клієнту[1] не треба нічогісінько знати про його реалізацію.

Перш за все, відділення інтерфейсу від реалізації суттєво спрощує використання класу його клієнтами. Крім того, зміни в реалізації одного класу не потребують змін у реалізації інших класів та в коді клієнта. Такий підхід дозволяє досягти модульності програмних продуктів і мати кілька взаємозамінних реалізацій окремого модуля. Це підвищує ефективність колективної розробки проекту, адже його частини модифікуються незалежно одна від одної.

Принцип відділення інтерфейсу від реалізації зустрічається в повсякденному житті. Користуючись мобільним телефоном, праскою, електрочайником, телевізором або іншим пристроєм, ми знаємо, які кнопки слід натиснути для досягнення певного ефекту, але не замислюємося, як це все влаштовано всередині. Це зручно й суттєво економить час.

✓ Проектування класу варто починати з визначення його інтерфейсу, після чого переходити до полів і решти методів. Більш того, деякі приховані методи можуть з'явитися лише на етапі кодування.

✓ В означенні класу не варто, хоча й можливо, записувати повні означення методів – тут достатньо лише оголошень методів.

Принцип відокремлення інтерфейсу від реалізації цілком узгоджується з проектуванням згори донизу: спочатку проектуються операції, які має надавати клас клієнтам, потім вони уточнюються й поступово з'являється реалізація класу.

---

[1] Під **клієнтами** класу розуміють частини програми, що використовують клас або його об'єкти.

2.3. Класи та безпечне керування ресурсами

Ще одна причина проектування структур даних саме у вигляді класів пов'язана з організацією *безпечного керування ресурсами* (англ. *exception-safe resource management*). У широкому сенсі під ресурсами можна розуміти все те, що використовує програма: динамічну пам'ять, дискові файли тощо. За безпечного керування ресурсами має не бути **витоків ресурсу** (англ. *resource leaks*) – ситуацій, коли програма не повертає в користування отримані ресурси. Розглянемо потенційні проблеми детальніше.

За зберігання наборів даних у автоматичній чи статичній пам'яті кількість їхніх елементів має бути доволі невеликою відносно можливостей сучасних комп'ютерів. Також необхідно заздалегідь резервувати пам'ять під дані незалежно від їхнього реального розміру. Як наслідок або буде зарезервовано зайву пам'ять (і її далі може не вистачити для інших обчислень програми), або дані не вмістяться в зарезервований обсяг. Для того, щоб уникнути таких проблем, доцільніше розміщувати дані в динамічній пам'яті.

Динамічні змінні (на відміну від автоматичних) не знищуються самі по собі. Зайнята ними пам'ять звільняється тільки в результаті виконання явно записаної інструкції. Для того, щоб ця інструкція була виконана, *до неї має дійти черга*. Однак, якщо було згенеровано та відразу не перехоплено виняток, то звичайний порядок виконання інструкцій переривається, виконання поточного виклику завершується й інструкції звільнення пам'яті, які найчастіше містяться в кінці тіла функції, не виконуються. Як результат, накопичується "сміття", тобто незвільнені ділянки динамічної пам'яті, подальше використання яких неможливе.

Ідея використання винятків виходила з того, щоб зробити основний код прозорішим, очистивши його від коду обробки помилок. Як сумістити використання винятків і коректну роботу з динамічною пам'яттю?

Насправді аналогічні проблеми виникають при роботі як з динамічною пам'яттю, так і з іншими ресурсами. Розглянемо, як за умови використання винятків організувати безпечне керування ресурсами.

Один зі способів безпечного керування ресурсами полягає у використанні таких принципів:

– керування ресурсом відбувається в об'єкті (екземплярі класу), при цьому об'єкт відповідає не більш ніж за один ресурс;

– ресурс виділяється, коли об'єкт ініціалізується (не обов'язково в конструкторі; можлива так звана *відкладена ініціалізація*, коли початкове значення присвоюється не під час ініціалізації, а пізніше);
– ресурс вивільняється в деструкторі, застосованому до об'єкта.

Коректно реалізований клас дозволяє уникнути витоків ресурсу, оскільки при знищенні об'єктів для них обов'язково викликається деструктор, у якому можна коректно вивільнити ресурс.

Отже, якщо контейнер розміщує дані в динамічній пам'яті, то відповідний клас обов'язково має містити власний деструктор, а також гарантувати коректність роботи з динамічною пам'яттю під час присвоювання та створення об'єктів одного за іншим. Відповідні операції мають бути або належно реалізовані, або заборонені для використання.

2.4. Приклад контейнера з прямим доступом до елементів

Продемонструємо наведені вище принципи та деякі можливості мови C++ на прикладі простого контейнера. Нехай об'єкти створюваного класу зберігають послідовність цілих, забезпечують операцію додавання елемента в кінець послідовності й надають прямий доступ до її елементів. Це буде відповідальністю, або призначенням, нашого класу.

Якщо певну частину даних можна отримати безпосередньо за її номером або адресою, то кажуть, що до даних є **прямий**, або **довільний, доступ** (англ. *direct/random access*). Це означає, що до елементів послідовності є доступ за їхніми номерами, тобто позиціями в послідовності. Наприклад, масиви забезпечують прямий доступ до своїх елементів, змінна – до свого значення, розіменування вказівника – до даних, які зберігаються за відповідною адресою.

Якщо ж для того, щоб отримати значення деякого елемента послідовності (навіть маючи його номер), треба спочатку переглянути всі елементи перед ним, то це – приклад **послідовного доступу** (англ. *sequential access*). На відміну від прямого, послідовний доступ передбачає, що елементи переглядаються в заздалегідь визначеному порядку, наприклад від першого до останнього. Типовим прикладом структур даних із послідовним доступом до елементів є списки [2].

Проектування класу почнемо з його інтерфейсу.

Під час конструювання створюється порожня послідовність.

Метод `size` повертає довжину поточної послідовності.

Для організації прямого доступу до елементів послідовності перевантажимо оператор `[]`, який за індексом повертає посилання на відповідний елемент послідовності (індексацію почнемо з `0`). При наданні доступу до елементів послідовності будемо економити час, а тому відмовимося від перевірок коректності індексу запитуваного елемента. Це не дуже добре, але деякі системні бібліотечні класи використовують таку стратегію заради економії часу.

Аналогічно стандартним бібліотечним класам, додамо до класу метод `at`, що, як і оператор `[]`, повертає посилання на відповідний елемент послідовності, але перевіряє коректність індексу запитуваного елемента й, коли такого елемента в послідовності немає, кидає виняток типу `out_of_range`.

Також означимо метод `insert`, який додає елемент у кінець послідовності та повертає посилання на щойно доданий елемент.

З вищенаведеного випливає такий інтерфейс класу:

```
class Sequence{
public:
  Sequence();
  unsigned size() const; //розмір в елементах
  int& operator[](unsigned); //без перевірок,
    //посилання коректне поки довжина
    //послідовності незмінна
  int& at(unsigned); //throw(out_of_range)
    //посилання коректне поки довжина
    //послідовності незмінна
  void insert(int elem);//додає елемент у кінець
                        //throw(bad_alloc)
//решту буде додано далі
};
```

Міркуючи над реалізацією класу, вирішуємо зберігати послідовність у динамічному масиві, щоб не обмежувати можливу кількість її елементів. Далі означимо поля даних. За використання динамічного масиву екземпляр класу містить довжину послідовності, довжину (кількість елементів) масиву та вказівник на динамічний масив цілих. Додамо таку частину до означення класу:

```
private:
  unsigned size_=0;
  unsigned capacity_=0;
  int *arr=nullptr;
```

Стандарт C++14 *дозволяє вказувати ініціалізатори полів об'єкта безпосередньо в означенні класу*. Якщо в конструкторі не вказаний ініціалізуючий вираз для поля об'єкта, то використовується ініціалізатор із означення класу. Тому стандартний (англ. *default*) конструктор можна означити за умовчанням (англ. *defaulted*). Отже, змінимо означення конструктора:

```
Sequence()=default;.
```

Виконання конструктора ініціалізує поля `size_`, `capacity_`, `arr` значеннями виразів, указаними в їх оголошенні.

Зауваження. Замість внесення змін в означення класу можна у cpp-файлі з реалізацією методів класу записати таке.

```
Sequence::Sequence()=default;
```

Під час додавання елемента може виникнути ситуація, коли поточний масив повністю заповнено. Тоді перед додаванням елемента масив необхідно збільшити. Для цього треба замінити поточний динамічний масив більшим, у який скопійовано елементи послідовності. Відповідні дії виконує метод `resize`, а крок збільшення кількості елементів масиву задає поле класу `capacityStep`:

```
private:
  void resize(); //throw(bad_alloc)
  static const unsigned capacityStep=32;
```

У cpp-файлі з реалізацією методів класу напишемо таке:

```
#include "Sequence.h"
#include <memory.h>
unsigned Sequence::size()const {return size_;}
int& Sequence::operator[](unsigned n) const{
  return arr[n];
}
int& Sequence::at(unsigned n) const{
  if (n>=size_) throw std::out_of_range(
        "There is no element with index " +
        std::to_string(n) +" in the sequence.");
  return arr[n];
}
```

```
void Sequence::insert(int elem){
  if (size_==capacity_) resize();
  arr[size_++]=elem;
}
void Sequence::resize(){
  unsigned capacityNew=capacity_+capacityStep;
  int *arrNew=new int[capacityNew];
  if (arr && size_>0)
    memcpy(arrNew, arr,size_*sizeof(int));
  delete[] arr;
  arr=arrNew;
  capacity_=capacityNew;
}
```

У наведеному коді масив копіює функція memcpy, оголошена в стандартному бібліотечному файлі memory.h. Бібліотечна функція to_string за числом повертає його десяткове зображення рядком типу string.

Зауваження. Метод operator[] можна оголосити як константний, адже він не змінює поля об'єкта. Додавання специфікатора const дозволяє використовувати його для об'єктів типу як Sequence, так і const Sequence. Проте об'єкту належить динамічний масив, який можна змінити за допомогою цього методу. У подібних випадках краще мати пару методів, а саме:

– int& operator[](unsigned), який надає доступ для читання та запису елемента;

– int operator[](unsigned) const, який надає доступ тільки для читання елемента.

Аналогічне зауваження стосується й методу at, який варто замінити такою парою методів:

```
int at(unsigned)const;
int& at(unsigned);
```

Реалізації "константних" методів operator[] та at залишаються ті самі.

Наведені методи відрізняються специфікатором const, тому компілятор вважає їх різними. Виклик того чи іншого з пари методів залежить від контексту й самостійно визначається компілятором. Якщо метод викликається для незмінюваного об'єкта, то компілятор вибирає саме "константний" метод.

Однак як утилізувати пам'ять із-під масиву? Бажано робити це саме перед тим, коли об'єкт вивантажується з пам'яті. Згадаємо: перед вивантаженням об'єкта викликається деструктор. Якщо програміст його не написав, то компілятор додає свій, що виконує необхідні завершальні дії. Проте компілятор не може знати, чи потрібен ще динамічний масив (можливо, він використовуватиметься й надалі). Отже, додамо до класу власний деструктор, який знищує масив. Ім'я деструктора завжди фіксоване: це ім'я класу із символом ~ попереду. Деструктор не має аргументів та нічого не повертає:

```
public:
  ~Sequence ();
```

Також додамо його реалізацію:

```
Sequence::~Sequence(){
  delete[] arr; arr=nullptr; size_=0;
}
```

Ситуації, у яких дійсно є потреба в програмі мовою C++ явно викликати деструктор, дуже специфічні й трапляються рідко. Варто пам'ятати, що звернення до об'єкта, для якого вже було виконано деструктор, може призводити до непередбачуваних наслідків, тобто для одного об'єкта *виконувати деструктор двічі не можна*. Реалізацію деструктора можна спростити, адже після його виконання "повноцінне життя" об'єкта завершене й значення його полів уже неважливі. Отже, деструктор лише звільняє пам'ять:

```
Sequence::~Sequence (){delete[] arr;}
```

Проте на цьому проблеми не закінчуються. Розглянемо такий код:

```
Sequence a;
a.insert(1);
Sequence b(a), c;
c=a;
```

Тут викликається конструктор копії та оператор присвоювання копіюванням. Ми ці методи не написали, стандарт не забороняє додавати їх за наявності написаного програмістом деструктора, отже, компілятор додав їх за нас. Їхні дії зводяться до копіювання відповідних полів. Нагадаємо: об'єкт зберігає не масив, а вказівник на масив елементів. Тому копіюватися буде не послідовність, а тільки адреса, за якою вона розміщена, тобто має місце *неглибоке копіювання*.

Подальша зміна значень елементів послідовності в одному з об'єктів a, b або c одночасно є зміною у двох інших об'єктах. Набагато гірше те, що у викликах деструктора для кожного з трьох об'єктів буде робитися спроба звільнити пам'ять, виділену масиву в екземплярі a. До того ж пам'ять із-під масиву, спочатку створеного об'єктом c, звільнена не буде, оскільки після присвоювання адреса цього масиву втрачається.

Вилучені методи. Некоректна робота з пам'яттю є результатом неглибокого копіювання полів-вказівників у конструкторі копії та в операторі присвоювання копіюванням, неявно доданих компілятором. Отже, або треба запрограмувати ці методи явно, щоб вони правильно обробляли вказівники (якщо пам'ять під динамічний масив було виділено, то спочатку його слід знищити, а потім створити новий і скопіювати в нього значення), або заборонити використання методів, неявно доданих компілятором. Підемо другим шляхом. Починаючи зі стандарту C++11, відповідні методи можна оголосити відкритими, але означити їх як **вилучені** (англ. *deleted*):

```
public:
  Sequence(const Sequence&)=delete;
  Sequence& operator=(const Sequence&)=delete;
```

Тепер довільна спроба використати їх у коді викличе помилку на етапі компіляції.

Є ще два методи, які можуть призводити до неправильної утилізації пам'яті – це конструктор переміщення та оператор присвоювання переміщенням. Проте, якщо в класі явно оголошено деструктор, то стандарт C++14 забороняє[2] неявно додавати ці методи до класу.

Перевантаження операторів і методи класу. Оголошення різних об'єктів програми з тим самим ім'ям в одній області дії називається **перевантаженням** (англ. *overloading*) **імені** (інколи, не точно, перекладають як *переозначення імені*). Мова C++ дозволяє перевантажувати імена функцій, крім імені `main`. Під час компіляції необхідне означення вибирається за кількістю й типами параметрів. Тому однойменні функції повинні мати різні кількості параметрів або різні послідовності їхніх типів. Типи значень, що повертаються, можуть бути як однаковими, так і різними. Означення однойменних функцій, які відрізняються лише типами значень, що повертаються, є помилкою.

---

[2] Хоч стандарт і забороняє, але не всі компілятори цілком відповідають стандарту. Якщо в класі явно оголошено деструктор, а компілятор усе-таки генерує конструктор переміщення та оператор присвоювання переміщенням, то їх можна означити як вилучені (за допомогою конструкції `=delete`).

Для класів дозволяється перевантажувати як методи, так і оператори. При цьому більшість операторів може бути перевантажена не тільки як методи класу, але й як звичайні функції за межами класу. У наведеному прикладі оператор [] перевантажено як метод класу.

Перевантажувати можна й **оператори перетворення до типу**. Перевантажимо для класу Sequence оператор перетворення до типу bool так, щоб за порожньої послідовності отримувати false, за непорожньої – true. До означення класу слід додати

```
public:
  operator bool() const;
```

Цей метод має ім'я operator bool і його синтаксичною особливістю є те, що тип результату перед його іменем не вказується.

У файл із реалізацією методів додамо його означення:

```
Sequence::operator bool() const{
  return size()!=0;
}
```

Зауваження. У коді оператора перетворення до типу bool замість поля size_ указано виклик методу, що повертає відповідне значення, і це не випадковість.

Припустимо, що в майбутній версії класу поле size_ матиме інший зміст (наприклад, вирішимо зберігати не кількість елементів, а індекс останнього використаного). Тоді необхідно змінити методи, що відповідають за його зміну, і методи, що використовують його значення. Причому зміни в усіх методах, що використовують значення, мають бути однаковими. Якщо таких методів багато, то виникають зміни в багатьох місцях коду. Якщо за логіку перетворення значення поля size_ на кількість елементів, що зберігається в контейнері, відповідає єдиний метод (size), то достатньо змінити тільки його. Отже, використання саме виклику методу дозволяє уникнути дублювання коду обчислення кількості елементів. За бажанням у поточному класі метод size можна зробити вбудованим.

Операції введення/виведення для контейнера. Операції введення/виведення прийнято за можливістю відділяти від операцій з даними. Тому в розробленому класі немає жодних засобів з його виведення. Натомість перевантажимо оператор << вставки в потік. Для цього в заголовний файл після означення класу запишемо такий заголовок (ще необхідно включити стандартний заголовний файл ios):

```
ostream& operator <<(ostream&, const Sequence&);
```

У відповідний файл із реалізаціями додамо його означення (ще необхідно включити стандартний заголовний файл ostream):

```
ostream& operator<<(ostream& f, const Sequence&s){
  if (s.size()==0) f<<"sequence is empty";
  else {
    unsigned last=s.size()-1;
    for(unsigned i=0; i<=last;++i) f<<s[i]<<" ";
  }
  f<<endl;
  return f;
}
```

Перевантаження оператора >> вставки з потоку залишаємо для вправи.

2.5. Конструктори копії/переміщення та оператори присвоювання копіюванням/переміщенням

Для класу може бути означено кілька різних конструкторів, кожен з яких викликається (неявно або явно) у певних ситуаціях. Конструктори та конструктори копії в тому вигляді, як вони існували до введення стандарту C++11, розглядалися у [2]. Починаючи зі стандарту C++11, у випадку, коли новий об'єкт створюється за вже існуючим об'єктом того самого класу, може використовуватися **конструктор копії** (англ. *copy constructor*) або **конструктор переміщення** (англ. *move constructor*).

Опишемо різницю між цими двома конструкторами. Припустимо, що об'єктами є конспекти. Для того, щоб за одним конспектом сконструювати новий, можна або переписати конспект у новий зошит, або вирвати аркуші та вставити в нову обкладинку. В обох випадках значення сконструйованого конспекту має збігатися[3] зі значенням конспекту, за яким його було сконструйовано, але очі-

[3] Звісно, можливі випадки, коли в програміста виникає дивне бажання, щоб конструктори копії/переміщення робили не зовсім точні копії.

кується, що конструктор копії створює копію даних зразка, а конструктор переміщення "забирає" дані зразка собі, тобто дані *переміщуються* від існуючого об'єкта до створюваного.

✓ Для класів, які керують ресурсами, копіювання означає створення нової копії ресурсу, а переміщення – що ресурс передається створюваному об'єкту.

При присвоюванні об'єктів, аналогічно конструюванню, теж розглядають копіювання та переміщення. Відмінність від конструкторів лише в тому, що значення копіюється або переміщується не в створюваний об'єкт, а у вже існуючий.

Нехай у класі наявні конструктор копії та конструктор переміщення. Якщо вираз, що задає об'єкт, за яким конструюється новий, є *l*-виразом, то має виконуватися конструктор копії, інакше – конструктор переміщення. Останній випадок виникає, зокрема, тоді, коли об'єкт створюється за тимчасовим об'єктом. Зазначимо, що стандарт дозволяє компілятору оптимізувати кількість тимчасових об'єктів і прибирати зайві виклики конструкторів.

Якщо в класі наявний тільки конструктор копії, то він може застосовуватися замість конструктора переміщення.

Вибір оператора присвоювання відбувається аналогічно. Наприклад, у коді

```
string s1("Example"), s2(s1),
       s3(s1+" of move constructor");
s3=s2;
s3=s1+"of move-assignment";
```

для конструювання об'єкта `s2` використовується конструктор копії (оскільки ім'я `s1` є *l*-виразом), а для конструювання об'єкта `s3` – конструктор переміщення, оскільки конструюється за виразом, що задає тимчасовий об'єкт. Аналогічно у присвоюванні `s3=s2;` викликається оператор присвоювання копіюванням, а в останньому присвоюванні – оператор присвоювання переміщенням.

Власні конструктори копії/переміщення та оператори присвоювання копіюванням/переміщенням для класу `Sequence`. У поточній версії класу `Sequence` відповідні конструктори та оператори присвоювання було вилучено, щоб уникнути неправильної роботи з виділеною під масив динамічною пам'яттю. Модифікує-

мо клас Sequence, додавши власні означення конструкторів копії й переміщення та операторів присвоювання копіюванням і переміщенням (у коді їх наведено в порядку переліку):

```
public:
  Sequence(const Sequence&);
  Sequence(Sequence&&);
  Sequence& operator=(const Sequence&);
  Sequence& operator=(Sequence&&);
```

Під час копіювання створюємо нову ділянку пам'яті для збереження послідовності й копіюємо в неї послідовність; при переміщенні передаємо вказівник від одного об'єкта іншому.

В операторі присвоювання перед тим, як виконувати присвоювання, спочатку необхідно звільнити пам'ять, виділену під "стару" послідовність об'єкта. Так само треба звільняти пам'ять і в деструкторі. Оскільки після виклику деструктора об'єкт вважається зруйнованим, то використовувати деструктор для звільнення пам'яті не можна. Для зменшення дублювання коду додамо прихований метод release, який звільняє пам'ять (його можна використати й у деструкторі):

```
private:
  void release();
```

Запишемо його означення:

```
void Sequence::release(){
  delete[] arr;
  arr=nullptr;
  size_=0;
  capacity_=0;
}
```

Тоді **оператор присвоювання копіюванням** виглядає так:

```
Sequence& Sequence::operator=
              (const Sequence &other){
  if (this!=&other){
    release();
    if (other.size_>0) {
      arr=new int[other.capacity_];
      capacity_=other.capacity_;
      size_=other.size_;
      memcpy(arr,other.arr,size_*sizeof(int));
    }
```

```
    }
    return *this;
  }
```

Тут ураховано, що *за присвоювання об'єкта самому собі жодних дій з копіювання виконувати не треба.* Конструктор копії спочатку звільняє свій динамічний масив, потім створює новий і копіює в нього елементи з іншої послідовності. Зазначимо, що наведений код можна дещо оптимізувати. Зокрема, створювати новий масив тільки тоді, коли поточного недостатньо для збереження іншої послідовності.

Запишемо **конструктор копії**, у якому об'єкт спочатку ініціалізується за допомогою стандартного конструктора[4], а потім йому присвоюється значення іншого об'єкта:

```
пробіли як тут
Sequence::Sequence
  (const Sequence &other):Sequence(){*this=other;}
```

У цьому конструкторі використано **делегування** – коли одна функція (метод, конструктор) *делегує*, тобто передає, роботу іншій. Конструктор копії спочатку за допомогою стандартного конструктора встановлює значення полів (делегує роботу зі створення об'єкта стандартному конструктору), а потім виконує власні дії. Делегування в конструкторах допускається стандартом C++14 і дозволяє *уникати дублювання коду*, що встановлює початкові значення полів.

Делегування вказується тільки в секції ініціалізації конструктора, який у цьому випадку називають **делегуючим** (англ. *delegating*). При цьому інші ініціалізатори вже записувати не можна. Ланцюжок делегувань може бути доволі довгим, але рекурсивні виклики конструкторів не допускаються.

Оператор присвоювання переміщенням передає послідовність від одного об'єкта іншому. Його реалізація також має коректно обробляти присвоювання об'єкта самому собі:

```
Sequence& Sequence::operator=(Sequence &&other){
  if (this!=&other){
    release();
```

[4] У старих версіях мови C++ наведений тут код може не відкомпілюватися.

```
    size_=other.size_;
    capacity_=other.capacity_;
    arr=other.arr;
    other.arr=nullptr;
    other.release();
  }
  return *this;
}
```

Тоді **конструктор переміщення** можна записати так:

```
Sequence::Sequence(Sequence &&other):Sequence(){
  *this=(Sequence &&)other;
}
```

Зауваження. Наявне в коді явне перетворення змінної типу `Sequence &&` на тип `Sequence &&` *не є ані зайвим, ані помилковим*. Без нього вираз `other` розглядається компілятором як *l*-вираз і далі компілятор вибирає присвоювання копіюванням. Водночас вираз `(Sequence &&)other`, який насправді не задає жодного перетворення, уже не розглядається компілятором як *l*-вираз і за ним компілятор вибирає присвоювання переміщенням.

Приклад. Проілюструємо одну з багатьох ситуацій, коли бажано мати конструктор переміщення. Напишемо функцію `create5`, яка має повертати перші п'ять натуральних чисел, тобто послідовність 0, 1, 2, 3, 4 (за міжнародним стандартом ISO 80000-2:2009 число 0 також є натуральним):

```
Sequence create5(){
  Sequence sa;
  for(int i=0;i<5;++i)
    sa.insert(i);
  return (Sequence&&)sa;
}
```

У тілі функції створюється автоматична змінна, у яку записується шукана послідовність. Під час виконання інструкції `return` виконується конструктор, який за записаним в інструкції виразом створює об'єкт, що є результатом функції. Якщо записати `return sa;`, то буде виконано конструктор копії, під час виконання якого має відбуватися копіювання даних, зокрема виді-

лення нової ділянки пам'яті та запис у неї даних. Однак із завершенням виклику функції її автоматичні змінні знищуються. Отже, насправді *немає потреби робити копію даних*. Достатньо *перемістити* дані від автоматичної змінної функції в точку виклику. За інструкцією `return (Sequence&&)sa;` компілятор згенерує код, що використовує конструктор переміщення.

Для розглянутої функції зайве копіювання не є критичним, але у випадку більшого обсягу даних переміщення дозволяє суттєво економити час і пам'ять.

Приклад копіювання та переміщення об'єктів. Розглянемо такий код:

```
Sequence sa, sb(sa), sc(create5());  //1
sb=sa;                               //2
sb=create5();                        //3
sb=Sequence();                       //4
```

За виконання інструкції з рядка 1 створюються три об'єкти. Під час створення для об'єкта `sa` використовується стандартний конструктор, для об'єкта `sb` – конструктор копії (оскільки використовується *l*-значення), для об'єкта `sc` – конструктор переміщення (створюється за тимчасовим об'єктом, який не є *l*-значенням). Під час виконання присвоювання з рядка 2 відбувається копіювання значення одного об'єкта в інший. У рядку 3 об'єкту `sb` присвоюється значення тимчасового об'єкта, отриманого як результат виклику функції, а в рядку 4 об'єкту `sb` присвоюється анонімний об'єкт, побудований за допомогою стандартного конструктора. В обох останніх випадках об'єкти, що присвоюються, не є *l*-значеннями і для них виконується присвоювання переміщенням.

Якби для класу `Sequence` не були означені конструктор переміщення й оператор присвоювання переміщенням, то замість переміщення використовувалося б копіювання. Тоді за виконання інструкцій рядка 3 двічі відбулося б копіювання даних: спочатку при поверненні з виклику функції, а потім під час безпосередньо присвоювання.

Запишемо невелику програму, що демонструє використання розробленого класу:

```
int main(){
  Sequence sa;
  try{
    cout<<"created sequence is: "<<sa;
    sa.insert(1);
    cout<<"inserted 1, the sequence is: "<<sa;
    sa.insert(2); sa.insert(2);sa.insert(3);
    cout<<"inserted 2,2,3, the sequence is: "<<sa;
    cout<<"element with index 1 is "<< sa[1]<<endl;
    cout<<"element with index 56 is "<< sa.at(56);
  }
  catch (const exception &e) {cout<<e.what()<<endl;}
  return 0;
}
```

За її виконання буде виведено:

```
created sequence is: sequence is empty
inserted 1, the sequence is: 1
inserted 2,2,3, the sequence is: 1 2 2 3
element with index 1 is 2
There is no element with index 56 in the sequence.
```

Можна бачити, що виконання останньої інструкції блоку переривається появою винятку під час обчислення `sa.at(56)` і далі керування передається пастці, яка виводить повідомлення про помилку.

Повний код прикладу разом з розподілом по одиницях трансляції та демо-програмою наведено на сайті книги.

2.6. Неявні конструктори та оператори присвоювання у стандарті C++14

Згадаємо, що за відсутності оголошених програмістом (англ. *user-declared*) конструкторів копії/переміщення та операторів присвоювання копіюванням/переміщенням компілятор здатен неявно додавати відповідні методи до класу. Записані програмістом оголошення називають **явними**. Додані компілятором оголошення вважають **неявними** (англ. *implicit*), а додані ним означення – **означеннями за умовчанням** (англ. *defaulted*).

Стандарт C++14 вводить певні обмеження на неявні дії компілятора. Стандарт забороняє компілятору:

1) за наявності оголошеного програмістом хоча б одного з п'яти методів (конструктори копії та переміщення, оператори присвоювання копіюванням і переміщенням, деструктор) використовувати неявні конструктор переміщення та оператор присвоювання переміщенням;

2) за наявності оголошеного програмістом конструктора переміщення або оператора присвоювання переміщенням використовувати неявні конструктор копії та оператор присвоювання копіюванням.

Стандарт C++14 поки що явно не забороняє, але *не рекомендує* (англ. *deprecated*) використання неявних конструктора копії та оператора присвоювання копіюванням, якщо в класі явно оголошено хоча б один із трьох методів – конструктор копії, оператор присвоювання копіюванням, деструктор.

Нерекомендованими зазвичай позначають ті конструкції, які збираються унеможливити в майбутніх стандартах і версіях компіляторів. Код, що використовує нерекомендовані стандартом конструкції, краще не писати.

Деякі компілятори ще підтримують нерекомендовані конструкції, але деякі – уже ні. Щоб не залежати від версії компілятора у випадках, коли використання неявних методів є небажаним, відповідні методи варто декларувати явно на основі такого правила.

✓ **Правило п'яти** (англ. *Rule of five*). Якщо для класу явно оголошений хоча б один з методів:

– деструктор,
– конструктор копії,
– конструктор переміщення,
– оператор присвоювання копіюванням,
– оператор присвоювання переміщенням,

то для цього класу мають бути явно оголошені всі ці методи.

Явно означати для класів усі п'ять методів дещо обтяжливо, тому застосовують ще одну тезу.

✓ **Правило нуля** (англ. *Rule of zero*). Клас, у якому хоча б один із зазначених вище методів оголошений явно, має ексклюзивно володіти ресурсом, причому єдиним; класи, що не володіють ресурсами, жодного із зазначених методів явно оголошувати не можуть.

Контрольні запитання

1. Що таке контейнери?
2. Які найтиповіші операції над контейнерами?
3. Які переваги дає відділення інтерфейсу класу від його реалізації?
4. Із чого слід починати розробку класу?
5. Що називають клієнтами класу?
6. Що таке витоки ресурсу?
7. У чому полягає безпечне керування ресурсами?
8. Чим прямий доступ відрізняється від послідовного?
9. Чи можна в означенні класу вказувати ініціалізуючі вирази для його нестатичних полів?
10. З якою метою методи означають як вилучені?
11. Які синтаксичні особливості має для класів перевантаження операторів перетворення типу?
12. Чим конструктор копії відрізняється від конструктора переміщення?
13. У яких випадках може викликатися конструктор переміщення?
14. Чим присвоювання копіюванням відрізняється від присвоювання переміщенням?
15. Що таке делегування?
16. За яких умов у конструкторах припустиме делегування? Як воно має синтаксично оформлюватися?
17. Які існують обмеження на створення неявних конструкторів та операторів переміщення?
18. У чому полягає суть правила п'яти?
19. У чому полягає суть правила нуля?

Вправи

1. Для класу Sequence перевантажити оператори:

а) == ;

б) != .

2. Записати для класу Sequence власні означення операторів присвоювання.

3. Для класу Sequence оптимізувати код оператора присвоювання так, щоб новий масив створювався тільки якщо поточного недостатньо для збереження іншої послідовності.

4. Перевантажити для класу Sequence оператор вставки з потоку. Вважати, що введення припиняється за першої помилки зчитування.

5. За допомогою класу Sequence розв'язати задачі:

а) перевірити, чи всі числа послідовності є непарними;

б) перевірити, чи є послідовність неспадною;

в) перевірити, чи є послідовність монотонною;

г) у послідовність цілих додати задане значення після першого елемента з мінімальним значенням;

д) у кінець однієї послідовності дописати іншу.

3. КОНТЕЙНЕР З ПОСЛІДОВНИМ ДОСТУПОМ ДО ЕЛЕМЕНТІВ

Побудуємо контейнер з послідовним доступом до елементів, що зберігає множину цілих (без повторень) і надає можливість додавати новий елемент та перевіряти належність значення множині. Скористаємося зв'язаними списками.

3.1. Зв'язані списки

Нехай задано тип T. Набір даних типу T можна зберігати як масив – послідовність однотипних елементів, що займає в пам'яті неперервну ділянку пам'яті. До елементів масиву є як прямий, так і послідовний доступ. Дещо інакше можна зобразити набір елементів типу T, якщо використати як програмну модель **списки**. У найпростішому випадку до кожного елемента послідовності додається зв'язок – службові дані про те, де зберігається наступний за ним елемент. При цьому список у цілому не обов'язково займає в пам'яті неперервну ділянку пам'яті.

Елемент типу T разом зі зв'язком утворює **вузол** (англ. *node*). Перший вузол списку прийнято називати **головою** (англ. *head*), останній інколи (і не дуже точно[5]) називають **хвостом** (англ. *tail*). Щоб мати можливість обробляти список, достатньо знати його голову або хвіст, залежно від наявних зв'язків між вузлами списку (рис. 3.1).

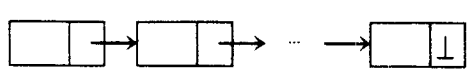

Рис. 3.1. Однобічно зв'язаний список

Структуру даних, що складається із зв'язаних між собою вузлів, називають **зв'язаною структурою даних** (англ. *linked data structure*). Найпоширенішими прикладами зв'язаних структур даних є **зв'язані списки** (англ. *linked list*), дерева тощо.

---

[5] У функціональних мовах програмування під хвостом списку розуміють усе те, що після голови.

Розглянемо зв'язані списки детальніше.

Якщо кожен вузол списку містить зв'язок з наступним вузлом, то такий список називають **однобічно зв'язаним** (англ. *singly linked*), або інколи, не дуже точно, **однозв'язним**. Список, у якому кожен вузол містить зв'язок як з наступним, так і з попереднім вузлом, називають **двобічно зв'язаним** (англ. *doubly linked*), або інколи, не дуже точно, **двозв'язним** (рис. 3.2).

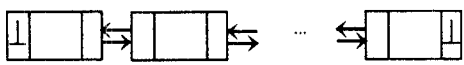

Рис. 3.2. Двобічно зв'язаний список

Список, у якому останній вузол містить зв'язок з першим, називають **циклічним** (англ. *circular*). З погляду зручності обробки циклічні списки інколи ідентифікують не за головою, а за хвостом. Циклічні списки можуть бути як однобічно, так і двобічно зв'язані.

Масиви versus списки. Масиви надають як прямий, так і послідовний доступ до своїх даних, списки – тільки послідовний, тому виконання операцій доступу до елемента за номером для масивів виконується швидше, ніж для списків.

На відміну від списків, масиви завжди займають неперервну ділянку пам'яті. Зазвичай операції вставки/вилучення елемента для масивів вимагають більше часу, ніж аналогічні операції зі списками. При вилученні, наприклад значення якого-небудь із початкових елементів масиву, необхідно зсунути на одну позицію значення всіх елементів, що йдуть за ним. Для зв'язаних списків у цьому випадку треба тільки перевизначити зв'язки одного-двох вузлів. За використання масивів ділянку пам'яті необхідно відразу виділяти під весь масив. Якщо в результаті додавання елементів розмір виділеної ділянки буде вичерпано, то необхідно виділяти нову, більшу ділянку, і переносити туди дані, що не завжди зручно та можливо. Якщо ж використовуються зв'язані списки, у яких кожен вузол є динамічною змінною, то такої проблеми не виникає.

Отже, кожен спосіб зображення набору даних має свої переваги й недоліки, і більш ефективний для однієї задачі може бути менш ефективним для іншої.

Операції над даними. В обробці наборів даних найчастіше використовуються такі операції: вставка даних у набір (англ. *insert*, *add*), вилучення (англ. *remove, delete*) з набору, пошук елемента, перегляд елемента, перевірка, чи є набір порожнім, визначення кількості елементів. Ефективність виконання цих операцій залежить від реалізації відповідної структури даних.

Зв'язаний список моделює послідовність елементів. Елементи можуть вставлятися (додаватися) і вилучатися на початку послідовності, в її кінці або взагалі в довільному місці, наприклад так, щоб послідовність даних була впорядкованою за зростанням. При іменуванні операцій з послідовностями прийнято відображати, де вони відбуваються. До назв операції з початком послідовності (головою) додають слова `head`, `begin`, `front`, а до операцій з кінцем (хвостом) – `tail`, `end`, `back` тощо.

Якщо ввести обмеження на вставку й вилучення елементів, то виникають такі структури даних, як стек і черга (див. [2]).

Нехай на значеннях типу `T` задано операцію порівняння. Якщо послідовність елементів типу `T`, яка зберігається в списку, є монотонною (неспадною згідно із заданим порівнянням), то кажуть, що список є **відсортованим** (англ. *sorted*). Узагалі для зображення монотонних послідовностей існують більш ефективні, але й більш складні структури даних, ніж відсортований список, проте для деяких задач цілком достатньо відсортованих списків.

Треба шукати компроміс між часом виконання операцій над послідовністю та часом розробки й налагоджування програми. Завжди слід пам'ятати, що ефективніші структури даних потребують більше часу на розробку й налагоджування, а отже, і більше коштів. У задачах, що не передбачають великих обсягів даних, використання складних структур даних може бути надлишковим: зайві витрачені на розробку кошти не дадуть належного економічного ефекту. Тоді доцільніше зупинитися на дуже спрощеному, неоптимальному за часом обробки даних, але дешевшому варіанті.

Зв'язані списки можуть реалізовуватися різними способами. Розглянемо реалізацію **динамічних зв'язаних списків**, тобто списків, вузли яких є динамічними змінними.

Типи для зображення вузлів списків. Вузол однобічно зв'язаного списку елементів типу `T` має складатися із двох полів: елемент типу `T` (тобто дані, що зберігаються у вузлі) і зв'язок з наступним вузлом. Як зв'язок використовують вказівник на наступний вузол (вузол, що містить наступний елемент послідовності).

Опишемо тип Node вузлів однобічно зв'язаного списку за допомогою типу структур мови C++:

```
struct Node {T data; Node *next;};
```

Ключове слово struct указує, що означається тип структур. Так само як і типи, означені з ключовим словом class, типи структур *у мові C++* є класами та складаються із членів, що можуть бути як полями, так і методами. (У класичному варіанті структури складаються тільки з полів.) Відмінність класів, означених у мові C++ як class, від тих, що означаються як struct, полягає в тому, що за не вказаного специфікатора доступу в класах (class) члени вважаються прихованими, а в структурах (struct) – відкритими.

Вузол двобічно зв'язаного списку елементів типу T має складатися із трьох полів: елемент типу T, що зберігається у вузлі, вказівники на наступний і попередній (англ. *previous*) елементи списку. Відповідний тип вузлів має вигляд

```
struct Node {T data; Node *next, *prev;};
```

У випадку нециклічних списків вказівник на наступний вузол для останнього елемента списку та на попередній вузол для першого елемента списку мають бути нульовими. Це реалізується логікою обробки вузлів, а не типом даних самого вузла.

Тип для зображення списку. Списки з розглянутими вище вузлами ідентифікуються вказівниками на перший та/або останній вузол. Чи можна додати до типу даних Node вузла операції вставки елемента в список та вилучення зі списку, розглядаючи при цьому Node як тип списків? Технічно можна, але якщо, наприклад, операцію додавання елемента в початок списку застосувати до проміжного елемента, то порушиться цілісність списку. Отже, так робити *не слід*. Для отримання повноцінного типу даних для зображення списку використовують іншу схему, у якій *список-об'єкт монопольно володіє вказівником на перший та/або останній вузол і реалізує операції маніпулювання даними*.

Зауваження щодо реалізації списків. Якщо потрібно виконувати операції вставки елементів у початок і кінець та вилучення їх із початку та кінця списку, то доцільно мати прямий доступ не тільки до першого його вузла, але й до останнього. Якщо зберігати вказівник на останній вузол списку, то після вилучення елемента з кінця списку останнім буде вузол, що пере-

дував вилученому. Отже, щоб кожного такого разу не шукати вузол, який має стати останнім (для цього доведеться послідовно перебирати всі вузли, починаючи з першого), доцільно використати двобічно зв'язаний список.

Під час реалізації вставки/вилучення елементів виникає ситуація, коли при додаванні елемента в порожній список змінюються як перший, так і останній його вузли. Щоб уникнути зайвих перевірок, як голову використовують *фіктивний вузол*, який не містить даних. Вказівники в ньому вказують на наступний (що містить перший елемент послідовності) і попередній (що містить останній елемент послідовності) вузли. Також для зручності обробки список роблять циклічним. Для ідентифікації такого списку достатньо вказівника на голову (рис. 3.3).

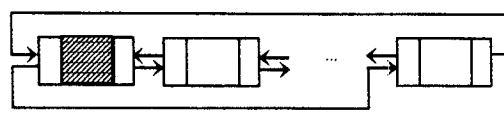

Рис. 3.3. Двобічно зв'язаний циклічний список із фіктивним вузлом-головою

Така реалізація списку менш очевидна, але дозволяє дещо економніше реалізувати операції вставки/вилучення елементів.

3.2. Клас, що зберігає множину цілих

Побудуємо структуру даних, що зберігає множину цілих і реалізує операції додавання елемента до множини, пошуку елемента, перегляду елементів, визначення кількості та перевірки, чи є множина порожньою. На основі цієї структури даних розглянемо певні синтаксичні можливості мови та стандартні прийоми програмування. Для технічного спрощення операцію вилучення елемента розглядати не будемо.

Оскільки операція вилучення елемента з множини не передбачається, то для збереження множини цілих використаємо однобічно зв'язаний відсортований список, що ідентифікується вказівником на голову. Звичайно, така реалізація не є оптималь-

ною, але більш ефективні структури даних відвертали б увагу від певних деталей, якими далі "обростатиме" наш клас.

Визначимо спочатку інтерфейс розроблюваного класу IntList, що містить стандартний конструктор, який створює порожню множину, деструктор, а конструктори копії та оператори присвоювання означимо як вилучені. У подальших релізах їхні означення можна буде змінити на власні версії цих вилучених методів. Крім того, додамо метод size, що повертає кількість елементів, і метод empty, що повертає істину, коли множина порожня. Метод find перевіряє, чи належить елемент множині, а метод insert додає елемент у множину, якщо його там немає; якщо елемент є, то метод повертає хибність.

```
class IntList{
public:
  IntList();
  IntList(const IntList&)=delete;
  IntList(IntList&&)=delete;
  IntList& operator=(const IntList&)=delete;
  IntList& operator=(IntList&&)=delete;
  ~IntList();
  unsigned size() const;
  bool empty() const;
  bool find (int elem) const;
  bool insert(int elem); //throw(bad_alloc)
    //false, якщо елемент уже наявний у списку
//решту буде додано далі
};
```

Почнемо реалізовувати наш клас. Додамо тип вузлів Node для зображення однобічно зв'язаного списку, вказівник phead на голову списку та поле size_, що зберігає кількість елементів множини:

```
protected:
  struct Node {int data; Node *next=nullptr;};
  Node *phead=nullptr;
  unsigned size_=0;
```

Для вказівника на голову списку та кількості елементів указано ініціалізатори. У такому випадку конструктор класу можна відразу означити за умовчанням:

```
IntList::IntList()=default;
```

Відсортованість списку визначається означенням методу insert. Новий елемент має додаватися не в довільне місце списку, а так, щоб список залишався відсортованим. Тоді перед додаванням слід відшукати вузол списку, після якого вставити вузол з новим елементом. Дуже схожий пошук у списку відбувається під час перевірки належності елемента множині. Щоб уникнути дублювання коду, додамо метод lastLE, що повертає останній вузол списку, значення в якому менше або дорівнює заданому. За відсутності такого вузла метод повертає nullptr:

```
protected:
  Node* lastLE(int elem) const;
  /* вказівник на останній вузол з data<=elem;
     nullptr, якщо такого немає */
```

Далі реалізуємо методи:

```
intList::~IntList(){
  Node* tmp;
  while (phead) {
    tmp=phead->next;
    delete phead;
    phead=tmp;
  }
}
unsigned IntList::size()const
  { return size_;}
bool IntList::empty()const
  { return size()==0;}
IntList::Node* IntList::lastLE(int elem) const{
  if (!phead || phead->data> elem) return nullptr;
  Node* tmp=phead;
  while (tmp->next && tmp->next->data<=elem)
    tmp=tmp->next;
  return tmp;
}
bool IntList::find(int elem) const{
  Node* tmp=lastLE(elem);
  return (tmp && tmp->data==elem);
}
bool IntList::insert(int elem){
  Node *prev=lastLE(elem);
  if (!prev){
```

```
    phead=new Node{elem, phead};
  }
  else if (prev->data==elem)
    return false;
  else prev->next=new Node{elem, prev->next};
  ++size_;
  return true;
}
```

Звернемо увагу на вираз `new Node{elem, phead}`. За ним створюється динамічна змінна типу `Node`, відкриті поля якої ініціалізуються значеннями зі списку `{elem, phead}`. Оскільки всі поля об'єкта є відкритими й відсутні явно означені конструктори, то така ініціалізація можлива.

Надалі розроблений клас зазнає певних змін. Повний остаточний код разом з розподілом по одиницях трансляції та демопрограмою наведено на сайті книги.

3.3. Ітератори

3.3.1. Поняття ітератора

Розглянемо задачу перегляду всіх даних, що зберігаються в списку (або контейнері). Наприклад, необхідно від першого до останнього вузла списку роздрукувати дані, що в ньому містяться, або знайти суму елементів.

За межами класу `IntList` доступу до вузлів списку немає, тому обійти всі вузли можна тільки в коді методів класу. Найпростіше та дещо хибне розв'язання: додати до класу списків метод, що послідовно обходить список і виводить його елементи:

```
void IntList::print()const{
  for(Node *tmp=phead;tmp;tmp=tmp->next)
    cout<<(tmp->data)<<" ";
}
```

А якщо далі виникне потреба надрукувати список якось інакше або, наприклад, тільки парні числа зі списку цілих? Додавати до типу списків метод друку в іншому форматі та метод друку всіх парних? А потім всіх непарних тощо? Це не дуже зручно.

Ще одна проблема виникає тоді, коли алгоритм вимагає послідовно рухатися по послідовності й необхідно реалізувати кілька таких проходів одночасно. Саме таким є, наприклад, тривіальний алгоритм розв'язання задачі: для кожного елемента невпорядкованої послідовності знайти кількість елементів, розташованих після нього і менших ніж він; потім надрукувати елемент, для якого ця кількість найменша (якщо таких кілька, то надрукувати найменший).

Масив можна обійти, використовуючи для переміщення цілі індекси. Для списку необхідно мати доступ до поточного вузла. Однак надати відкритий доступ до вузлів списку буде грубим порушенням принципу інкапсуляції. Додавати до класу `IntList` новий метод для кожного такого алгоритму – хибний підхід, адже неможливо передбачити всі алгоритми. Крім того, записувати в означенні класу алгоритми для специфічних задач не дуже правильно.

Для обходу контейнера можна використати механізм **ітераторів**. Ітератори дозволяють працювати з різними за внутрішньою будовою контейнерами однаковим способом. У найпростішому випадку від ітератора вимагається забезпечувати послідовний обхід контейнера, наприклад відвідування всіх його елементів. Зазвичай ітератор забезпечує доступ до поточного елемента контейнера (якщо такий є) і перехід до наступного (у порядку обходу). У випадку обходу масиву аналогом ітераторів є цілочислові індекси: за індексом можна знайти елемент масиву; щоб перейти до наступного елемента, достатньо змінити індекс. Для динамічних зв'язаних списків об'єкт класу ітераторів має зберігати вказівник на поточний вузол. Також в ітераторі можуть зберігатися додаткові дані, наприклад вказівник на голову списку, що його обходить ітератор, відстань між поточним елементом і головою списку тощо.

За допомогою ітераторів можна розв'язувати задачі перегляду не тільки всіх елементів контейнера, але й лише тих елементів, що задовольняють певну умову. Для цього ітератор треба параметризувати функцією, яка буде задавати таку умову[6].

Використання належно реалізованих ітераторів не порушує інкапсуляцію даних і дозволяє не змінювати означення класу щоразу, коли виникає потреба під час перегляду елементів контейнера виконувати над ними які-небудь інші дії або обчислення.

---

[6] Див. приклад у коді.

3.3.2. Ітератор для класу IntList

Розглянемо найпростіший ітератор для нашого списку. Додамо до класу `IntList` клас `Iterator` як член (членами класів можуть бути не тільки дані та функції, але й інші типи даних, зокрема класи). Тип, означений у класі, називають **вкладеним** (англ. *nested*).

Вкладений клас `Iterator`, окрім конструктора, який ініціалізує ітератор вузлом списку, містить такі методи:

– оператор *, який повертає значення, що зберігається в поточному вузлі, або кидає виняток типу `out_of_range`, якщо поточний вузол відсутній;

– оператор ++, який зсуває ітератор на наступний вузол, якщо поточний вузол існує, інакше залишає все як є;

– оператор зведення до типу `bool`, який повертає ознаку того, чи коректно встановлений ітератор, тобто чи існує поточний вузол.

Для реалізації зазначених операцій в об'єкті ітератора достатньо зберігати вказівник на вузол списку:

```
public: // відкрита частина класу IntList
  class Iterator{
  public:
    explicit Iterator(Node *current=nullptr);
    int operator *() const;//throw(out_of_range)
    virtual Iterator& operator ++();
    virtual operator bool() const;
  protected:
    Node *current;
  };
```

У класі `IntList` вкладений тип `IntList::Node` не є відкритим, оскільки не треба відкривати деталі реалізації. Тому працювати з його значеннями можна тільки в методах класів `IntList` та `Iterator`. Звідси за межами класу `IntList` відкритий конструктор `Iterator(Node*)` можна викликати тільки зі значенням, що нікуди не вказує, тобто з `nullptr`.

Ключове слово `explicit` (укр. *явний*) перед конструктором указує на те, що цей *конструктор не може бути використаний для неявного перетворення типів*, тобто неявне перетворення значень типу `Node*` до типу `Iterator` заборонено. Тому, наприклад, за дії оголошення `Iterator i;` вираз `i==nullptr` є синтак-

сично неправильним, оскільки значення nullptr не може бути неявно зведене до типу Iterator. Якби конструктор не був оголошений як explicit, то наведений вираз був би еквівалентний виразу i==Iterator(nullptr).

Також до інтерфейсу класу IntList додамо метод begin(), що повертає ітератор, установлений на перший елемент списку:

```
public:
 Iterator begin() const;
```

Запишемо реалізацію доданих методів. Зауважимо, що за межами класу IntList вкладені в нього типи слід позначати кваліфікованим ім'я, наприклад IntList::Iterator замість простого імені Iterator:

```
IntList::Iterator::Iterator(Node *c):current(c){}
int IntList::Iterator::operator *() const {
  if (current) return current->data;
  else throw
         out_of_range("wrong iterator position");
}
IntList::Iterator& IntList::Iterator::operator++(){
  if (current) current=current->next;
  return *this;
}
IntList::Iterator::operator bool() const{
  return current!=nullptr;
}
IntList::Iterator IntList::begin() const {
  return Iterator(phead);
}
```

3.3.3. Використання ітераторів

Розглянемо використання ітераторів для розв'язання задачі виведення всіх елементів послідовності, що зберігається в списку. Нехай діє оголошення IntList s;. Установимо ітератор на початок списку. Далі в циклі, поки ітератор коректно встановлений (не вийшов за кінець списку), будемо виводити поточний елемент та пересувати ітератор:

```
auto i=s.begin();
while (i) //тут неявно працює bool(i)
  { cout<<*i<<" "; ++i;}
cout<<endl;
```

У цьому коді змінна i оголошена без явного зазначення типу з ключовим словом auto. За такого оголошення тип змінної i визначається за типом ініціалізуючого виразу, тобто IntList::Iterator. Наявність ініціалізації у випадку використання auto обов'язкова. Ця синтаксична конструкція полегшує модифікацію коду, оскільки оголошення з auto не потребує змін.

Ітератор приховав від користувача класу IntList деталі реалізації списку. Однак обхід списку все-таки тут реалізовано нетипово. Зазвичай обхід списку реалізують як рух від однієї позиції в списку до іншої, причому обидві позиції задають ітераторами. Для цього для ітераторів визначають оператори порівняння на рівність і відношення. Для реалізації порівняння на рівність у багатьох випадках можна порівнювати адреси вузлів, на які встановлено ітератори. Реалізація відношень (<, > тощо) найчастіше використовує відстань у елементах між поточним елементом списку та головою. За наявності порівнянь ітераторів стають можливими алгоритми обходу від одного ітератора до іншого. У нашому випадку для реалізації операції виведення всього списку до класу IntList потрібно додати метод, що повертає ітератор, установлений *за кінець* списку.

Додамо до класу ітераторів оператори порівняння ітераторів на рівність:

```
class Iterator{
// те, що було раніше, залишається без змін
public:
  bool operator ==(const Iterator&) const;
  bool operator !=(const Iterator&) const;
};
```

Ітератори вважаємо рівними, якщо вони встановлені на один і той самий вузол списку.

```
bool IntList::Iterator::operator==
      (const Iterator& other) const{
  return current==other.current;
}
bool IntList::Iterator::operator!=
      (const Iterator& other) const{
  return current!=other.current;
}
```

Додамо до класу IntList метод end(), що повертає ітератор, установлений за кінцем списку. Цей ітератор задає кінець обходу списку:

```
public:
  Iterator end() const;
```

Реалізується він так:

```
IntList::Iterator IntList::end()const {
  return Iterator();
}
```

Тепер можлива функція output, що виводить значення елементів списку з проміжку [from; to) у вихідний потік f.

```
std::ostream& output(std::ostream& f,
     IntList::Iterator from, IntList::Iterator to){
  for(auto i=from;i!=to;++i) f<<*i<<" ";
  return f;
}
```

Використовуючи цю функцію, перевантажимо оператор << вставки в потік:

```
ostream& operator <<(ostream& f, const IntList& c){
  return output(f, c.begin(), c.end());
}
```

Оголошення наведеного оператора можна додати до заголовного файлу з означенням класу IntList, але *за межами класу*. Додавання до класу, який відповідає за збереження даних, елементів введення/виведення суперечить такому принципу.

✓ *Дані, засоби їх візуалізації та засоби керування ними мають відділятися одне від одного.*

Зокрема, внутрішнє зображення даних (модель даних) має не залежати від засобів візуалізації та інтерфейсу користувача, що керує цими даними. Більш того, одна й та сама модель даних може використовуватися з *різними* засобами візуалізації та інтерфейсами керування. Такий підхід до проектування збільшує можливості повторного використання коду і спрощує його розробку та модифікацію. Зокрема, одне й те саме зображення даних можна використовувати з різними засобами візуалізації; крім того, модифікація реалізації зображення даних (за незмінного інтерфейсу) не викликатиме необхідність змін у засобах візуалізації.

Зауваження. Результатом пошуку елемента в контейнері дуже часто стає відповідно встановлений ітератор. Змінимо оголошення та означення методу find класу IntList, щоб він повертав ітератор, установлений на знайдений вузол (якщо елемент знайдено), або за межі списку (якщо такого елемента в контейнері немає):

```
Iterator find(int elem) const;
IntList::Iterator IntList::find(int elem) const{
  Node *tmp=goLE(elem);
  if (tmp && tmp->data!=elem)
    tmp=nullptr;
  return Iterator(tmp);
}
```

Зауваження. Означення функції output не залежить від особливостей реалізації контейнера й може бути використане з ітераторами різних структур даних (за умови заміни типів ітераторів у заголовку).

Зауваження. У випадку двобічно зв'язаних списків ітератор може рухатися списком як у прямому напрямку (від початку до кінця), так і в оберненому. Тоді можна розглядати:

– однонаправлений ітератор, що рухається в прямому напрямку,

– однонаправлений ітератор, що рухається в оберненому напрямку,

– двонаправлений ітератор, що може рухатися в обох напрямках.

Зауваження. Після зміни контейнера ітератор може стати "недійсним", наприклад тому, що відповідний вузол списку було вилучено. Цей факт треба враховувати.

3.3.4. Ітератор, що обходить елементи за певною умовою

Використовуючи механізм успадкування, побудуємо ітератор, який обходить не всі елементи контейнера, а тільки ті, що відповідають деякій умові, наприклад парні або ті, що націло діляться на 3. Зрозуміло, що кожна нова умова не повинна породжувати новий клас ітераторів. Тому побудуємо клас ітераторів, параметризованих умовою. Як умову використовуватимемо функцію, яка за елементом, що зберігається в контейнері, повертає логічне значення. У нашому прикладі функція має приймати аргумент типу int, а повертати значення типу bool.

Новий клас IteratorC означимо всередині класу IntList і зробимо його нащадком класу IntList::Iterator. Єдиною відміною в його інтерфейсі є те, що при конструюванні він додатково буде приймати умову – функцію, яка за значенням типу int (тип елементів, що зберігаються в контейнері) повертає значення типу bool.

Міркуючи про реалізацію, зазначимо, що оператор ++ для цього класу має переходити не на наступний елемент, а *на наступний елемент, що відповідає умові*. Тому його треба перевантажити. Крім того, об'єкт класу має зберігати вказівник на функцію, що реалізує перевірку умови: до прихованої частини додамо поле condition. Отже, отримуємо таке означення класу:

```
public:
  class IteratorC:public Iterator{
  public:
    explicit IteratorC(Node *current=nullptr,
                       bool (*f)(int)=nullptr);
    IteratorC& operator ++();
  protected:
    bool (*condition)(int);
  };
```

До відкритої частини класу IntList відразу додамо метод, який повертає ітератор типу IteratorC, установлений на перший елемент списку, що відповідає умові:

```
IteratorC beginC(bool (*f)(int)) const;
```

Запишемо реалізацію цих методів:

```
IntList::IteratorC
IntList::beginC(bool (*f)(int)) const {
  return IteratorC(phead,f);
}
IntList::IteratorC&
IntList::IteratorC:: operator ++() {
  do
    { Iterator::operator++();}
  while(operator bool() && !condition(operator*()));
  return *this;
}
```

Для переходу до наступного елемента рухаємося списком за допомогою відповідного методу базового класу, поки не буде досягнуто кінця списку або виконано умову:

```
IntList::IteratorC::
```

```
IteratorC(Node * c, bool (*f)(int)):
            Iterator(c), condition(f) {
  if (!f) condition=[](int)
    {return true;};
  if (operator bool() && !condition(operator*()))
    operator++();
}
```

Тут використано **лямбда-вираз** `[](int){return true;}`, за яким компілятор генерує функцію, що приймає аргумент типу `int` і повертає значення `true`. Тип результату цієї функції визначається компілятором як `bool` за типом значення, що повертається. У дещо загальнішому випадку в круглих дужках указується перелік формальних параметрів функції, за яким розташовується її тіло (послідовність інструкцій у фігурних дужках). Детальний синтаксис лямбда-виразів див. у документації мови C++.

Під час виконання умовної інструкції, якщо значенням `f` є нульова адреса, то змінній `condition` присвоюється адреса згенерованої компілятором функції.

Лямбда-вирази зручні, коли необхідно швидко створити нескладну за кодом функцію. Розглянемо ще один приклад їх використання.

Припустимо, що стоїть задача друкування всіх елементів списку `IntList s`, які націло діляться на 3. Можна створити функцію, що перевіряє умову подільності на 3:

```
bool div3(int n){return n%3==0;}
```

та з її використанням розв'язати задачу:

```
auto i=s.beginC(div3);
while (i) { cout<<*i<<" "; ++i;}
```

Однак можна використати лямбда-вираз і не створювати функцію явно:

```
auto i=s.beginC([](int n){return n%3==0;});
while (i) { cout<<*i<<" "; ++i;}
```

Можна помітити, що обхід списку ітераторами різних типів має багато спільного. Однак якщо записати

```
output(cout,
  st.beginC([](int n){return n%3==0;}), st.end());
```

то будуть виведені всі елементи поспіль, починаючи з першого, що ділиться на 3. Це пов'язано з тим, що локальна змінна `i` функції `output` є ітератором базового типу, оскільки її тип визначається за типом відповідного параметра функції.

Дещо змінимо цю функцію:

```
ostream& output(std::ostream& f,
ostream& output(std::ostream& f,
                IntList::Iterator &&from,
                IntList::Iterator &&to){
  for(; from!=to; ++from)
    f<<*from<<" ";
  return f;
}
```

Тепер `from` є посиланням, а локальну змінну `i` усунуто. Завдяки цьому при переході до наступного елемента використовується версія віртуального методу `operator ++`, що відповідає типу об'єкта, а не типу змінної в оголошенні методу. За таких змін за попередньою інструкцією буде виведено саме те, що треба.

Зауваження. Доволі типовою є ситуація, коли на місця останніх двох аргументів функції `output` підставляються *r*-значення, наприклад

```
output(f, c.begin(), c.end());
```

Тому останні два формальні параметри функції `output` мають тип посилань на *r*-значення (&&). За використання посилань на *l*-значення (&) підставити *r*-значення неможливо, а за використань посилань на незмінюване *l*-значення (**const** `IntList::Iterator &`) у тілі функції `output` не можна було б змінювати значення `from`, і ми повернулися б до необхідності вводити локальну змінну.

Якщо ж функцію `output` необхідно викликати з параметрами, що є *l*-значеннями, то їх необхідно явно перетворити до типу `IntList::Iterator &&`, як у такому коді:

```
IntList::Iterator i1=st.begin(), i2=st.end();
output(cout, (IntList::Iterator&&)i1,
             (IntList::Iterator&&)i2);
```

Приклад використання розробленого класу в програмі. Наведена нижче програма додає в контейнер числа 5, 1, 3, 2, 1, 7, 12, після чого виводить вміст контейнера. Далі програма виконує пошук значень 3, 30, -5 і виводить результати пошуку. Після цього за допомогою функції `output` виводяться всі числа з контейнера, а потім тільки ті числа, що діляться націло на 3. В останньому виведенні елементів контейнера умова задається лямбда-виразом:

```
bool div3(int n){return n%3==0;}
int main(int argc, char *argv[]){
  IntList st;
  st.insert(5); st.insert(1); st.insert(3);
  st.insert(2); st.insert(1); st.insert(7);
  st.insert(12);
  cout<<"output list using <<: "<<st<<endl;
  cout<<boolalpha;
  cout<<"find(3) "<<st.find(3)<<endl;
  cout<<"find(30) "<<st.find(30)<<endl;
  cout<<"find(-5) "<<st.find(-5)<<endl;
  cout<<"output all list using output function: ";
  output(cout,st.begin(), st.end());cout<<endl;
  cout<<"output elements satisfied n%3==0: \n";
  cout<<" using output function and \
              div3 function: ";
  output(cout,st.beginC(div3), st.end());
  cout<<endl;
  cout<<" using output function and \
            lambda-expression: ";
  output(cout,
       st.beginC([](int n){return n%3==0;}),
       st.end());
  cout<<endl;
  return 0;
}
```

Повний остаточний код класу разом з розподілом по одиницях трансляції та демо-програмою наведено на сайті книги.

У наступному розділі буде розглянуто інший спосіб того, як уникнути дублювання коду, що може виникати через необхідність обходити контейнер ітераторами різних типів.

Контрольні запитання

1. У чому відмінність між однобічно та двобічно зв'язаними списками?
2. Які списки називають динамічними?

3. Які переваги та недоліки має використання динамічних зв'язних списків порівняно з використанням масивів?

4. Запишіть тип, що зображує вузол однобічно зв'язаного динамічного списку.

5. Запишіть тип, що зображує вузол двобічного зв'язаного динамічного списку.

6. Який список називають циклічним?

7. Чи підійде для зображення вузла циклічного списку тип, що використовується для вузлів аналогічного списку, але без умови циклічності?

8. Який список називають відсортованим?

9. Чи підійде для зображення вузла відсортованого списку тип, що використовується для вузлів аналогічного списку, але без умови відсортованості?

10. Чому тип для зображення вузла динамічного зв'язаного списку не варто використовувати як тип для зображення списку?

11. Яке основне призначення ітераторів?

12. Що задає ключове слово `explicit` в оголошенні конструктора?

13. Який тип буде мати змінна, в оголошенні якої замість конкретної назви типу вказано `auto`?

14. Чому дані, засоби їх візуалізації та засоби керування ними рекомендують проектувати як окремі частини коду?

Вправи

1. Для класу `IntList` написати власні означення:
   а) конструктора копії, що створює копію даних списку;
   б) конструктора переміщення, що переміщує дані у створюваний об'єкт;
   в) оператора присвоювання копіюванням, що копіює дані списку;
   г) оператора присвоювання переміщенням, що переміщує дані від одного об'єкта до іншого.
2. Для класу `IntList` перевантажити оператор `==`.

3. Для класу `IntList` перевантажити оператор <= так, щоб він виконував перевірку включення однієї множини в іншу.

4. Для класу `IntList` перевантажити оператори +, *, - так, щоб вони обчислювали об'єднання, перетин і різницю множин. Результат має зберігатися в новій множині.

5. Написати функцію, яка за множиною цілих, зображеною об'єктом класу `IntList`, знаходить суму її елементів.

6. Змінити клас `Iterator` так, щоб до нього можна було додати операції порівняння < та <=. Додати їх.

7. Додати до класу `IntList` метод, що вилучає заданий елемент (за наявності).

4. ШАБЛОНИ

Доволі часто функції та абстрактні типи даних відрізняються лише типом, який лежить у їх основі. Наприклад, якщо для типу даних T означено оператор <, то можна написати функцію min, що повертає мінімальне з двох значень цього типу. Якщо означено присвоювання, то можна обміняти місцями значення двох змінних цього типу (функція swap).

Тіла цих функцій можуть виглядати так:

```
if (arg1<arg2) return arg1; else return arg2;//min
T tmp; tmp=arg1; arg1=arg2; arg2=tmp;        //swap
```

Можна бачити, що описані дії не залежать від типу даних, над якими вони виконуються. Тому, щоб не писати майже однаковий код і уникнути дублювання, варто *параметризувати код типами.* Цього можна досягти використанням **шаблонів** (англ. *template*).

4.1. Шаблон функції

Розглянемо шаблон функції myMin обчислення мінімального з двох значень, тип яких є параметром шаблону.

У файлі templates.cpp запишемо таке:

```
template <class T>
const T& myMin(const T &arg1,const T &arg2){
  if (arg1<arg2) return arg1; else return arg2;
}
```

Функція, означена в шаблоні, називається **шаблонною**. Оголошення й означення шаблону функції має починатися зі службового слова template, за яким у кутових дужках іде перелік типів, які є **параметрами шаблону**[7]. Назві типу передує службове слово class, яке в даному контексті є синонімом слова "тип". Кожен указаний тип має використовуватися в заголовку шаблонної функції.

Якщо шаблон функції використовується для якогось типу у вигляді виклику шаблонної функції з аргументами цього типу, то компілятор генерує версію функції з підставленим у неї відповідним типом. У такій ситуації кажуть про **інстанціювання шаблону**,

[7] Шаблони, параметризовані значеннями, тут не розглядаємо.

а відповідну функцію називають **інстанційованою** (англ. *instantiated function*) та **спеціалізацією** (англ. *specialization*) **шаблону**. Якщо шаблон використовується з якимось типом кілька разів, то для цього типу генерується єдина функція. Для типів, з якими функція не використовується, версії функції не генеруються.

✓ Якщо шаблон функції інстанціюється для деякого типу, то для останнього мають бути означені всі операції, що застосовуються до нього в цій функції.

Зауваження. У шаблон замість типу `T` можна підставити майже довільний тип, у тому числі той, що керує складною структурою даних або не підтримує операцію копіювання. Тому в шаблонах функцій як типи формальних параметрів, на місця яких мають підставлятися значення типів-параметрів шаблону, часто використовують посилання. Це робиться для того, щоб уникнути зайвих копіювань значень під час виклику інстанційованої функції. Якщо у виклику аргументи мають не змінюватися, то замість `T` доцільно використати `const T&` (посилання на незмінюване значення типу `T`), інакше варто використати `T&` (або `T&&`).

Багато компіляторів не дозволяють окрему компіляцію шаблонів. Тому для того, щоб компілятор міг згенерувати інстанційовану функцію, при використанні шаблонів необхідно включати файл із шаблоном. Виходячи з цього, на початку файлу `templates.cpp` слід додати такі директиви:

```
#ifndef TEMPLATES_CPP
#define TEMPLATES_CPP
```

Відповідно в кінці файлу додати ще одну директиву:

```
#endif //TEMPLATES_CPP
```

Замість `TEMPLATES_CPP` можна взяти якесь інше ім'я, але воно не може використовуватися в інших одиницях трансляції. Тут ім'я `TEMPLATES_CPP` природно зв'язано з іменем файлу. Для файлів із означеннями класів доволі часто в такій ситуації вибирають ім'я, що включає назву класу.

Наведені директиви компілятора гарантують, що записаний у файлі шаблон не включиться більше одного разу (у реальних проектах можливі доволі складні ланцюжки включень, за яких той самий файл включається кілька разів). Альтернативою може бути розміщення на початку такої директиви:

```
#pragma once
```

Директива #pragma once підтримується компіляторами фірми Microsoft та деякими іншими, але, на відміну від першого варіанта, стандарт не гарантує, що кожен компілятор мови C++ має її зрозуміти.

Розглянемо приклад використання шаблону у файлі demo.cpp:

```
#include <string>
#include "templates.cpp"
using namespace std;
int main(){
  int n,n1=10,n2=15;
  double x,x1=10,x2=15;
  string s,s1="as",s2="12";
  n=myMin(n1,n2);      //myMin для int
  x=myMin(x1,x2);      //myMin для double
  s=myMin<string>(s1,s2); //myMin для string
  n=myMin<int>(n1,n2);  //myMin для int
  x=myMin<double>(n1,n2); //myMin для double
  return 0;
}
```

Бачимо, що виклик шаблонної функції можна записати без явного зазначення типу, що має підставлятися замість типу T. У такому разі тип визначається за типом аргументів виклику. Наприклад, у виразі myMin(n1,n2) викликається відповідна функція для типу int, а у виразі myMin(x1,x2) – для типу double. Однак тип можна вказати і явно. Наприклад, у виразі myMin<string>(s1,s2) буде викликано версію функції для типу string, а у виразі myMin<double>(n1,n2) – для типу double, незважаючи на типи аргументів.

Шаблон функції, що обходить контейнер. Розглянемо шаблон функції, що обходить контейнер і виводить його вміст у вихідний потік за допомогою ітераторів:

```
template<class Iterator> ostream& output
        (ostream &f, Iterator from, Iterator to){
  for(auto i=from; i!=to; ++i) f<<*i<<" ";
  return f;
}
```

Така функція може бути використана не тільки для ітераторів класу IntList, але й для довільних ітераторів, що підтримують операції =, ++, != та *.

За наявності оголошення `IntList s;` для виведення елементів списку, що відповідають умові, можна записати:

```
output(cout, s.beginC([](int n){return n%3==0;}),
       IntList::IteratorC());
```

Використаний тут лямбда-вираз розглядався в кінці розділу 3. У наведеному коді суттєво використовується те, що для обох типів ітераторів (`Iterator` та `IteratorC`) кінець обходу можна задати значенням ітератора, який установлено в нульову адресу. Змінимо шаблон:

```
template<class Iterator1, class Iterator2> ostream&
output(ostream &f, Iterator1 from, Iterator2 to){
  for(auto i=from; i!=to; ++i) f<<*i<<" ";
  return f;
}
```

Зазначимо, що типи `Iterator1` та `Iterator2` мають бути сумісними для `!=`. Тепер можливий виклик

```
output(cout, s.beginC([](int n){return n%3==0;}),
       s.end());
```

4.2. Шаблон класу на прикладі контейнера

Для побудови контейнера `IntList`, що зберігає множину цілих, насправді не дуже важливо, що зберігаються саме цілі числа. Замінивши тип елементів множини, можна отримати множину елементів іншого типу. Отже, природно параметризувати клас `IntList` типом елементів `T`, які зберігатиме множина. Дамо шаблонному класу (англ. *template class*) ім'я `SList` (від англ. *sorted list* – відсортований список).

Аналогічно шаблону функції оголошення та означення шаблону класу починається зі службового слова `template`, за яким у кутових дужках іде перелік типів – параметрів шаблону.

Порівняно з `IntList` внесемо в клас `SList` деякі зміни.

1. Тип `T` може бути достатньо складно організований і не підтримувати операцію копіювання. Тому всі параметри методів і значення, що повертаються з методів, зробимо посиланнями. У

випадку, коли зміни не передбачаються або можуть порушити цілісність даних, використаємо *посилання на незмінюване значення*, тобто `const T&`.

2. Вилучення з класу `IntList` конструкторів копії та переміщення було зроблене лише заради спрощення прикладу. На практиці такий підхід у більшості випадків суттєво обмежує можливості використання класів, тому означимо власні версії конструкторів копії та переміщення. Заради спрощення нехай обидва вони підтримують семантику переміщення.

3. Результатом методу додавання елемента в контейнер насправді може бути не тільки ознака того, чи було виконано додавання або елемент уже був у наявності, але й ітератор, установлений на доданий елемент. Змінимо метод `insert` так, щоб він повертав пару значень: ітератор та ознаку успішності. Типом параметра зробимо `T&&`, тобто представлене ним значення переміщується в структуру даних. Зазвичай потрібен ще метод вставки з аргументом типу `const T&`, який копіює значення в контейнер. Додамо і його.

4. Ця зміна є суто синтаксичною. У межах означення класу `SList` залишимо тільки оголошення вкладених класів `Iterator`, `IteratorC`, а їхні означення розмістимо окремо.

Означення класу `SList` набуде вигляду

```
template <class T> class SList{
public:
  class Iterator;
  class IteratorC;
  SList()=default;
  SList(SList&);
  SList(SList&&);
  SList& operator=(const SList&)=delete;
  SList& operator=(SList&&)=delete;
  ~SList();
  unsigned size() const;
  bool empty() const;
  pair<Iterator,bool> insert(const T& elem);
  pair<Iterator,bool> insert(T&& elem);
    //throw(bad_alloc)
    //false, якщо елемент уже наявний у списку
protected:
  struct Node {T data; Node *next=nullptr;};
```

```
  Node *phead=nullptr;
  unsigned size_=0;
  Node* lastLE(const T &elem) const;
    /* вказівник на останній вузол,
         для якого data<=elem;
       nullptr, якщо такого немає */
public:
  Iterator begin() const;
  Iterator end() const;
  Iterator find(const T &elem) const;
  IteratorC beginC(bool (*f)(const T&)) const;
};
```

Тут використано стандартний бібліотечний клас `pair` із простору імен `std`. Для роботи з ним слід включити стандартний бібліотечний файл `utility`. Клас `pair` є шаблонним класом, параметризованим двома типами. Він призначений зберігати пари значень, типи яких задаються параметрами шаблону. Клас `pair` надає доступ до компонентів пари через методи `first` та `second`.

У результаті підстановки конкретних типів `Iterator` та `bool` замість відповідних типів-параметрів шаблону (тобто *інстанціювання* шаблону) утворюється конкретний тип `pair<Iterator, bool>`, інстанційований від класу `pair`. Цей тип зберігає пари значень, де першою компонентою є значення з типу `Iterator`, а другою – значення з типу `bool`.

Означення вкладеного класу `Iterator` запишемо за межами означення класу `SList`. Для вкладеного класу слід використовувати *кваліфіковане ім'я*, що складається безпосередньо з назви класу та назви класу, у якому його було оголошено. Серед методів зміниться тільки оголошення оператора `*`. Типом значення, що повертається, має стати посилання. Оскільки структура підтримує властивість відсортованості, то зміна значення елемента в загальному випадку може порушити цю властивість, а тому результатом має бути `const T&`.

Наведемо тільки зміни порівняно з аналогічними класами для `IntList`:

```
template<class T> class SList<T>::Iterator{
  //решта залишається без змін
  const T& operator *() const;
};
```

Тепер запишемо означення вкладеного класу IteratorC. Аналогічно попередньому наведемо тільки зміни:

```
template<class T> class SList<T>::IteratorC:
                        public SList<T>::Iterator{
  //решта залишається без змін
  explicit IteratorC(Node *current=nullptr,
                     bool (*f)(const T&)=nullptr);
  //решта залишається без змін
  bool (*condition)(const T&);
};
```

У мові C++ клас може бути похідним від інстанційованого класу, але не від шаблону класу. Тому запис public SList::Iterator на початку означення класу був би *синтаксично неправильним*. Базовий клас не обов'язково має бути інстанційованим для типу T: він може бути інстанційований для довільного типу, включаючи T, наприклад public SList<double>::Iterator тощо. Однак це не відповідає призначенню класу IteratorC, тому так робити не будемо і за базовий клас візьмемо SList<T>::Iterator.

Додамо шаблони функцій, що здійснюють виведення:

```
template <class Iterator1, class Iterator2>
  ostream& output(ostream& f, Iterator1 from,
                  Iterator2 to);
template <class T> ostream& operator <<(ostream&,
                const SList<T>&);
```

Далі слід записати означення методів і функцій виведення. Означення методів шаблонного класу має починатися з template <class T>. За межами означення шаблонного класу та переліків формальних параметрів і тіл його методів для його іменування слід використовувати SList<T>, а не просто SList. Ці зміни необхідно внести в усі означення.

Наведемо реалізацію частини методів класу (повний код прикладу шаблону класу див. на сайті книги).

Запишемо означення конструктора переміщення:

```
template <class T> SList<T>::SList(SList&& other){
  phead=other.phead;
  size_=other.size_;
  other.phead=nullptr;
  other.size_=0;
}
```

Конструктор копії реалізуємо так, щоб він виконував переміщення. Для уникнення дублювання коду скористаємось конструктором переміщення. Для цього достатньо перетворити тип посилань і використати делегування:

```
template <class T> SList<T>::SList(SList& other):
    SList<T>((SList<T>&&)other){}
```

У реалізаціях методів класу IntList для типу елементів використовувалися оператори <, ==, <=, !=. Щоб шаблонний клас SList можна було використати з типом T, для останнього також мають бути означені зазначені оператори. Однак <=, != можна виразити через <, ==. У методах класу SList це варто зробити. Тоді для класу T буде достатньо перевантажити тільки < та ==.

Ще однією синтаксичною особливістю шаблонів є те, що за межами шаблону класу вираз SList<T>::Node не позначає вкладений тип. Синтаксис мови C++ вимагає явно вказувати, що вираз позначає тип. Для цього використовується ключове слово typename. Правильним позначенням типу буде typename SList<T>::Node. Як приклад запишемо означення методу lastLE:

```
template <class T> typename SList<T>::Node*
SList<T>::lastLE(const T& elem) const{
  if (!phead || elem<phead->data) return nullptr;
  Node* tmp=phead;
  while (tmp->next &&
   (tmp->next->data<elem || tmp->next->data==elem))
    tmp=tmp->next;
  return tmp;
}
```

Використання простого імені типу Node *в тілі* означення методу та переліку формальних параметрів є припустимим. У цьому випадку шаблонний клас Node інстанціюється тим типом T, який указано у кваліфікованому імені SList<T>::lastLE методу.

В означення решти методів, типом результату яких є вкладені класи, слід внести аналогічні зміни, додавши typename.

Запишемо означення методу insert:

```
template <class T>
pair<typename SList<T>::Iterator,bool>
SList<T>::insert(T &&elem){
  Node *prev=lastLE(elem);
  Node *added;
  if (!prev){
```

```
      phead=added=new Node{elem, phead};
    } else
    if (prev->data==elem)
      return make_pair(Iterator(prev),false);
    else
      prev->next=added=new Node{elem, prev->next};
    ++size_;
    return make_pair(Iterator(added),false);
  }
```

Для утворення пар значень можна використати або конструктор `pair<Iterator,bool>` інстанційованого від `pair` класу (зазначення типів тут обов'язкове), або стандартний шаблон функції `make_pair`. Використання шаблону `make_pair` синтаксично зручніше, оскільки не потребує від програміста явно вказувати типи, якими інстанціюється шаблон. Альтернативою останній інструкції могла б бути така:

```
return pair<Iterator,bool> (Iterator(added),false);
```

Шаблон `make_pair` оголошений у стандартному просторі імен; для його використання слід включати стандартний бібліотечний файл `utility`.

Ще раз нагадаємо, що для використання шаблону необхідно включати не тільки файл із означенням класу, але й файл із означенням його методів. Тому означення методів краще записати відразу після означення класу. *Створювати окремий заголовний файл сенсу немає*. Запишемо шаблон у файл `SList.cpp`.

За включення шаблону

```
#include "sList.cpp"
```

можливий такий код:

```
SList<int> st;
```

Типом змінної `st` є інстанційований типом `int` шаблонний клас `SList`. Після цього оголошення зі змінною можна виконувати всі дії, наведені в демо-програмі в кінці розділу 3, наприклад `st.insert(5);` тощо. Повний код прикладу наведено на сайті книги.

Контрольні запитання

1. Для чого використовуються шаблони?
2. Укажіть синтаксичні особливості запису шаблонів функцій.
3. Що таке інстанціювання шаблону?
4. Якщо шаблон функції в коді програми три рази використовується з тим самим типом, то скільки функцій для нього буде згенеровано компілятором?

5. Для чого параметри шаблонних функцій, на місця яких мають підставлятися значення типу, що є параметром шаблону, роблять посиланнями?

6. Яке призначення має стандартний бібліотечний клас `pair`?

7. Чому оператор * для класу `SList::Iterator` не може повертати значення типу `T&`? Чи міг би він повертати значення цього типу, якби список не був відсортованим? Чи змінилося б при цьому тіло методу для оператора *?

8. Як за межами шаблонного класу позначити вкладений у нього тип?

Вправи

1. Змінити означення конструктора копії класу `SList` так, щоб він копіював список у створюваний об'єкт.

2. Для класу `SList` написати власні означення:

а) оператора присвоювання копіюванням, що копіює дані списку;

б) оператора присвоювання переміщенням, що переміщує дані від одного об'єкта іншому.

3. Для класу `SList` написати означення методу `insert`, що не переміщує, а копіює значення, яке додається в структуру даних.

4. Для класу `SList` перевантажити оператор ==.

5. Для класу `SList` перевантажити оператор <= так, щоб він виконував перевірку включення однієї множини в іншу.

6. Для класу `SList` перевантажити оператори +, *, - так, щоб вони обчислювали об'єднання, перетин і різницю множин. Результат має зберігатися в новій множині.

7. Додати до класу `SList` метод, що вилучає заданий елемент (за наявності).

8. З використанням шаблонного класу `SList` реалізуйте тип, що моделює множину дійсних. Модифікуйте оператор вставки в потік так, щоб множина виводилася у звичному для математики вигляді, наприклад: {1.9, 3.2, 10.4} (перелік елементів множини у фігурних дужках). Напишіть програму, що додає до множини числа 0, π, e та далі виводить її вміст.

9. З використанням шаблонного класу `pair` запишіть оголошення типу, що зберігає пари дійсних. Для цього типу означте операції порівняння (==, !=, <) і напишіть оператор вставки пари дійсних у потік так, щоб пара виводилася у звичному для математики вигляді, наприклад: (3.14, 2.78) (два елементи через кому у фігурних дужках).

10. Реалізуйте тип, що моделює множину пар дійсних. Напишіть програму, що додає в множину вершини одиничного квадрата, а потім виводить вміст множини.

11. Реалізуйте шаблон класу, що зберігає циклічний двобічно зв'язаний список, у якому голова списку є виділеним вузлом, що не містить даних. Мають підтримуватися операції додавання елемента на початок списку, вилучення елемента з кінця списку, визначення кількості елементів, перевірки, чи є структура даних порожньою, пошуку елемента в списку, а також операції отримання ітераторів, установлених на початок списку та за його кінець. Для розробленого шаблонного класу означте оператор вставки в потік.

5. ЗАДАЧА ПРО ОБРОБКУ ТЕКСТУ

Щоб розглянути використання шаблонних класів на прикладі, повернемося до *задачі про лабораторні роботи.*

5.1. Постановка задачі

Дано текстовий файл, що містить інформацію стосовно виконання лабораторних робіт студентами. У кожному рядку файлу через роздільник указується така інформація: 1) прізвище, ім'я та по батькові (ПІБ) студента; 2) кодова назва лабораторної роботи; 3) кількість набраних балів. При цьому студенти могли здавати лабораторні роботи кілька разів, і тоді як остаточний за лабораторну роботу береться бал найкращої спроби.

Необхідно для кожного студента вивести інформацію щодо суми набраних балів із розшифровкою по лабораторних роботах із зазначенням бала найкращої спроби та кількості спроб.

Нехай як роздільник у вхідному файлі використовується `;`. Перед тим, як проектувати код, наведемо приклад вмісту вхідного файлу `text.txt`:

```
Sidorov I I; Lab4; 10
Ivanchuk I I; Lab1; 12
Ivanchuk I I; Lab1; 1
Petrenko P P; Array;14
Ivanchuk I I; Lab1; 13
Ivanchuk I I; Lab4; 8
Ivanchuk I I; Lab3; 2
```

Очевидним обмеженням на вхідні дані є те, що роздільник має бути відсутнім усередині інформації (зокрема, він не може бути частиною назви лабораторної роботи), оскільки інакше вміст текстового файлу не можна однозначно інтерпретувати. Наприклад, рядок

```
Ivanchuk I I; Lab;4; 8
```

можна сприймати по-різному: вважати його помилковим (за лабораторну роботу `Lab` отримано 4 бали, а далі записано щось зайве) або розглядати `Lab;4` як назву. Оскільки обидві ситуації можливі, то найчастіше такий рядок вважають помилковим.

5.2. Загальна структура розв'язку

Сформулюємо загальний алгоритм розв'язання:

1) завантажити дані з файлу в контейнер;
2) вивести інформацію, що зберігається в контейнері.

Нехай контейнером буде клас Students. Він відповідатиме лише за зберігання інформації в пам'яті. Вхідна інформація може надаватися в різних форматах і від зовнішнього зображення маніпулювання даними в оперативній пам'яті ніяк не залежить. Тому *засоби, що безпосередньо працюють із зовнішнім зображенням, мають не бути частиною контейнера*. Отже, завантаження даних із файлу в контейнер має реалізовувати не метод контейнера, а інший клас, наприклад Loader. Тоді зміна вхідного формату не викличе змін у контейнері. Натомість буде достатньо змінити тільки клас-завантажувач. Якщо інформація одночасно надходитиме у файлах із різними форматами, то замість того, щоб збільшувати кількість методів контейнера, відстежувати й перебудовувати всі проекти, де використовується контейнер, буде достатньо для кожного формату мати свій клас-завантажувач. Ще однією перевагою такого підходу до проектування є те, що *збільшуються можливості сумісної розробки*: клас-контейнер і клас-завантажувач можна розробляти паралельно.

Якщо клас Students має нічого не знати про клас Loader, то клас Loader повинен мати доступ до методів, що реалізують запис інформації в контейнер Students, але при цьому він *має не залежати від способу зображення інформації в контейнері*.

Будемо проектувати розв'язання нашої задачі в стилі об'єктно-орієнтованого програмування. Отже, об'єкти класу Students зберігатимуть інформацію про студентів. Тоді елементами контейнера будуть об'єкти класу, наприклад Student, що містять інформацію про конкретного студента.

З погляду нашої задачі інтерфейс і внутрішня будова класу Student можуть бути майже довільні. Отже, не дуже правильно використовувати об'єкти цього класу як аргументи відкритих методів контейнера Students хоча б тому, що якщо виникне бажання або приховати, або змінити будову класу Student, то це суттєво вплине на частину коду, що реалізовує завантаження даних із файлу в контейнер.

Контейнер Students може підтримувати різні притаманні контейнерам операції, зокрема операцію "додати студента". Однак, виходячи з попередніх міркувань, додамо до класу Students такий відкритий метод:

```
void addLabAttempt(const string &name,
    const string &labCode, unsigned score);
```

Він додає в контейнер Students інформацію про спробу здати лабораторну роботу студентом.

5.3. Завантаження даних із файлу в контейнер

Клас Loader відповідає тільки за завантаження даних із файлу в контейнер типу Students. Інтерфейс класу Loader складається з єдиного методу

```
class Loader{
public:
  bool load(Students&, const string &fname);
  //ігнорує рядки з неправильним вмістом
  //throw(runtime_error) за помилок читання файлу
```

Вважаємо, що метод ігнорує рядки з неправильним вмістом, а у випадку помилок роботи з файлом кидає виняток. Це зазначено в коментарі. Результатом є ознака успішності виконання (тобто чи весь файл був успішно завантажений).

Деталізуємо алгоритм методу. Відмовляємося від повномасштабного синтаксичного аналізу й після відкриття файлу з іменем fname рядками читаємо його вміст. Кожен рядок розбираємо та записуємо інформацію в контейнер. У кінці повідомляємо підсумки обробки вхідного файлу:

```
відкрити файл з іменем fname для читання;
поки (вдалося прочитати рядок вхідного тексту)
  обробити рядок;
вивести підсумки обробки;
```

Продовжуючи деталізацію, додамо до класу прихований метод обробки рядка void loadOneLine(); . Для оптимізації виконання викликів допоміжних методів усю інформацію зберігатимемо не в локальних змінних методів, а в самому об'єкті. Поля

s, line, problems, loaded містять оброблюваний рядок вхідного файлу, його номер, кількість проігнорованих помилок та успішно завантажених непорожніх рядків, відповідно. Останні три не є обов'язковими, але дозволять повідомити діагностику помилок і підсумки обробки. Поле st об'єкта зберігає вказівник на контейнер, у який записуємо інформацію.

Додамо до класу Loader відповідні приховані поля:

```
private:
string s;            //поточний рядок
unsigned line=0;     // номер поточного рядка
unsigned problems=0;//кількість помилок
unsigned loaded=0;  //кількість успішно завантажених
                     // непорожніх рядків
Students *st=nullptr;
```

Зауваження. У реальних задачах до таких текстових файлів на початку додають ще кілька рядків заголовка (англ. *header*), а в кінці – кілька рядків із підсумковою інформацією щодо вмісту (завершальна частина документа, англ. *footer*). Якщо заголовок найчастіше містить стислий опис документа, наприклад "відомість ... від такої-то дати", то в кінці зазначають, скільки рядків з інформацією включає файл, яка загальна сума балів міститься в його рядках тощо. Якщо в процесі передавання текстового файлу інформацію буде спотворено (наприклад видалено якийсь рядок або змінено бали за якусь лабораторну), то за підсумковою інформацією це можна помітити. У нашій спрощеній задачі наявність таких елементів тексту не передбачається. Однак усетаки додамо до підсумків обробки інформацію щодо загальної суми успішно оброблених балів. Відповідну величину зберігаємо в прихованому полі totalScore:

```
private: unsigned totalScore=0;
```

Метод loadOneLine уточнимо пізніше, але нехай у випадку знаходження помилкової інформації він генерує виняток типу logic_error. Маємо таку реалізацію методу:

```
bool Loader::load(Students& st_,
                  const string &fname){
  ifstream f;
  st=&st_;
  cout<<"opening input file "<< fname<< endl;
```

```
    f.clear(); f.open(fname);
    if (!f) throw runtime_error(
            "Unable to open input file "+fname);
    line=0; problems=0; loaded=0; totalScore=0;
    while (getline(f,s)) {
      ++line;
      try {loadOneLine();}
      catch (logic_error &ex) { ++problems;
        cout<<s<<"\nLine "<<line<<": "<<
              ex.what()<<endl;}
    }
    if (!f.eof()) throw runtime_error(
            "Unable to read input file "+fname);
    f.close();
    if (problems) cout<<"there are "<< problems<<
      " problem lines while processing..."<< endl;
    cout<< line << " lines were read"<<endl;
    cout<< loaded <<
           " nonempty lines were processed"<<endl;
    cout<<"The total loaded score= "<<
          totalScore<<endl;
    return problems==0;
  }
```

Продовжуючи проектування згори донизу, уточнимо метод `loadOneLine` обробки рядка.

Узагалі, рядок або містить дані про студентів, або є заголовком чи завершальним у документі, чи "білим" (містить тільки білі символи). Від цього залежить обробка рядка.

На верхньому рівні алгоритм цього методу виглядатиме так:

`залежно від типу вмісту рядка викликати відповідний метод-обробник.`

Оскільки елементи інформації (ПІБ, назва лабораторної роботи, кількість балів) розміщуються в рядку через роздільник, то необхідно розв'язувати задачу їх виділення, розглядаючи їх як лексеми. Задача виділення лексем – типова задача лексичного аналізу. Зробимо це відповідальністю класу `Lexer`.

Клас для лексичного розбору рядка. Об'єкт класу `Lexer` у методі `load` отримує рядок, який треба розібрати, ініціює початок розбору й визначає, інформація якого ґатунку міститься в рядку, адже

структура початку рядків тексту в нашій задачі дозволяє визначити, належить рядок до заголовка, завершальної інформації або це звичайний рядок. Якщо дозволити записувати у вхідний файл білі рядки, то рядки поділяться на порожні (EMPTY) і рядки з інформацією (LINE). Заради цього у клас тип-перелік LineType. Тип-перелік інформативніший за логічний. З ним простіше модифікувати код для обробки рядків заголовка та рядків із завершальною інформацією.

Метод next виділяє з рядка черговий елемент і повертає ознаку успішності. Метод eof повертає ознаку того, чи розібрано рядок до кінця.

Звідси маємо такий інтерфейс класу Lexer:

```
class Lexer{
public:
  enum LineType {EMPTY, LINE};
  LineType load(const string &source);
  bool next(string &lex);//"" якщо рядок розібрано
  bool eof() const;
```

Міркуючи над реалізацією методів класу, вирішуємо зберігати в об'єкті: рядок source, розбір якого виконується; ознаку eof_ завершення розбору рядка; позицію pos першого нерозібраного символу. Поки не будемо параметризувати клас роздільником ';', натомість збережемо його як статичне поле delim класу. Звідси маємо таку сукупність полів, що завершує означення класу:

```
protected:
  string source;
  bool eof_=true;
  basic_string<char>::size_type pos=0;
  static const char delim=';';
};
```

Тут basic_string<char>::size_type – це тип, яким індексуються позиції символів рядків типу string.

Наведемо реалізацію методів цього класу.

```
Lexer::LineType Lexer::load(const string &source_) {
  if (!trim(source_).size()) {
    source=""; eof_=true;
    pos=string::npos; return EMPTY; }
  else { source=source_; eof_=false;
         pos=0; return LINE; }
}
```

Наведений метод використовує функцію trim, яка повертає рядок з вилученими білими символами на початку та в кінці. Наприклад, за рядка string(" qwerty ") вона поверне новий рядок із значенням string("qwerty"). Ця функція не є стандартною й означена в окремо розробленій бібліотеці (повну версію коду див. на сайті книги). За білого рядка функція поверне рядок довжиною 0.

```
bool Lexer::eof() const {return eof_;}
```

Якщо розбір рядка ще не завершено, то метод next намагається знайти наступний від поточної позиції роздільник. За лексему береться частина рядка від поточної позиції до знайденого роздільника (не включаючи його), якщо він є, інакше – до кінця рядка. В останньому випадку виставляється ознака завершення розбору рядка. Також переноситься поточна позиція:

```
bool Lexer::next(string &lex){
  lex="";
  if (eof()) return false;
  auto pos1=source.find(delim,pos);
  lex=source.substr(pos,pos1-pos);
  if (pos1<string::npos)
    ++pos1;
  else eof_=true;
  pos=pos1;
  return true;
}
```

Завантаження рядка з інформацією в контейнер. Додамо до прихованих полів класу Loader об'єкт класу Lexer, який буде здійснювати розбір, і поля для збереження частин рядка, що містять результати розбору, а саме: ПІБ, назву лабораторної роботи й кількість балів:

```
private:
  Lexer lex;
  string sname, slabCode, sscore;
```

Метод обробки рядка ініціює лексичний розбір поточного рядка й далі за потреби передає керування відповідному обробнику.

```
void Loader::loadOneLine(){
  switch(lex.load(s)){
    case Lexer::EMPTY: return;
    case Lexer::LINE: loadLine(); return;
    default: ++problems;
```

```
      throw logic_error("Unknown answer from Lexer::\
  load. Revise code for Loader::loadOneLine\n");
    }
  }
```

Тип-перелік LineType є вкладеним типом класу Lexer. Тому за межами класу Lexer константи переліку мають іменуватися кваліфікованим іменем: Lexer::LINE тощо.

Для розбору та завантаження рядка з інформацією викликається метод loadLine. Якби у вхідному тексті були присутні рядки інших типів, то для кожного з них мав би бути власний метод-обробник.

Додамо до класу Loader прихований метод void loadLine(); . Алгоритм роботи loadLine такий:

1) розібрати вхідний рядок на частини;
2) за необхідністю перетворити рядкові значення на відповідні числові;
3) записати конвертовану інформацію в контейнер;
4) обновити підсумкову інформацію.

Перший крок алгоритму буде виконувати прихований метод void parseLine();. Він відповідає за розділення рядка на частини згідно з роздільниками й має переконатися, що частин правильна кількість (у нашому прикладі три).

```
void Loader::parseLine(){
  if (!lex.next(sname)) throw logic_error(
    "Too few data in line: student name is absent");
  if (!lex.next(slabCode)) throw logic_error(
    "Too few data in line: lab code is absent");
  if (!lex.next(sscore)) throw logic_error(
    "Too few data in line: score is absent");
  if (!lex.eof()) throw logic_error(
    "Too much data in line");
}
```

Запишемо означення методу loadLine (рядки коду відповідають крокам алгоритму):

```
void Loader::loadLine(){
  parseLine();
  score=convert_u(sscore);
  st->addLabAttempt(sname,slabCode,score);
  totalScore+=score; ++loaded;
}
```

Тут функція convert_u (з окремо розробленої бібліотеки) виконує перетворення рядка на беззнакове ціле. Для запису інформації в контейнер використовується метод контейнера.

Зауваження. Семантичні обмеження на довжину прізвища, назву лабораторної роботи тощо мають визначатися не завантажувачем, а типом елементів контейнера. Тому жодних перевірок у цьому методі не виконується. Надалі побачимо, що вони будуть здійснені під час конструювання об'єктів, що зберігаються в контейнері. Водночас перевірка коректності рядка з балами виконується саме тут. Неправильно віддавати методу контейнера рядкове зображення числових даних, оскільки це буде означати залежність контейнера від зображення зовнішніх даних.

Перетворення рядка на беззнакове ціле. Для розв'язання поставленої задачі існують стандартні бібліотечні засоби. Однак тут є нюанс: при виклику stoul(" -1 ") стандартної бібліотечної функції, що перетворює рядок типу string на беззнакове довге ціле, буде повернуто значення ULONG_MAX (яке кодується тією самою послідовністю бітів, що й довге ціле -1) і жодної ознаки помилки. Більш того, за стандартом така поведінка звичайна для зазначеної функції.

Наша функція convert_u виконує перевірку на від'ємність. Також вона перевіряє, чи немає в рядку іншої небілої інформації, крім перетвореного зображення числа (стандартна функція має засоби це побачити, але для неї така ситуація не є помилковою). Крім того, вона буде перехоплювати винятки, що їх кидає стандартна stoul, і замість них кидати винятки, що містять більше інформації стосовно проблеми.

```
unsigned convert_u(const string& s){
  if (s.find("-")!=string::npos)
    throw out_of_range
      ("unsigned ["+s+"] could not be negative");
  unsigned long ul; size_t ind;
  try { ul=stoul(s,&ind); }
  catch(out_of_range &ex) {
    throw out_of_range( string("unsigned [" + s +
      "] is out of range (") + ex.what() + ")");
  }
  catch(invalid_argument &ex) {
```

```
    throw invalid_argument(string("unsigned [" + s +
      "] is invalid (") + ex.what()+")");
  }
  if (s.find_first_not_of(WHITES_C,ind) !=
      string::npos)
    throw invalid_argument(
       "unsigned ["+s+"] is invalid");
  if (ul>UINT_MAX)
    throw out_of_range(
      "unsigned [" + s + "] is out of range");
  return ul;
}
```

Другий аргумент виклику `stoul` – адреса змінної `ind`. Під час виклику в цю змінну буде записано індекс позиції рядка, на якій зупинилося перетворення. Якщо, починаючи з неї, рядок містить тільки білі символи, перелік яких записано в рядку

```
const string WHITES_C="\011\012\013\014\015\040";,
```

то все гаразд, інакше рядок містить якусь нечислову інформацію і тому генерується виняток.

Зауваження щодо тестування класу Loader. На даному етапі для тестування класу `Loader` достатньо мати такий клас `Students`:

```
class Students : public SList<Student>{
public:
  void addLabAttempt(const string &name,
    const string &labCode, unsigned score);
};
```

Замість реального завантаження інформації в контейнер можна написати "заглушку" (*stub*), яка дасть можливість використати клас `Students` у тестувальній програмі для класу `Loader`:

```
void Students::addLabAttempt(const string &name,
     const string &labCode, unsigned score){
  cout<< "Students::addLabAttempt \n";
  cout<< " now is a stub\n";
  cout<< "arguments: name=[" +name+ "], labCode=[" +
    labCode + "], score=" + to_string(score) << endl;
}
```

5.4. Обгортки для контейнерів

Повернемось до проектування контейнера, що зберігає інформацію з файлу.

Коли необхідно використовувати структури даних для різних типів елементів, що в них зберігаються, шаблони дають потужний спосіб уникнути дублювання коду. Для конкретніших задач виникає потреба в більш специфічних операціях над контейнером. Тоді можна побудувати **обгортку** (англійці використовують терміни *wrapper*, *decorator*), яка додає до загального інтерфейсу специфічну функціональність.

Припустимо, що є клас `Student`, який зберігає інформацію про конкретного студента, а також шаблонний клас `Container`, який реалізує певний контейнер. Раніше вже зазначалося, що клас `Students`, який відповідає за збереження всієї інформації та, по суті, також є контейнером, повинен мати відкритий метод `addLabAttempt`. Міркуючи про його реалізацію та реалізацію методів доступу до інформації, можна піти різними шляхами. Можна "з нуля" записати реалізацію класу `Students`, але такий підхід вкрай неефективний. Натомість кращим рішенням буде використати наявний клас `Container`. Оскільки його треба підлаштувати під конкретні потреби, зокрема додати метод `addLabAttempt`, то клас `Students` має стати його обгорткою.

Існують два основні способи реалізувати обгортку: використати *композицію* класів або *успадкування*. Розглянемо їх обидва.

У випадку *композиції* об'єкт класу-обгортки містить об'єкт-контейнер як поле. Для того, щоб ззовні класу мати доступ до операцій над контейнером, необхідно додати відповідні відкриті методи. У наведеному прикладі використано саме композицію:

```
class Students0{
public:
  SList<Student>::Iterator begin() const;
  SList<Student>::Iterator end() const;
  unsigned size() const;
  void addLabAttempt(const string &name,
         const string &labCode, unsigned score);
private:
  SList<Student> students;
};
```

Однозначна перевага такого підходу в тім, що клас Student може бути прихованим вкладеним класом класу-обгортки Students0, тобто внутрішній устрій контейнера-обгортки за межами його методів невідомий.

У випадку *успадкування* об'єкт класу-обгортки містить об'єкт-контейнер як об'єкт базового класу. За відкритого успадкування всі відкриті методи базового класу увійдуть в інтерфейс похідного:

```
class Student;
class Students:public SList<Student>{
public:
  void addLabAttempt(const string &name,
    const string &labCode, unsigned score);
};
```

Більш того, об'єкти класу Students можна використовувати всюди, де можна використовувати об'єкти класу SList<Student>. Тому для виведення об'єктів класу Students можна використати означений для SList шаблонний operator <<. Клас Student має бути оголошений за межами Students.

Реалізація методу addLabAttempt намагається додати нового студента до контейнера (або просто знаходить, якщо він існує), та *делегує* подальшу роботу зі збереження інформації про виконану лабораторну роботу відповідному методу класу Student:

```
void Students::addLabAttempt(const string &name,
        const string &labCode, unsigned score){
  auto first=insert(Student(name)).first;
  (const_cast<Student&>(*first)).addLabAttempt(
                labCode, score);
}
```

Оскільки результатом розіменування ітератора класу SList є значення типу const Student& – посилання на незмінюване значення, то треба явно перетворити його тип на Student&. Ця ситуація виникла не тому, що шаблонний клас було спроектовано неправильно (як раз навпаки, клас SList повністю підтримує цілісність даних), а тому, що для цієї задачі краще було б використати іншу структуру даних. Якщо не йти шляхом кардинальних змін у напрямі ефективних структур даних, то, наприклад, можна було б використати список, що не є відсортованим, а для виведення елементів

у певному порядку додати метод, що його сортує. Варіант зі зміною оголошення оператора розіменування ітератора все-таки краще відкинути: метод, що здатний викликати порушення цілісності даних за необережного або неуважного застосування, може спричинити великі проблеми з налагоджуванням коду.

У житті доволі часто виникає *необхідність пристосовувати наявні бібліотеки до поточних задач* (наприклад тому, що пристосувати наявне виявляється дешевше, ніж розробити нову бібліотеку, або тому, що використання іншої бібліотеки неможливе). Тому жодної трагедії тут немає: будемо використовувати те, що є, до того ж мова C++ дає для цього синтаксичні можливості. У наступному розділі побачимо, що для ефективніших стандартних бібліотечних контейнерів розв'язувана задача теж має певні "підводні камінці".

Для зняття незмінюваності слід використати оператор `const_cast`. Дана операція в загальному випадку є небезпечною. У нашому прикладі вона може призвести до порушення цілісності даних, зокрема властивості відсортованості, оскільки після перетворення типу зміни елемента списку вже можливі. Трошки забігаючи наперед, скажемо, що студенти будуть упорядковуватися за ПІБ, що буде оголошено незмінним, а метод `addLabAttempt` класу `Student` не буде намагатися змінити ПІБ, тому з упорядкуванням списку все буде гаразд.

Розглянемо детальніше будову класу `Student`. Крім методу `addLabAttempt` та конструктора, у його відкриту частину ввійдуть методи `name`, `nlab` та `score`, що повертають ПІБ студента, кількість зданих лабораторних робіт і набрану за них суму балів, відповідно. Також додамо оператор перетворення до типу `string`, який генерує рядок з інформацією, оператори `==` та `<` порівняння студентів (вони необхідні для використання типу `Student` як типу елементів контейнера; в обох випадках лексикографічно порівнюються ПІБ).

```
class Student{
public:
  explicit Student(const string& name);
    //throw invalid_argument if not valid name
  string name() const;
  unsigned nlab() const;
  unsigned score() const;
  operator string() const;
```

```
  bool operator==(const Student&) const;
  bool operator<(const Student&) const;
  void addLabAttempt(const string &labCode,
                     unsigned score);
```

Перевірку правильності рядка, що має задавати ПІБ студента, виконаємо в конструкторі. Однак призначення конструктора – конструювання об'єкта. Якщо він ще безпосередньо виконуватиме перевірку, то це буде його додатковим призначенням. Однією з тенденцій сучасного проектування є **функціональне проектування**, згідно з яким кожна функція або метод повинні мати *єдине призначення*. Тому перевірку коректності імені виділимо в окремий прихований метод `isValidName`, який викликається конструктором. Таке відокремлення логіки перевірки в майбутніх релізах дозволить точно локалізувати код, що виконує перевірку, і, за необхідності змін, локалізувати ці зміни. Також додамо поля, що зберігають ПІБ, набрану за лабораторні роботи кількість балів і контейнер з інформацією про лабораторні роботи. Лабораторні роботи промоделюємо об'єктами класу `Lab`. Оскільки це прихований вкладений клас, то його можна оголосити з ключовим словом `struct`, щоб у методах класу `Student` мати повний доступ до всіх членів класу `Lab`:

```
private:
  void isValidName(const string&);
  const string name_;
  unsigned score_=0;
  struct Lab;
  SList<Lab> labs;
};
```

Зауваження. У цій ситуації можна було б зробити клас `Student` похідним від класу `SList<Lab>`. Оскільки зміни інформації в контейнері `SList<Lab>` мають викликати зміни поля `score_`, то успадкування в цьому випадку мало б бути прихованим або захищеним: за межами методів класу `Student` зміни об'єкта його базового класу мають бути неможливими. Також клас `Lab` треба було б оголосити за межами класу `Student`. Звісно, це не проблема, але зазначений внутрішній клас доволі специфічний, його устрій краще приховати. Отже, для побудови класу `Student` обрано варіант композиції.

Для класу Student перевантажимо оператор вставки в потік:

```
ostream& operator<<(stream&, const Student&);
```

Наведемо реалізацію методів класу. Конструктор відкидає початкові й заключні білі символи рядка, що має містити ПІБ, та ініціалізує отриманим значенням відповідне поле. Далі виконується перевірка коректності й за необхідності кидається виняток:

```
Student::Student(const string& name):
            name_(trim(name)) { isValidName(name_); }
void Student::isValidName(const string&s) {
  if (s.size()<3 || s.size()>80)
    throw invalid_argument("The name [" + s +
                "] is invalid for student.\n");
}
```

Зауваження. Оскільки поле name_ оголошене як незмінюване, то надати йому значення можна *тільки під час ініціалізації*.

Такі два методи повертають значення відповідних полів:

```
string Student::name() const { return name_; }
unsigned Student::score() const { return score_; }
```

Кількість лабораторних робіт визначається як кількість елементів контейнера з лабораторними роботами:

```
unsigned Student::nlab() const
  { return labs.size(); }
```

Результатом перетворення до типу string є рядок, що містить ПІБ та агреговану інформацію про студента (кількість лабораторних, суму набраних балів), до якої під час обходу контейнера з лабораторними роботами додається інформація по кожній з них:

```
Student::operator string() const {
  string res=name_ + " has " + to_string(score()) +
      " points and pass " + to_string(nlab()) +
      " labs:";
  auto from=labs.begin(), to=labs.end();
  for(; from!=to; ++from) res+="\n"+string(*from);
  return res;
}
```

Порівняння студентів виконується як порівняння їхніх ПІБ:

```
bool Student::operator ==(const Student &other)const
  { return name_ == other.name_; }
bool Student::operator <(const Student &other)const
  { return name_ < other.name_; }
```

Оператор вставки в потік виводить у потік об'єкт, перетворений на тип string:

```
ostream& operator <<(ostream&f, const Student&s){
  f<<(string)s;
  return f;
}
```

Метод addLabAttempt додає лабораторну роботу, якщо її ще немає в переліку, інакше він порівнює поточний бал за лабораторну роботу та бал спроби, що додається, і залишає максимальний. В усіх випадках він оновлює суму балів студента. Реалізацію методу наведемо після означення класу Lab.

Поточна версія класу Lab не містить детальної інформації щодо спроб. Натомість об'єкти класу будуть зберігати назву лабораторної роботи, кількість спроб її здати й результат найкращої спроби в полях labCode_, n та score, відповідно. За коректної назви конструктор класу створює об'єкт, що відповідає лабораторній роботі, яка ще не здавалася; інакше він кидає виняток типу invalid_argument. Перевірку коректності назви виділено в метод isValidLabCode. Методи nAttempts та bestScore повертають значення полів n та score, відповідно. Для поточної версії можна було б обійтися без них, але у випадку модифікації класу, щоб він зберігав історію спроб, їх наявність дозволить уникнути змін решти коду, який використовує значення цих полів. Оператор перетворення до типу string повертає рядок з інформацією про лабораторну роботу. Оператори порівняння порівнюють назви лабораторних робіт. Метод addAttempt відповідає за додавання спроби й повертає величину, на яку збільшилася остаточна кількість балів за лабораторну:

```
struct Student::Lab{
  explicit Lab(const string& labCode);
      //throw(invalid_argument)
```

```
  void isValidLabCode(const string&);
      //throw(invalid_argument)
  unsigned nAttempts() const;
  unsigned bestScore() const;
  operator string()const;
  bool operator ==(const Lab&) const;
  bool operator <(const Lab&) const;
  unsigned addAttempt(unsigned points);
  const string labCode_;
  unsigned n=0;      //кількість спроб
  unsigned score=0; //результат найкращої спроби
};
```

Наведемо означення тільки деяких методів цього класу.

```
Student::Lab::Lab(const string& labCode):
                labCode_(trim(labCode))
  { isValidLabCode(labCode_); }
void Student::Lab::isValidLabCode(const string& s){
  static const unsigned maxLabCodeLen=8;
  if (s.empty())
    throw invalid_argument(
      "The white name is invalid for lab code.\n");
  if (s.size()>maxLabCodeLen)
    throw invalid_argument(
      "Too long lab code. Should be no more than " +
       to_string(maxLabCodeLen) + " symbols.\n");
}
unsigned Student::Lab::addAttempt(unsigned points){
  unsigned delta=0;
  if (score<points)
    { delta=points-score; score=points; }
  ++n;
  return delta;
}
```

Означення решти методів тривіальні й залишаються як вправи. Повна версія коду доступна на сайті книги.

Тепер запишемо означення методу addLabAttempt класу Student:

```
void Student::addLabAttempt(const string &labCode,
                                    unsigned score){
```

```
    auto first=labs.insert(Lab(labCode)).first;
    auto delta=(const_cast<Student::Lab&>(*first)).
                addAttempt(score);
    score_+=delta;
  }
```

Тут спочатку додаємо лабораторну роботу із заданою назвою. Першою компонентою результату є ітератор, установлений або на додану, або на існуючу лабораторну. Далі до цієї роботи додаємо спробу. У результаті може змінитися остаточний бал за неї, величина зміни повертається з методу та на цю величину оновлюється загальна кількість балів студента. Аналогічно відповідному методу класу Students тут необхідно явно знімати незмінюваність посилання на елемент, яке є результатом розіменування ітератора.

Приклад клієнтського коду та його роботи. Реалізований для списку шаблонний оператор << виводить елементи через пробіл. Для списку студентів це не дуже підходить, тому перевантажимо << для класу Students явно. Скористаємося цим оператором у функції testLoader, яка зчитує дані з файлу, додає їх у об'єкт класу Students і виводить результати додавання:

```
  ostream& operator <<(ostream& f, const Students&c){
    auto from=c.begin(), to=c.end();
    if (from!=to)
      { f << (*from) << endl; ++from; }
    for(; from!=to; ++from)
      f << endl << (*from) << endl;
    return f;
  }
  void testLoader(const string &fname){
    Loader l;
    Students c;
    try{
      cout<<"load " << fname << " to container\n";
      if (l.load(c,fname))
        cout << " OK;\n";
      else
        cout << " with errors\n";
      cout << c << endl;
      cout << endl;
```

```
  }
  catch (runtime_error &ex)
    { cout << "runtime: " << ex.what() << endl; }
  catch (logic_error &ex)
    { cout << "logic: " << ex.what() << endl; }
  catch (exception &ex)
    { cout << "other: " << ex.what() << endl; }
}
```

Розглянемо обробку файлу text.txt з таким змістом:

```
Sidorov I I; Lab4; 10
Ivanchuk I I; Lab1; 12
Ivanchuk I I; Lab1; 1
Petrenko P P; Array;14
Ivanchuk I I; Lab1; 13
Ivanchuk I I; Lab4; 8
Ivanchuk I I; Lab3; 2
```

За виконання виклику testLoader("text.txt") буде виведено діагностику завантаження та вміст контейнера:

```
load text.txt to container
opening input file text.txt
7 lines were read
7 nonempty lines were processed
The total loaded score= 60
 OK;
Ivanchuk I I has 23 points and pass 3 labs:
[Lab1]=13 in 3 attempts
[Lab3]=2 in 1 attempts
[Lab4]=8 in 1 attempts

Petrenko P P has 14 points and pass 1 labs:
[Array]=14 in 1 attempts

Sidorov I I has 10 points and pass 1 labs:
[Lab4]=10 in 1 attempts
```

Контрольні запитання

1. Чому завантаження даних із зовнішнього носія в контейнер не слід проектувати як метод контейнера?

2. З якою метою в текстові файли з інформацією додають підсумкову інформацію?

3. Які помилки знаходить і діагностує стандартна бібліотечна функція `stoul`? Чи вважає вона знакове зображення помилкою?

4. У чому відмінність підходів у реалізації класу-обгортки за допомогою композиції та за допомогою успадкування? Які особливості має кожен із цих підходів?

5. Що таке функціональне проектування?

6. Для чого використовують оператор `const_cast`? У чому небезпечність цієї операції?

Вправи

1. Змінити клас `Lexer` так, щоб його можна було застосувати для розбору рядків, які використовують інший символ-роздільник між частинами.

2. Нехай текст, крім рядків з інформацією, містить на початку один рядок із заголовком, а в кінці – один рядок з підсумковою інформацією. Рядок із заголовком починається із символів `HEADER:`, підсумковий рядок – із символів `FOOTER:`, решта рядків містять інформацію про виконання лабораторних робіт (див. задачу про лабораторні роботи), до якої на початку додано порядковий номер рядка в документі та роздільник. Змінити клас `Lexer` так, щоб при завантаженні в нього рядка він визначав тип рядка, що обробляється.

3. Попередня задача, але текстовий файл може містити в довільній кількості й довільних місцях "білі рядки" (порожні рядки та рядки з білих символів), які мають ігноруватися при обробці.

4. Змінити клас `Loader` так, щоб він міг успішно завантажувати в контейнер текстові файли з попередньої задачі. При цьому має виконуватися перевірка, чи рядки з інформацією про виконання лабораторних робіт пронумеровані послідовними числами, починаючи з 1.

5. Попередня задача, але за умови, що в підсумковому рядку після FOOTER: через роздільник указується загальна кількість рядків з інформацією та загальна сума балів по всіх рядках. Успішне завантаження текстового файлу має передбачати, що вміст рядків з інформацією відповідає підсумковому рядку файлу.

6. Змінити структуру даних для задачі про лабораторні роботи так, щоб наприкінці для кожної виконаної студентом лабораторної роботи виводилися не тільки бал найкращої спроби та загальна кількість спроб, але ще й інформація по кожній спробі.

7. До попередньої задачі внесемо такі зміни: крім кількості балів для кожної спроби через роздільник указується кілька приміток (перелік зауважень тощо). Модифікувати структуру даних для збереження інформації та клас Loader відповідно до змін умови задачі.

6. Використання стандартних асоціативних контейнерів

Власний шаблон контейнера `SList` було створено переважно з метою показати внутрішню будову контейнера та його ітератора, а також продемонструвати можливості та синтаксичні нюанси мови C++.

Насправді мова C++ надає розвинену бібліотеку контейнерів: контейнери послідовностей (англ. *sequence containers*) та асоціативні контейнери (англ. *associative containers*). Шаблонні контейнери послідовностей `array`, `deque`, `list`, `vector`, `forward_list` та їхні адаптери `queue`, `priority_queue`, `stack` призначені для зберігання послідовностей значень. Хоча деякі з них надають прямий доступ до елементів, але все-таки вони передбачають, що під час обробки переглядається вся послідовність; операція пошуку елемента в послідовності їм не притаманна.

Асоціативні контейнери передбачають, що під час обробки не обов'язково переглядати всі елементи контейнера; крім послідовного доступу до елементів вони додатково реалізують операцію пошуку. Більш того, вони забезпечують реалізацію додавання, вилучення й пошуку елемента за логарифмічний час. У випадку асоціативних контейнерів суттєвим є поняття **ключа**. Під ключем розуміють ключову частину інформації про об'єкт, тобто таку частину даних, що дозволяє однозначно ідентифікувати ціле. Наприклад, паспорт однозначно визначає конкретну людину з усіма її характеристиками та властивостями, тому паспортні дані можна розглядати як ключ.

Зараз нашою метою є *з мінімальними зусиллями використати стандартні бібліотечні контейнери* для задачі про лабораторні роботи. Оскільки ця задача потребує пошуку елемента в послідовності, то від контейнерів послідовностей відмовимось. Спробуємо використати асоціативні контейнери `set` та `map`. Решту асоціативних контейнерів і невпорядковані асоціативні контейнери залишимо поза увагою.

6.1. Реалізація класу Students за допомогою стандартного бібліотечного контейнера set

Контейнер `set` зберігає тільки ключі, які мають не повторюватися. У нашому випадку ключем для даних об'єкта `Student` є його прізвище, ім'я та по батькові, що зберігається в полі `name_`; ключем для даних об'єкта `Lab` є кодова назва лабораторної роботи, що зберіга-

ється в полі labCode_. Хоча насправді ключем є не все значення об'єкта, але це не є проблемою: оператори < та == для класів Student та Lab означено так, що вони порівнюють саме ключі. Тому інстанціюємо шаблонний контейнер set для класів Student та Lab.

Далі наведено тільки специфічні зміни раніше розроблених класів:

```
class Students: public set<Student> {
  //решта без змін
};
class Student {
  // решта без змін
  set<Lab> labs;
};
```

Насправді клас SList проектувався так, щоб його інтерфейс відповідав частині інтерфейсу стандартних бібліотечних асоціативних контейнерів (стандартні контейнери "вміють" дещо більше, зокрема вилучати дані). Тому реалізації методів та оператора вставки в потік жодних змін не зазнають.

Зазначимо, що, аналогічно нашому класу SList, клас set не передбачає зміну елемента (тому необхідність перетворення незмінюваного типу на змінюваний під час додавання інформації залишається). Клас set розглядає елемент як ключ. За зміни ключа порушуються внутрішні властивості об'єкта контейнера, які гарантують його коректну роботу. Необхідні нам перетворення знайденого в контейнері значення не впливають на результат виконання операцій порівняння елементів контейнера, тому виконувані зміни елементів контейнера є допустимими.

Зауваження. У загальному випадку для зміни елемента контейнера set цей елемент треба вилучити з контейнера та замість нього додати новий – його змінене значення. У випадку, коли елементами контейнера є доволі складні структури даних, ця операція може призвести до великої кількості зайвих копіювань інформації й написання значної кількості рядків коду для реалізації зазначеного підходу.

6.2. Реалізація класу Students за допомогою стандартного бібліотечного контейнера map

Контейнер map зберігає пари (ключ, значення), причому ключі мають бути унікальними. Він також дає можливість пошуку за ключем і навіть зміни значення, що відповідає ключу.

З одного погляду, для нашої задачі це непогано: алгоритм додавання інформації в контейнер передбачає пошук необхідного елемента й додавання до нього інформації (тобто його зміну). Тому необхідність перетворення незмінюваного типу на змінюваний зникає й використовується менше потенційно небезпечних операцій.

З іншого погляду, тоді треба ключ (прізвище, ім'я, по батькові) зберігати окремо від об'єкта, який він ідентифікує. А це вже порушує принципи об'єктно-орієнтованого програмування, зокрема принцип інкапсуляції, оскільки дані про студента не будуть зберігатися як цілісний об'єкт.

Оскільки ми й надалі збираємося працювати з об'єктом класу `Student` як з цілісною одиницею, що містить вичерпну інформацію, то залишимо ключ (прізвище, ім'я та по батькові) частиною даних і будемо зберігати пари (ПІБ, студент), а також (кодова назва лабораторної, лабораторна). Це спричинить невеликі, але зайві витрати пам'яті (ключ буде зберігатися як у значенні, так і окремо). Крім того, маючи можливість змінити значення, можна змінити також його ключову частину й тим самим порушити цілісність даних. Однак поле із прізвищем студента є незмінюваним, а дані лабораторної роботи приховані, тому порушення цілісності даних унаслідок зовнішнього втручання неможливе.

Проте покажемо, як можна було б вийти із ситуації, щоб гарантувати відповідність ключів до значень. Для цього достатньо використати захищене успадкування:

```
class Students:protected map<string,Student>{
```

За такого способу успадкування об'єкт базового класу є захищеним і доступ до нього можливий тільки з методів класу `Students`. Отже, клієнти класу `Students` не зможуть використати методи базового класу `map<string,Student>`, щоб змінити значення, яке відповідає ключу. Однак тоді клієнти класу не зможуть використати ітератори, що надають доступ до елементів контейнера. Тому додамо відкриті методи, які повертають ітератори, установлені на початок і за кінець послідовності значень:

```
public:
  void addLabAttempt(const string &name,
         const string &labCode, unsigned score);
  auto begin()const noexcept;
  auto end() const noexcept;
```

```
};
class Student{
  // решта без змін
  map<string, Lab> labs;
};
```

Зауваження. Якщо як тип результату методу вказано auto, то компілятор самостійно визначить тип за типом виразу, указаного в інструкції return в означенні відповідного методу.

Зауваження. За аналогією з відповідними методами класу map методи begin та end оголошено з ключовим словом noexcept, яке зазначає, що їх виклик не може завершитися в результаті кидання винятку. Ця специфікація винятків не є обов'язковою.

Додані методи делегують (тобто передають) свої обов'язки відповідним методам базового класу:

```
auto Students::begin() const noexcept {
  return map<string, Student>::begin();}
auto Students::end() const noexcept {
  return map<string, Student>::end();}
```

Оскільки в контейнері map зберігаються не тільки значення, а й пари (ключ, значення), то необхідно внести зміни в реалізації деяких методів і функцій: відповідні дії мають виконуватися не над парою, яка є елементом контейнера, а над другою компонентою пари. Крім того, операція вставки елемента в контейнер також додає не окреме значення, а пару. Далі наведено зміни реалізації методів:

```
void Students::addLabAttempt(const string &name,
        const string &labCode, unsigned score) {
  auto first=insert(
  make_pair(name,Student(name))).first;
  ((*first).second).addLabAttempt(labCode, score);
}
ostream& operator <<(ostream& f, const Students&c) {
  auto from=c.begin(), to=c.end();
  if (from!=to)
    {f << (*from).second << endl; ++from;}
  for(; from!=to; ++from)
    f << endl << (*from).second << endl;
  return f;
}
```

```
Student::operator string() const {
  string res=name_ + " has " + to_string(score()) +
  " points and pass " + to_string(nlab()) + " labs:";
  auto from=labs.begin(), to=labs.end();
  for(; from!=to; ++from)
    res+= "\n" + string((*from).second);
  return res;
}
void Student::addLabAttempt(
        const string &labCode, unsigned score) {
  auto first= labs.insert(
        make_pair(labCode,Lab(labCode))).first;
  auto delta=((*first).second).addAttempt(score);
  score_+=delta;
}
```

Задачі для самостійного розв'язання

У наведених нижче задачах вхідний текстовий файл починається з рядка-заголовка, за яким ідуть рядки з інформацією (записи) і в самому кінці – підсумковий рядок. У файлі в довільній кількості можуть зустрічатися білі рядки, які мають ігноруватися при обробці.

Рядок із заголовком починається із символів HEADER:, підсумковий рядок – із символів FOOTER:, решта рядків (записи) містять інформацію, до якої на початку додано порядковий номер рядка (нумерація з 1, білі рядки та заголовок не нумеруються) у документі та роздільник. Окремі частини рядків з інформацією розділено роздільником (;). Роздільник усередині інформації не зустрічається. Вміст рядків з інформацією для задач наведено в дужках.

Обробка файлу має передбачати контроль відповідності заголовка та підсумкового рядка до його вмісту, а також контроль відповідності до умови вмісту рядків файлу. За виникнення довільних розбіжностей завантаження файлу слід вважати неуспішним і виводити відповідне повідомлення.

1. Текстовий файл містить результати складання студентами однієї групи іспитів зимової сесії (номер залікової книжки, прізвище, ім'я, по батькові студента, дисципліна, оцінка в балах за 100-бальною системою, оцінка за державною шкалою). Знайти 5 найкращих студентів, а також що й на скільки вони склали. Якщо 5-те місце посідають кілька студентів одночасно, то вивести інформацію по всіх них.

У заголовку вказується кількість студентів, у підсумковому рядку через роздільник – кількість записів у файлі й сумарний бал.

2. Текстовий файл містить перелік відвантажених накладних: 13-символьний номер накладної, назва товару, кількість (дійсне число, яке має від одного до трьох знаків після коми), ціна (дійсне число, два знаки після коми), вартість (дійсне число, два знаки після коми). Знайти найпопулярніші товари (ті, що відвантажувалися в найбільшій кількості накладних), визначити, у скількох чеках, скільки й на яку суму їх було відвантажено.

У заголовку вказується загальна кількість чеків, у підсумковому рядку через роздільник – кількість записів у файлі й сума всіх вартостей.

3. Текстовий файл містить результати футбольних матчів: країни-учасники, рахунок, перелік бомбардирів (номер, прізвище, ім'я та хвилини зустрічі, на яких вони забивали голи). Знайти найкращих бомбардирів (країна, прізвище, ім'я та матчі, у яких вони забивали, а також рахунок цих матчів і кількість забитих ними голів).

У заголовку вказується кількість проведених зустрічей, у підсумковому рядку – кількість забитих м'ячів.

4. Текстовий файл містить статистику спринтерських біатлонних гонок кубку світу з двома вогневими рубежами: порядковий номер етапу, стартовий номер учасника, зайняте місце, прізвище, ім'я, країна, загальний час проходження дистанції (у секундах), час підходу й виходу з кожного вогневого рубежу (відносний, у секундах від початку гонки), загальна кількість промахів, загальний час подолання штрафних кругів. Для тих, хто не завершив гонку, указується місце 0 і загальний час проходження дистанції -1. Знайти тих, хто протягом сезону стріляв найточніше (незавершені гонки до уваги не брати). Вивести по них дані по всіх гонках (етап, стартовий номер, зайняте місце, загальний час проходження дистанції, кількість промахів) і вказати найкращий час стрільби та гонки з найкращою влучністю.

У заголовку вказується кількість етапів, у підсумковому рядку – кількість записів і загальна кількість промахів.

5. Текстовий файл містить результати оцінювання змагань суддівським журі. Кожен з 10 членів анонімного журі за кожним пунктом оцінювання виставляє бали (ціле від 0 до 10); таким чином отримується сумарна оцінка, яку суддя виставляє учаснику. Далі відкидаються найгірша та найкраща сумарні оцінки й береться середнє від тих, що залишилися. Це є остаточним результатом учасника. Вивести учасників та їхні остаточні оцінки за незростанням оцінок. Рядки файлу містять інформацію: стартовий номер, прізвище, ім'я учасника, пункт оцінювання (словами), номер судді, кількість балів.

У заголовку вказується перелік пунктів оцінювання (через роздільник), у підсумковому рядку – загальна кількість учасників і сума всіх виставлених суддями балів.

6. Текстовий файл містить розклад на семестр: день тижня, пара, курс, група, аудиторія, дисципліна, вид занять (лекція, практичне, семінар, лабораторне). Конфліктом аудиторій назвемо ситуацію, коли одночасно в одній аудиторії мають проводитися заняття більше ніж однієї групи, причому не лекції для груп одного курсу з однакової дисципліни.

Перевірити, чи існує в розкладі конфлікт аудиторій. Якщо існує, то повідомити про всі такі конфлікти (день, пара, які дисципліни й у кого там мають відбутися).

У заголовку вказується, що цей документ містить розклад, у підсумковому рядку – загальна кількість записів і кількість різних навчальних груп.

7. Текстовий файл містить розклад на добу приміських електропоїздів (номер поїзда, назва зупинки, час прибуття, час відправлення) за певним напрямком. За заданими двома зупинками знайти всі електропоїзди, що курсують від першої зупинки до другої. Вважати, що в правильному розкладі два електропоїзди не відправляються й не прибувають на одну зупинку одночасно.

Вивести поїзди в порядку відправлення з першої заданої зупинки. Для кожного електропоїзда виводити: номер, час відправлення з першої зупинки, час прибуття на другу зупинку, час у дорозі, чи є перша зупинка початком руху електропоїзда.

У заголовку вказується, що файл містить розклад електропоїздів певного напрямку та повний перелік через роздільник зупинок на цьому напрямку (відсортований за маршрутом руху). У підсумковому рядку вказується кількість різних електропоїздів, присутніх у розкладі, і кількість записів.

Післямова

У процесі розв'язання задачі про лабораторні роботи можна було бачити, що кожен наведений спосіб побудови загальної структури даних для збереження інформації зі вхідного текстового файлу мав певні недоліки.

Наприклад, можна було б побудувати іншу власну структуру, де б не було потреби використовувати перетворення незмінюваних типів на змінювані, зайвих витрат пам'яті під збереження копій ключів (як у випадку використання `map`). Однак зрештою наведені розв'язання забезпечують цілісність даних.

Можна було б створити клас, який додає до якогось бібліотечного контейнера послідовностей операцію пошуку елемента, і на його основі побудувати структуру даних для задачі, що розглядалася. Наприклад, операція пошуку була б неоптимальною за часом виконання, але розмір вхідного файлу навряд чи дозволить відчути переваги ефективних за часом структур.

Дещо інший підхід до розв'язання задачі міг полягати в тім, щоб структура даних зберігала структури, кожна з яких містить інформацію з рядка вхідного файлу. Тоді використання пам'яті було б дуже неоптимальним (насправді для сучасних комп'ютерів на реальних даних для задачі про лабораторні роботи це не було б помітно зовні), а виведення інформації, особливо підсумкової по кожному студенту, вимагало б набагато більше зусиль на розробку й написання коду. Об'єктно-орієнтований підхід тут має очевидні переваги.

І на завершення: *кожен замовник завжди бажає за мінімальні кошти отримати якомога кращий програмний продукт.* Зменшення вартості розробки можна досягти, зокрема, використанням та/або пристосуванням наявних бібліотек до розв'язуваної задачі.

З погляду вартості розробки для задачі про лабораторні роботи найбільше підходять контейнери `set` і `map` (останній мінімізує зусилля з розробки тільки за використання відкритого успадкування). Оскільки у випадку використання контейнера `map` програма буде вимагати трохи більше оперативної пам'яті, то найкращим варіантом для задачі про лабораторні роботи слід визнати використання контейнера `set`.

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. Вступ до програмування мовою С++. Організація даних / Т. О. Карнаух, Ю. В. Коваль, М. В. Потієнко, А. Б. Ставровський. – К. : ВПЦ "Київський університет", 2015.

3. Мейерс С. Эффективный и современный С++ / С. Мейерс. – М. : Вильямс, 2016.

4. Прата С. Язык программирования С++ (С++11). Лекции и упражнения / С. Прата. – М. : Вильямс, 2014.

5. Саттер Г. Стандарты программирования на С++ : серия "С++ In-Depth" / Г. Саттер, А. Александреску. – М. : Вильямс, 2008.

6. Седжвик Р. Алгоритмы на С++. Фундаментальные алгоритмы и структуры данных на С++ / Р. Седжвик. – М. : Вильямс, 2013.

7. Страуструп Б. Программирование: принципы и практика использования С++ / Б. Страуструп. – М. : Вильямс, 2012.

8. Шилдт Г. С++ : базовый курс / Г. Шилдт. – М. : Вильямс, 2014.

9. Шилдт Г. С++: методики программирования Шилдта / Г. Шилдт. – М. : Вильямс, 2008.

10. International Standard ISO/IEC 14882:2014(E) – Programming Language C++ : [Електронний ресурс]. – Режим доступу : https://isocpp.org/std/the-standard.

Навчальне видання

Веклич Ростислав Анатолійович
Карнаух Тетяна Олександрівна
Ставровський Андрій Борисович

ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ С++

СТРУКТУРИ ДАНИХ

Навчальний посібник

Редактор *Н. Земляна*

Оригінал-макет виготовлено ВПЦ "Київський університет"

Формат 60x841/16. Ум. друк. арк. 5,8. Наклад 100. Зам. № 218-8783.
Гарнітура Times New Roman. Папір офсетний. Друк офсетний. Вид. № К2.
Підписано до друку 12.09.18

Видавець і виготовлювач
ВПЦ "Київський університет"
б-р Т. Шевченка, 14, кімн. 43, Київ, 01601
☎ (044) 239 32 22; (044) 239 31 72; тел./факс (044) 239 31 28
e-mail: vpc_div.chief@univ.net.ua; redaktor@univ.net.ua
http: vpc.univ.kiev.ua
Свідоцтво суб'єкта видавничої справи ДК № 1103 від 31.10.02